BRANDON RICHEY | RYAN YU
ENDRE VEGH | THEOFANIS DESPOUDIS
ANTON PUNITH | FLORIAN SLOOT

THE REACT WORKSHOP

GET STARTED WITH BUILDING WEB APPLICATIONS USING PRACTICAL TIPS AND EXAMPLES FROM REACT USE CASES

Packt›

WEB DEVELOPMENT

THE REACT WORKSHOP

Get started with building web applications using practical tips and examples from React use cases

Brandon Richey, Ryan Yu, Endre Vegh, Theofanis Despoudis, Anton Punith, and Florian Sloot

Packt>

THE REACT WORKSHOP

Authors: Brandon Richey, Ryan Yu, Endre Vegh, Theofanis Despoudis, Anton Punith, and Florian Sloot

Reviewers: Souvik Basu, Daniel Bugl, Brandan Jones, SudarshanReddy Kurri, David Parker, and Cihan Yakar

Managing Editor: Ranu Kundu

Acquisitions Editors: Anindya Sil, Sneha Shinde, Alicia Wooding, Archie Vankar, and Karan Wadekar

Production Editor: Shantanu Zagade

Editorial Board: Megan Carlisle, Samuel Christa, Mahesh Dhyani, Heather Gopsill, Manasa Kumar, Alex Mazonowicz, Monesh Mirpuri, Bridget Neale, Dominic Pereira, Shiny Poojary, Abhishek Rane, Brendan Rodrigues, Erol Staveley, Ankita Thakur, Nitesh Thakur, and Jonathan Wray

First published: August 2020

Production reference: 2040321

ISBN: 978-1-83864-556-4

Published by Packt Publishing Ltd.

Livery Place, 35 Livery Street

Birmingham B3 2PB, UK

Table of Contents

PREFACE

ABOUT THE BOOK

You already know you want to learn React, and the smartest way to learn React is to learn by doing. *The React Workshop* focuses on building up your practical skills so that you can create dynamic, component-based web applications and interactive React UIs that provide exceptional user experiences. You'll learn from real examples that lead to real results.

Throughout *The React Workshop* book, you'll take an engaging step-by-step approach to understand React. You won't have to sit through any unnecessary theory. If you're short on time, you can jump into a single exercise each day or spend an entire weekend learning about two-way data binding. It's your choice. Learning on your terms, you'll build up and reinforce key skills in a way that feels rewarding.

Fast-paced and direct, *The React Workshop* is ideal for React beginners. You'll build and iterate on your code like a software developer, learning along the way. This process means that you'll find that your new skills stick, embedded as best practice, a solid foundation for the years ahead.

AUDIENCE

The React Workshop is for web developers and programmers who are looking to learn React and use it for creating and enhancing web applications. Although the workshop is for beginners, prior knowledge of JavaScript programming and HTML and CSS is necessary to easily understand the concepts that are covered in this book.

ABOUT THE CHAPTERS

Chapter 1, *Getting Started with React*, gets you typing code immediately. You will learn the basics of React, as well as how to install and configure React projects with Create React App.

Chapter 2, *Dealing with React Events*, provides the starting point for creating interactive web apps with React by introducing events. We'll not only create events but integrate them into our React components.

Chapter 3, *Conditional Rendering and for Loops*, is where we expand upon programmatically creating React components either when conditions are set or when we need to add multiple React components as a list of items.

Chapter 4, *React Lifecycle Methods*, is where we go from passively relying on React to build our components to hooking up our components to the different lifecycle methods, allowing us to determine what logic to execute at each stage of our React component's life.

Chapter 5, *Class and Function Components*, discusses industry best practices to identify the component hierarchy and break the UI down into logical components. This chapter forms the basis for your understanding of creating UIs in React, be they simple or complex, and provides you with the basic tools required to build React applications.

Chapter 6, *State and Props*, discusses the components that use state and props. You will learn how to handle state in a React application and how to change state variables according to requirements.

Chapter 7, *Communication between Components*, helps you understand how to pass data between React classes and functional components. Also, this chapter dives into more advanced patterns such as higher-order components, render props, and the Context API.

Chapter 8, *Introduction to Formik*, gives you a complete picture of using Formik to build declarative forms. There are many ways to handle forms in React but Formik combines the best approaches.

Chapter 9, *Introduction to React Router*, gets you comfortable with one of the most commonly used React libraries in modern React web apps: React Router. We will cover the basics of using React Router, including implementing our own version to understand what is happening under the hood.

Chapter 10, *Advanced Routing Techniques: Special Cases*, looks at advanced techniques with React Router and gets you onto the next level. You will learn how to catch missing routes, how to nest routes, and how to protect routes from unauthorized accesses.

Chapter 11, *Hooks – Reusability, Readability, and a Different Mental Model*, prepares you for using the latest addition to React: hooks. We'll explore how to make your code more readable and more reusable via hooks.

Chapter 12, *State Management with Hooks*, builds on your knowledge of hooks to completely replace class-based component state management by exploring more advanced topics such as building your own state hooks.

Chapter 13, *Composing Hooks to Solve Complex Problems*, brings you from novice React hooks knowledge to expert React hooks knowledge, putting context hooks into the mix to completely eliminate the need for class-based components.

Chapter 14, *Fetching Data by Making API Requests*, shows you the various ways to fetch data by making API requests in React. This chapter also covers fetching data from servers using RESTful APIs, the Fetch API, and Axios and compares these methods.

Chapter 15, *Promise API and async/await*, takes a deep dive into the Promise API and async/await, which are essential techniques and the modern way to fetch data from the server.

Chapter 16, *Fetching Data on Initial Render and Refactoring with Hooks*, shows you the techniques of fetching data on initial render and fixing issues when a component falls into an infinite loop.

Chapter 17, *Refs in React*, introduces you to how to use references in React. You will be able to apply the knowledge gained from the chapter to implement references in different ways.

Chapter 18, *Practical Use Cases of Refs*, introduces you to some use cases of references and how to leverage their functionalities in your code. You will be able to identify scenarios in which to use references so that you can manipulate DOM elements directly.

CONVENTIONS

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The Promise API also comes with several methods, such as **`then()`**, **`catch()`**, and **`finally()`**."

Words that you see on the screen, for example, in menus or dialog boxes, also appear in the text like this: "Let's search for **`London`**, and the **`temperature`** should be displayed."

A block of code is set as follows:

```
"eslintConfig": {
  "extends": "react-app"
},
```

New terms and important words are shown like this: "Installing Node.js will also give us npm (the Node Package Manager)."

"**Callback hell** is an anti-pattern that consists of multiple nested callbacks".

Long code snippets are truncated and the corresponding names of the code files on GitHub are placed at the top of the truncated code. The permalinks to the entire code are placed below the code snippet. It should look as follows:

App.js

```
17   <div>
18    Username: <input type="text" /><br />
19    Password: <input type="text" /><br />
20    Password Confirmation: <input type="text" /><br />
21    Email: <input type="text" /><br />
22    <br />
23    <button>Submit</button>
24   </div>
```

The complete example code is available at https://packt.live/35qT2L3.

BEFORE YOU BEGIN

You are here and you are ready to get started with building your web applications in React. We need to install Node.js and Node Package Manager (NPM) to get started, and then additionally include installations for Yarn and Create React App. The installation instructions work for Windows, Linux, and Mac operating systems.

INSTALLATION INSTRUCTIONS

NODE.JS/NPM

To install Node.js/NPM, you will need to do the following:

1. Visit https://nodejs.org and visit the **Download** page.
2. From there, choose the appropriate package for your platform (or you can opt to download the source code and compile it yourself).

3. Verify the results. Open a Terminal or Command Prompt and verify that Node.js and NPM are installed with the **`node -v`** and **`npm -v`** commands, which should return the version installed for each.

> **NOTE**
>
> For the purposes of this book, we will be working with the "current" version, not the "LTS" version.

YARN

To install Yarn:

1. With Node.js and NPM installed, you can now visit **`yarnpkg.com`** and navigate to the **`Installation`** section.
2. From there, choose the operating system you're using and the version (the latest stable version will work just fine for us).
3. Follow the instructions on the page and verify the results by typing **`yarn -v`** in your Command Prompt/Terminal window.

CREATE REACT APP

To install **`Create React App`**:

1. We will be working with the **`npx create-react-app`** command, which does not require us to install **`create-react-app`** locally. Should you want to, however, you can install it locally either via Yarn or NPM.

For NPM:

```
$ npm install -g create-react-app
```

For Yarn:

```
$ yarn add global create-react-app
```

After that, verify the installation is working by running **`create-react-app -v`**.

EDITORS/IDES

You can download and use Visual Studio Code as an editor (https://code.visualstudio.com/). Visual Studio Code is a lightweight yet highly efficient source code editor that runs on your desktop and is available for Windows, macOS, and Linux operating systems. It provides you with built-in support for JavaScript, TypeScript, and Node.js. It is an easy tool to help you get started coding in React.

INSTALLING THE CODE BUNDLE

Download the code files from GitHub at https://packt.live/3cx4XdN. Refer to these code files for the complete code bundle.

If you have any issues during installation (or have any other feedback) you can reach us at **`workshops@packt.com`**.

1

GETTING STARTED WITH REACT

OVERVIEW

This chapter will explain the context and the reasons for using React framework and Create React App by utilizing the different options and scripts available for Create React App. The Create React App CLI tool will enable you to bootstrap a React project quickly. You will learn how to build a React component entirely with functional and class syntax using basic JSX. You will also be able to use components together to better structure your code. In this chapter, you will learn about the context and history of the React JS framework and be able to bootstrap your first React project.

INTRODUCTION

React is a frontend JavaScript framework. Its design principles take into consideration a lot of the common problems that developers used to face while building the user-facing components in their web projects. React doesn't exist to fix server problems or database problems, but rather to fix the common design and development problems of the frontend that exist in more complex and stateful web applications.

React is a framework that was created during the frenzy of frontend JavaScript frameworks. React by itself isn't just JavaScript that you can use to build elements on a page. After all, the problem of adding Document Object Model(DOM) elements to a web page has had solutions since the early days of Prototype and jQuery (or even further back than that). Instead, it is helpful to think of React as existing in a world that bridges the gap between JavaScript and HTML or, more specifically, the code and the browser.

Let's look at the problems that existed before the React JavaScript framework came into the picture.

PROBLEMS BEFORE REACT

Web development is a trade that can sometimes be incredibly confusing and complex. The act of getting the buttons, pictures, and text that we see on a website is not always a simple endeavor.

Think about a web page with a little form to log into your account. You have at least two text boxes, usually for your username and your password, where you need to enter your details. You have a button or link that allows you to log in after the details have been entered, with maybe at least one extra little link in case you forget your password. You probably also have a logo for this site you are logging into, and maybe some sort of branding or otherwise compelling visual elements. Here is an example of a simple login page:

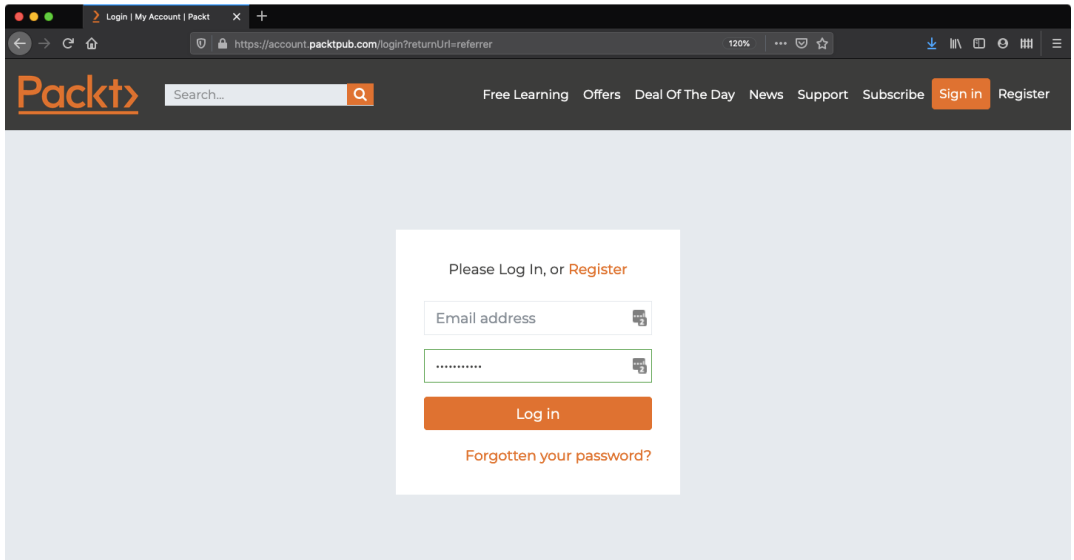

Figure 1.1: Login form

All of that doesn't just exist, though. A web developer needs to sit down and build each of these pieces and put them together in such a way so that the audience visiting that site can interact with each component individually. The real problem here is that all of that can be incredibly complicated to put together. For example, you might have multiple form inputs that all interact with each other depending on the values you entered and selected. Each piece needs to be able to interact with the next, and there are multiple different types of complex interactions that need to occur that go beyond just something happening as a result of a button being clicked.

For example, let's go back to our Packt login form example and examine some of the intricacies of interaction that exist even in such a simple and commonplace element. We will start off with the simplest example:

- Data needs to be entered into the username field.
- Data needs to be entered into the password field.
- The user needs to click the login button.
- The user is now logged in.

That's enough to make this work, but it's not a great user experience, is it? There's no validation, and nothing to help the user to get through the process smoothly or gracefully to let them know beforehand when the form fails to show the desired output.

So, we need to expand with additional use cases:

- If the user's username and password do not work, the user should see a failed login message.
- If the user fails to enter a username, the user should see a message reminding them to fill in the username field.
- If the user fails to enter a password, the user should see a message reminding them to fill in the password field.

These cases increase the complexity of the code pretty significantly.

Additionally, a few more elements for this web page are necessary:

- A message box displaying information if a username/password combination is incorrect, which evidently requires interaction with the server
- A message box displaying information if either of the username/password fields is left blank

The web page will get progressively more complicated as you move further along in improving the user experience.

For example, if we want to break it down further:

- What if we want to progressively check the username field to make sure it matches an email format if we are using emails for usernames?
- What if we want to validate whether each field has been filled (either for format or if the values have been filled) as we move through each of the fields, displaying a red box around input fields that are blank or skipped as we move along?

At each step, we are introducing new and increasingly complex levels of interaction between different UI elements and the collective state of each component.

Now, let's break the state of each component into:

- Their visual state (how the field is displayed to the user)
- Their data state (what is entered into the field)

- Their state in relation to other UI elements (how the login form knows about the data state of the username and password fields)
- Their interaction state (can the button be clicked if either the username or password fields are blank?)

Suddenly, you will realize that a few lines of HTML just will not cut it anymore. Now you need more complex blocks of JavaScript code, with state management both for visual and data states for each component, and a way to tie them all together.

You need to represent all of it in a way that won't require it to be rewritten every time someone needs to implement a similar form, and in a way that does not make a developer groan every time they have to touch the code. Let's see where React fits into all of this and how it addresses this problem.

INTRODUCING REACT

While the example in the previous section is a complex problem, the good news is that with React, the solution isn't terribly complex.

Instead of thinking about each element your browser sees and displays for the user as separate from your HTML code and JavaScript code (and thus separate from the states that each exhibits at any given point in time), React allows you to think of all of the code as part of the same data structure, all intertwined and working together:

- **The component**
- **The state**
- **The display (or render)**

This reduces a lot of the complexity and overhead while trying to amalgamate component state and component display with a mixture of CSS and JavaScript along the way. React represents a more elegant way to allow the developer to think holistically about what is on the browser and what the user is interacting with, and how that state changes along the way.

By itself, all of that is a great help to us as developers but React also introduces a full scaffold for setting things up quickly and easily in a way that lets us get to development faster and fiddle less with configuration along the way.

React introduces a way for us to set up how to represent the states and display of components, but additional tools make the setup and configuration process even easier. We can further optimize the process for getting started by introducing a special command-line tool for React projects called Create React App.

This tool minimizes the amount of time a developer needs to spend getting everything set up and allows developers to instead focus on the most important part of building your web application: **the development**.

GETTING STARTED WITH CREATE REACT APP

Create React App introduces a scaffold to React applications. Through this, configuration and setup are minimized around opinionated configuration (a set of configurations where a lot of the decisions have been made for you) and pre-built structures. Your directories, input, and output are all handled for you.

What is a scaffold? A scaffold in the development world is something that sets up boilerplate (that is, often-repeated bits of code or configuration) details for you with the idea that you will be using the same configuration, setup, and directory structure that the project is using. A scaffold allows you to get to development significantly faster, with the drawback that it may be harder to break away from decisions made in the scaffold later on in your development process, or that you may end up with more dependencies or structure than you need.

The Create React App scaffold is designed around the idea of the developer experience: that setting up a new project should be seamless and painless. If you want to start developing right now and not have to worry about how you are going to test your application, or how you are going to structure your application, or what libraries you are going to include in your application, then a scaffold such as Create React App is the way to go.

Try to think of a time that you had to do something that required a lot of setup (baking a cake, painting a picture, exercising, or whatever). Think of a scaffold you could use to bootstrap that process and speed it up for the next time that you want to repeat that activity. What would you do? What stuff would you include with it? Are there any parts of the process that might differ from person to person (these would indicate "opinionated" scaffolds or frameworks)?

Since we are trying to minimize friction at the starting point of our project, we are going to use this scaffold to get up and running quickly and painlessly.

SETTING UP A DEVELOPMENT ENVIRONMENT

Having digested the context of React, Create React App, and the problem at hand, it's time to start diving into getting everything ready for us to start writing our code. To do that, though, we will need to make sure that our development environment is ready for us to start using Create React App.

The first thing we will need to do is install Node.js (instructions can be found in the *Preface* of this book) to make sure we can use the command-line tool to create new React projects, and to make sure we can do so in a best-practices manner.

If you visit https://packt.live/34FodRT, you will see download links for your platform front and center, with a choice to either download the *LTS version* or the *current version*.

> **NOTE**
>
> LTS is the long-term support version of the Node.js runtime. This is designed more for things that are running on the server side in production environments and specifically need to not change over time, whether for security or stability reasons.

We will opt for the *current version* just to make sure we are using the latest and greatest for everything React along the way.

You will then need to follow the instructions to download and install Node.js via the installer. Each platform has a different way to get everything properly installed and set up, but for the most part, it's either a simple installer or a simple package-download to get started.

After getting everything installed, your next step should be to open a command-line/terminal window to verify that you can run Node.js. If you run the command **`node -v`**, you should see the version you just downloaded and installed:

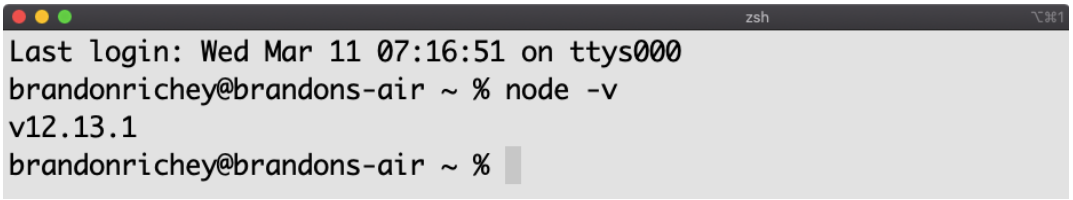

Figure 1.2: Verifying whether Node.js is installed

Installing Node.js will also give us npm (the Node Package Manager) as well, which we will also want to verify with the command **`npm -v`**:

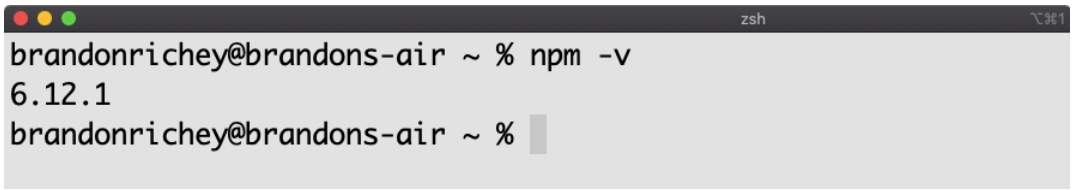

Figure 1.3: Verifying the version of Node.js

We can now start using another utility, **`npx`**, to execute Create React App and get started quickly.

> **NOTE**
>
> **`npx`** is a utility that executes **`npm`** package binaries (such as Create React App) with a minimal amount of fuss.

Instead of just running **`create-react-app`**, let's take a look at the help documentation instead with the **`--help`** flag in our call to **`npx`** .The output will be as shown below (the output will slightly vary for different operating systems):

```
$ npx create-react-app --help
Usage: create-react-app <project-directory> [options]
Options:
  -V, --version              output the version number
  --verbose                print additional logs
  --info                   print environment debug info
  --scripts-version <alternative-package> use a non-standard version of
react-scripts
  --use-npm
  --use-pnp
  --typescript
  -h, --help             output usage information
  Only <project-directory> is required.
A custom --scripts-version can be one of the following:
  - a specific npm version: 0.8.2
  - a specific npm tag: @next
  - a custom fork published on npm: my-react-scripts
  - a local path relative to the current working directory: file:../
my-react-scripts
```

```
  - a .tgz archive: https://mysite.com/my-react-scripts-0.8.2.tgz
  - a .tar.gz archive: https://mysite.com/my-react-scripts-0.8.2.tar.gz
It is not needed unless you specifically want to use a fork. If you have
any problems, do not hesitate to file an issue: https://github.com/
facebook/create-react-app/issues/new
```

> **NOTE**
>
> Take a look at the output from the help page to figure out what you need to pass to **`create-react-app`**.

Let's check the React app version using the command **`--version`**:

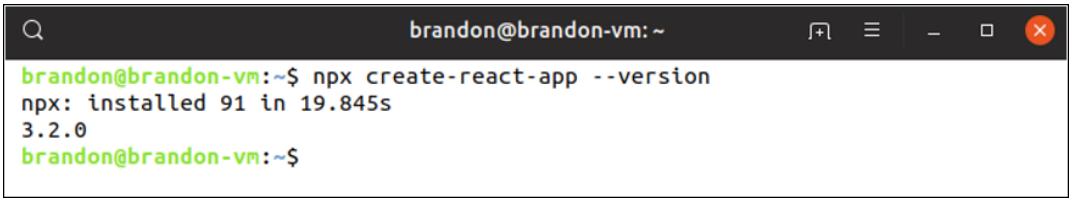

Figure 1.4: Checking the Create React App version

In the next section, we will discuss how we can tweak the options in Create React App.

TWEAKING CREATE REACT APP'S OPTIONS

We need to dive a little bit deeper to better understand the different options that are available when we are using Create React App to generate our new React project structure. We can use it with no options and only declare the project name, but if we wanted to tweak our default configuration at all (for example, if we wanted to use a specific version of **`react-scripts`** or use something like TypeScript with our project), we would need to use one of the other options available to us.

We already talked about getting the version, but we can also pass some additional flags to change what we see or change how our project structure is generated along the way:

If we pass **`--verbose`** (by running **`npx create-react-app --verbose`**), for example, we will see a lot more information (and more detailed information) about each step along the way as the project is created and structured. You will want to use something like this if your project has the chance of running into issues during the creation step and you need to figure out, in greater detail, what's happening along the way. You can use this and **`--info`** to really understand all of the things that might be influencing the creation process if you are running into any issues.

We have the **`--scripts-version`** flag, which can tell Create React App to use a custom version of react-scripts, which is a helpful collection of scripts, dependencies, and utilities that facilitate the React app creation process. For the purposes of our first projects, however, we will leave this alone for now.

The **`--use-npm`** and **`--use-pnp`** flags exist because by default, Create React App will use **`yarn`** as its default package manager. If you pass in the **`--use-npm`** flag, Create React App will avoid the dependency on **`yarn`**, similar to the **`--use-pnp`** flag. Again, we are fine with the default settings here, so we will not be using either of those flags in our early projects.

Finally, the last non-help option we can pass is **`--typescript`**, which generates a Create React App scaffold with TypeScript hooked in appropriately (more on this in detail in the following chapters). You might use Typescript if you are looking to implement more development structure or stricter adherence to types in your React projects.

USING CREATE REACT APP TO CREATE OUR PROJECT

It's finally time to roll up our sleeves and get to work. You should have a very thorough understanding of Create React App's offerings, but now it's time to actually put that understanding to use.

Again, remember that Create React App creates a project for us, and that includes a lot of extras that act as the main scaffolding for our project. Much like a painter's scaffold, from far away it can seem like a lot of things have been added just to begin working, but the beauty is that it is all work that we would have to do anyway. This allows us to bypass that step and go right into getting some work done!

We will create our first project, which we will call **`hello-react`**, by running the following command:

```
$ npx create-react-app hello-react
```

The output will be as follows(truncated output):

```
npx: installed 91 in 23.129s
Creating a new React app in /home/brandon/code/hello-react.
Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts...
…
├─ react@16.10.2
└─ scheduler@0.16.2
Done in 151.99s.
```

```
Done in 49.76s.
Initialized a git repository.
Success! Created hello-react at /Users/brandon.richey/Development/react-book-2/chapter1/code/hello-react
Inside that directory, you can run several commands:
yarn start
Starts the development server.
yarn build
Bundles the app into static files for production.
yarn test
Starts the test runner.
yarn eject
Removes this tool and copies build dependencies, configuration files and scripts into the app directory. If you do this, you can't go back.
We suggest that you begin by typing:
cd hello-react
yarn start
Happy hacking!
```

This will give us our initial project, scaffolding and all, and allow us to jump right into coding.

Before we do that, we can see the output from our **`npx create-react-app`** command that tells us some of the other scripts we can run via **`yarn`**: **`start`**, **`build`**, **`test`**, and **`eject`**. We will be using **`start`**, **`build`**, and **`test`** the most, so we will focus on those and skip **`eject`** for now.

Each of the commands is broken down into a very specific and common scenario of application development and maintenance:

- **`start`** spins up a server listening locally on your computer on a specific port, where it watches for any changes made in the project, and then, when those changes are made and saved to the filesystem, it automatically reloads the appropriate project files. This gives you the ability to see changes that you make in your code as you save each file and make tweaks and modifications as necessary along the way. This is something that by itself could take a lot of time and effort to set up, so we are using Create React App just to avoid having to do this every time. Almost immediately, this makes the app scaffold worth it.

- Next, we have **`build`**, which is used to produce an optimized production build of our code. This takes all of the project's code (not just the JavaScript, but other file types, such as CSS) and optimizes as much as possible for use in a production environment. This includes minification (the process of stripping code files down to be as small as they possibly can be) and any other major optimizations that allow the code to download faster and execute quicker when it's out and serving real production traffic.
- Finally, of the most common commands, we have **`test`**. Now, **`test`** will run all of the tests for your code and display the results in an easy-to-read manner. It will also automatically run unit tests for each code change, including just the tests that failed when related code was modified, resulting in a broken test suite.
- **`eject`** is not often used but is invaluable when you start hitting certain limitations of the Create React App scaffold. **`eject`** is a command that pulls the covers off all of the configuration and scripts behind the scenes in your project, allowing you to edit and tweak any configuration detail that you will like. This also removes the safety net of the scaffolding, so it becomes easier to accidentally change something in your project and potentially break things or negatively impact the reliability of your production builds. Also, when you run this command, there is no turning back, so this is definitely something that you only want to do when you are good and ready and understand the potential impact on your project.

Now, let's start exploring the crated project.

EXPLORING THE CREATED PROJECT

We have run the command to generate our project, but now it's time to take a deeper look and understand a little more of the scaffold and what we have been provided with. Here is an example of the files that are automatically generated in a new project:

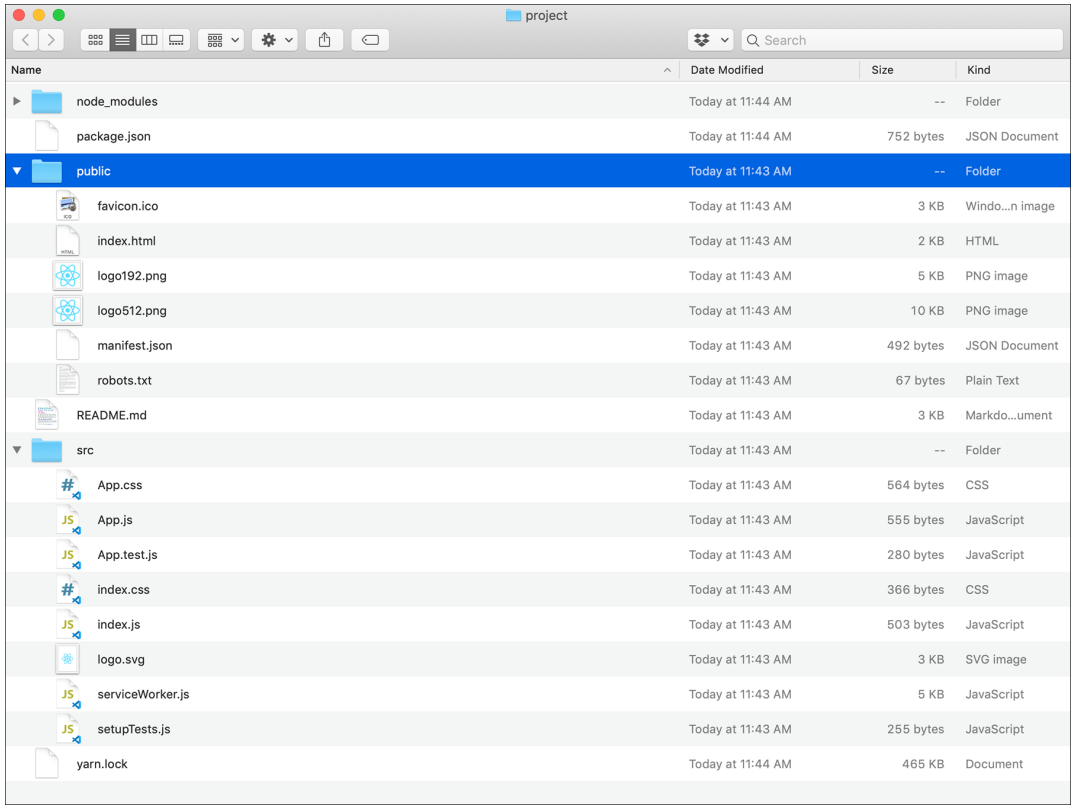

Figure 1.5: Files generated in a new react project

We will start off by just exploring the top levels and then look more closely at any noteworthy individual files along the way. You'll find that even the starter project includes quite a few files, and while we will not go into extreme detail on each individual file, we will talk a little bit about how they all fit together overall.

README.md

The first notable file, right in the project root, is our **`README.md`** file. This is a *markdown* file that tells people how to use and interact with your project. Usually, this should include instructions, a brief synopsis of the project, how to start the project, how to run the tests, and any additional dependencies or common catches that someone will need to know to use the code base you have created. Think of this as the first official piece of documentation for the project that you are building.

Deep dive: package.json

The next notable file in the root directory is the **`package.json`** file. **`package.json`** exists to tell Node.js more details about this project. This could be anything from metadata about the project (name, description, author, and more) to dependency lists.

Let's take a look at a sample generated package file:

`package.json`

```
1  {
2  "name": "hello-react",
3  "version": "0.1.0",
4  "private": true,
5  "dependencies": {
6  "react": "^16.8.6",
7  "react-dom": "^16.8.6",
8  "react-scripts": "3.0.1"
9  },
```

`The complete code can be found here:` https://packt.live/2y121qR

Here, we have some notable keys and some more metadata-style keys combined together:

- We have a **`name`** key, which is basically just the name of your project itself.
- Next, we have the **`version`**, which should be used to indicate the semantic versioning for your project.

> **NOTE**
>
> Semantic versioning is a numbering system used for a large number of software projects. It typically follows a numbering scheme where the numbers, separated by periods, indicate the major version, the minor version, and the patch version. In our example, **`0.10.0`** indicates a major version of **`0`**, a minor version of **`10`**, and a patch version of **`0`**. The major version is used for major rewrites or backward compatibility-breaking changes, the minor version is used for large changes that may not dramatically affect backward compatibility, and the patch version is typically used for tiny fixes, usually hotfixes or security updates, that don't affect backward compatibility at all.

- We also see a **`private`** flag, which tells **`npm`** (our package and dependency management system when using Create React App) if our project is **`public`** or **`private`**. If it is **`private`**, then it is not published to **`npm`** with any sort of public visibility.

> **NOTE**
>
> When **`private`** is set to **`false`**, the project instead gets public visibility.

- Moving on, we have our **`dependencies`** section. The **`dependencies`** list is critical to your project, as this is what is used to tell Node.js what other libraries your project depends on. This could be major dependencies, such as React in our example, or sometimes less critical sets of dependencies such as utilities to make writing your tests a little easier. Here's the **`dependencies`** section of our example:

```
"dependencies": {
  "react": "^16.8.6",
  "react-dom": "^16.8.6",
  "react-scripts": "3.0.1"
  },
```

If you look at our **dependencies** list, you'll just see three entries in there: **react**, **react-dom**, and **react-scripts**. Each of those dependencies includes their own sets of dependencies, and in the case of **react-scripts**, a lot of those dependencies are related to the functionality for example, live changes of development server, auto reload of those changes. Create React App provides functionalities which are found in one of those tucked-away dependencies.

- Next, we have our list of **scripts**:

```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject"
  }
```

These are commands that you can run via **npm** or **yarn**. Create React App provides the four that we have already talked about: **start**, **build**, **test**, and **eject**. Each of these can be executed either via **npm run [script]** or **yarn [script]**. If you wanted to add your own (for example, you can add something like a clean script to clean out temporary files during development and testing), you would just add it there.

- Moving on, we have our **eslintConfig** setting, which tells our code linter to use **create-react-app** style linting:

```
"eslintConfig": {
  "extends": "react-app"
},
```

> **NOTE**
>
> Code linting is the process of looking through the code files in your project and highlighting things that may violate best practices, either for security or developer sanity. You may find things such as whitespace violations, problems with naming conventions, or a myriad of other possible things to highlight and warn against. This is a great thing to include in your project (or alternatively, you can use something like Prettier, an automatic code formatter, instead).

- We also have **`browserslist`**, which tells Create React App which browsers to target for our projects to work with:

```
"browserslist": {
  "production": [
  ">0.2%",
  "not dead",
  "not op_mini all"
  ],
```

  The list is broken down into two subsections that specify for the build process whether we should be building things for production or for development. These are relatively sane defaults and don't tend to be modified or tweaked very often.

We have a pretty good idea of the **`package.json`** file and its contents, so we can safely move on to exploring the rest of the scaffold.

> **NOTE**
>
> **`yarn`** and **`npm`** are two different package managers in the JavaScript world. **`yarn`** sits on top of **`npm`** and installs **`npm`** packages. However, **`yarn`** manages its dependencies differently and adds those dependencies to your project differently than **`npm`** typically does. These are almost entirely internal differences that you typically do not need to think about during the build process, but that is why you see both **`yarn`** and **`npm`** dependencies listed in your Create React App project.

EXPLORING THE REST OF THE SCAFFOLD

`yarn.lock` provides a way to lock in the dependencies of our project, preventing a scenario where major changes might be introduced in the production stage due to an upgrade to the library version.

We also have a hidden file hanging out in our scaffold, **`.gitignore`**. The **`.gitignore`** file tells Git what files to ignore when you are committing to avoid them showing up when other people clone or fork your repository. Typically, you will use this to hide files that contain any sort of secrets or keys, or anything else that you don't want committed into the project's repository (**`node_modules`**, for example, is typically ignored).

We also have the **`node_modules`** directory, which stores all of the Node.js dependencies for your project; everything your project needs to run will go here. **`create-react-app`** has a lot of dependencies that are included for running the project for different environments and settings that allow you to use the latest and greatest in JavaScript syntax additions. This is typically ignored from committing to the code base via an entry in **`.gitignore`** (more on that later).

There is the public directory, which includes all the files that you will want to interact with after the build process is completed. This is a good place to put certain types of assets, such as images, movies, or any kind of documents or downloads you'd want to include in your project. For example, our **`favicon.ico`** file would be found here, along with an **`index.html`** file, which is used to pull in the built application when served from that directory, and the **`manifest.json`** file, which is used to describe Progressive Web Apps. Progressive Web Apps are apps that can be downloaded and used offline as well and may have faster startup times if cached appropriately by your browser.

Finally, we have the directory we are going to spend the most time in: **`src`**. Now, **`src`** contains all of your actual application code. All your React code will be located here, which is then built and output into **`public/`** after being combined, minified, and turned into files that make it easier for the browser to download, consume, and interpret the project you have built, all in the name of further optimization.

With all this knowledge, we need to now understand how to code an element or a component of a web page in React.

INTRODUCTION TO JSX

To be able to write components in React, we need to understand the primary templating language that React uses to build a component: JSX. The best way to describe JSX is as a type of hybrid between HTML and JavaScript.

JSX represents the easiest way to break down the boundaries between the HTML that you use as a basic markup for your web page, basically the UI of your app and the JavaScript that you need to make your app interactive. Ultimately, the JSX gets turned into JavaScript function calls behind the scenes, which enables you to write your components in a way that is comfortable and familiar for people used to working with writing HTML for a browser.

You can use most of the standard HTML tags and syntax that you already know (with a few notable exceptions), but you can also intersperse JavaScript into your templates to better approach laying things out programmatically. The idea is that if you can read HTML and JavaScript, you should be able to read JSX immediately.

For example, the following code is valid **JSX**:

```
<div>
  <strong>Hello {"World"}</strong>
</div>
```

The only real new thing in there if you are expecting to just see HTML is the curly braces, which indicate some JavaScript code that is to be executed and returned as a value.

DIVING DEEPER INTO JSX

It's easier to take each part of JSX apart piece by piece to make it easier to write our own React components, so let's do exactly that.

As mentioned before, JSX shares a lot of similarities with both JavaScript and HTML syntax, so a large number of tags that you are expecting to use (such as the block provided in the last example) will work just fine.

Any HTML tag that isn't a React component will work exactly the way you would expect it to: **strong** tags give you bold, **header** tags give you header text, and so on. It gets a little more complicated when you move down into things such as script tags or the various attributes in HTML, but we will talk about that a little further on.

So, if you want to use HTML inside a React component, you just use the tag you will expect.

** Hello World ** does the same thing in React and HTML. As mentioned before, things get trickier with the class attribute in HTML. Since **class** is a reserved word in JavaScript, we instead have to use **className** as a substitute:

```
<div className="greeting">Hello World!</div>
```

Moving forward, what if we want to declare styles on a particular HTML element? This gets a little trickier, since it requires you to write a little bit of JavaScript inside of your element. We can't just declare our style as a string inside of our element, as in the following example:

```
<div className="greeting" style="color:red;">
  Hello World!
</div>
```

Instead, you will need to embed a JavaScript object inside the **`style`** attribute that returns whatever CSS properties you need to include, as in the following:

```
<div className="greeting" style={ { "color": "red" } }>
  Hello World!
</div>
```

Following this example to its logical conclusion, what if we want to declare something else, such as an **`id`** attribute? We would follow the same rules we've been following along the way.

But what about those curly braces, **`{{ "color": "red" }}`**, that show up in the code? Those curly braces denote JavaScript code that should be evaluated and embedded inside your JSX code. In the preceding example, we can't represent an object in HTML or JSX without a little help, so we rely on JSX to interpret the return of the statement inside the curly braces and then turn it into something the browser can understand.

Now that you have learned a little bit of JSX, let's put it together.

EXERCISE 1.01: IMPLEMENTING JSX

In this exercise, we will use JSX to create a popup that will show a **`hello`** message. To do so, let's go through the following steps:

1. Create a new **`div`** element in JSX. Give it a CSS class of **`Example`** and give it an **`id`** of **`my-element`**.

   In **`App.js`**, remove the old **`div`** element and add the following code in its place:

```
<div className="Example" id="my-element">
</div>
```

2. Using the **`style`** attribute,
3. give the **`div`** element a black background and white text:

```
<div
  className="Example"
  style={{ background: "white", color: "white" }}
>
</div>
```

4. Next, fill the body with the text of **`Hello World`**:

```
<div className="greeting" style={{ background: "black", color:
"white" }}>
  Hello World
</div>
```

5. Finally, add an **`onClick`** handler that will display a **`hello`** alert when the **`div`** is clicked on:

```
<div
  className="greeting"
  style={{ background: "black", color: "white" }}
  onClick={() => alert('hello')}
>
Hello World
</div>
```

If we open up a browser and point it at the code for the exercise (which should be happening automatically if you are running your project via **`npm start`**), we will see this:

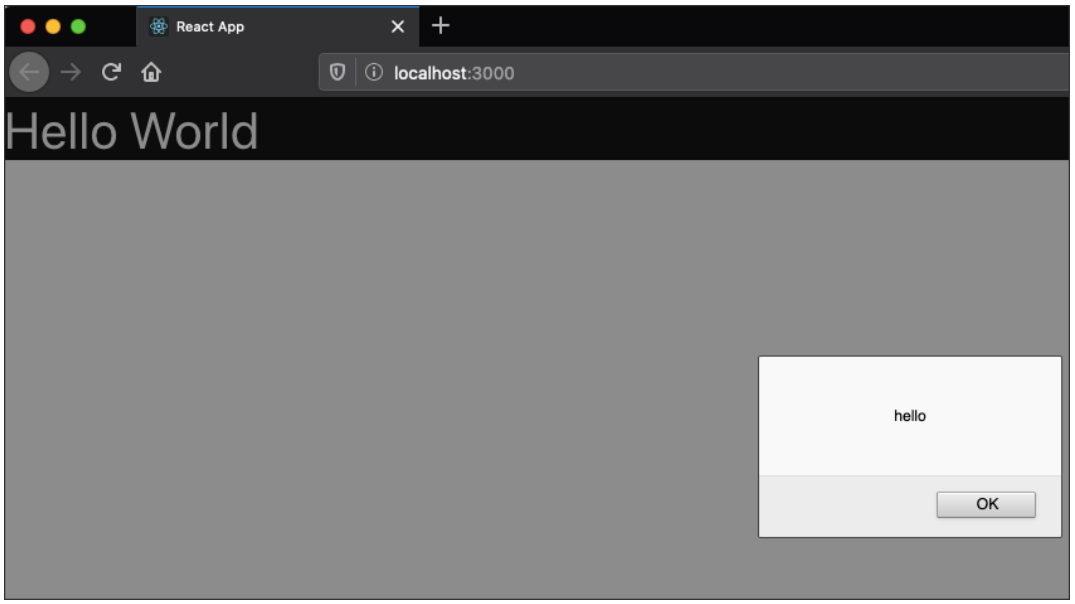

Figure 1.6: Implementing JSX

Think about one of the main uses for JavaScript in a web application: we need to be able to interact with the application, right? In fact, a pretty big part of any modern web application nowadays is how the DOM and the JavaScript interact with each other in rich, seamless ways. So, by adding an **`onClick`** to the **`div`** tag, we can provide a little bit of JavaScript interactivity into our React component (as you can see in the exercise).

As we have seen a few times before, there isn't 100% parity with HTML, but it's very close (at least close enough that you can read it pretty easily). Instead of onclick (all lowercase), we need to write it as **`onClick`**. In addition, you cannot just write the JavaScript for the **`onClick`** handler without wrapping it in another function. Without wrapping it inside a function, the code will instead execute immediately instead of waiting until the user actually clicks on it.

CREATING A REACT COMPONENT

Now that we have a strong grasp of JSX and React, we have everything that we need to be able to start building out our own React components. We now need to cover the work of actually writing a component. When creating new components in React, you have a few options that are commonly used, and they each serve slightly different purposes.

Sometimes you need a small, lightweight declaration of a component to just display something simply and easily where you are not worried about a lot of state modification or user interaction. The easiest way to declare a component in React is with functional syntax.

PREPARING OUR CODE TO START WRITING OUR COMPONENT

Remember that when creating a new project with **`create-react-app`**, we start off with a bunch of starter components and a lot of extra code pre-written for us. The best course for us is to wipe out the starter **`Create React App`** code and begin rewriting it with our own component. We will start off with a simple functional component and then move on to the **`class`** syntax for declaring more complex components. Let's take a look at the list of all of the files that we get in **`src/`** when we create a new app:

```
src/
  App.css
  App.js
  App.test.js
  index.css
  index.js
  logo.svg
  serviceWorker.js
  setupTests.js
```

This is all great, but there is a lot of code in these that we don't really need, and it will make our lives a little easier to keep things managed if we start clearing things out. We do not quite want to delete everything since some of the boilerplate code is actually pretty useful, but some of the files we also very much don't need.

First off, delete **`src/logo.svg`**, since we don't need that file at all anymore. We will leave **`src/index.css`** and **`src/index.js`** alone, as well as **`src/serviceWorker.js`**. Anything outside of **`src`** We will just leave alone as well. You will also want to remove any references to the files you deleted, like **`src/logo.svg`**, from **`src/App.js`**.

The next step is for us to delete the CSS file for the **`App`** component: **`src/App.css`** since we are not going to use it.

UNDERSTANDING PROPS

Props is just a shorthand name for properties, and you can think of them similarly to how you will think about passing in HTML attributes to modify the behavior of an HTML tag. Passing in props allows components to take in information along the way and change how the component is rendered to the end user.

An example of passing props to a component is moving the **`Greeting`** from the code snippet used in *Exercise 1.01*, *Implementing JSX* into its own component and then including that in our base **`App`** component:

```
<div
  className="greeting"
  style={{ background: "black", color: "white" }}
>
```

We will declare this new **`Greeting`** component in the same **`src/App.js`** file, and then we will pass down a **`name`** prop to the **`Greeting`** component. In our **`Greeting`** component, we will have it display the **`Hello React`** message from before, but we will replace the **`React`** message with the passed-in **`name`** prop. Let's see an example of how to implement props in React.

First, the code for our **`Greeting`** component will change to this:

```
const Greeting = props => <p>Hello {props.name}!</p>;
```

Next, we will need to use this new component in our **`App`** component and pass the new props down. Let's head back to our previous multiline component and modify it to instead call the **`Greeting`** component and pass a **`name`** prop:

```
const App = props => (
<div>
  <h1>My App</h1>
  <br />
  <Greeting name="User" />
</div>
);
```

Now, save and reload your app (which usually happens automatically), and you should see this:

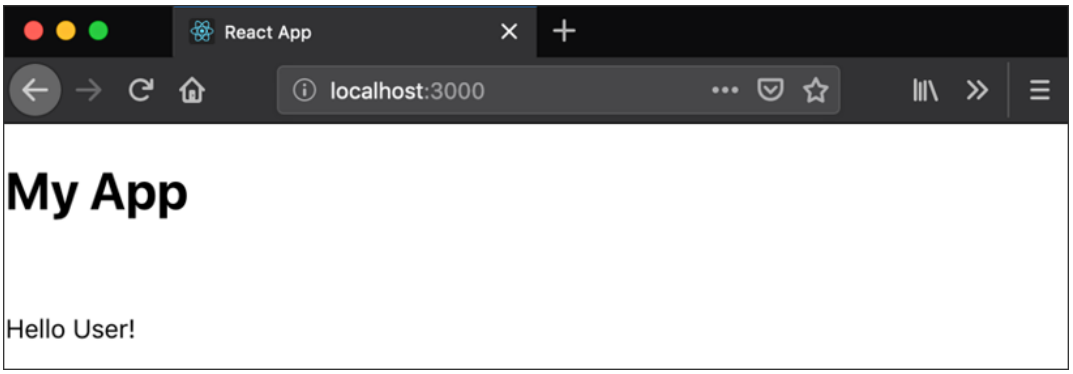

Figure 1.7: First React component created

Let's explore props with an exercise.

EXERCISE 1.02: CREATING OUR FIRST REACT COMPONENT

In this exercise, we will clean up the **`src/App.js`** file. We will follow a similar path as mentioned in *Exercise 1.01*, *Implementing JSX* by opening up **`src/App.js`** and replacing its contents with the component we will be building in this exercise:

1. Add an **`import`** statement and an **`export`** statement to the top and bottom of the file, respectively, to include the React library and export the component (named **`App`**, like the filename) as the default export:

```
import React from 'react';
export default App;
```

   There are a few things here we need to talk about before we start building out the component in full, though.

   First off, we need to import the **`React`** library into our file so that we can actually start building out our component. Without this statement, the rest of the component building would fail as JavaScript and Node.js would have no idea how to interpret the code.

   The next line will end up actually being the last line of our code file; this allows JavaScript to import code in this file into other files in a clean way. We have not actually created the app that we are exporting, but don't worry about that. We will have that built in just a minute.

2. Next, we will want to build out the initial component. This will start off very simple and should just be a **`div`** with the text **`Hello React!`** in it:

```
import React from 'react';
const App = () => <div>Hello React!</div>;
export default App;
```

   As mentioned back at the start, there is a lightweight method of declaring your components in React, and that will be the strategy that we use to start writing this all out. Declaring a functional component in React is incredibly simple since it is just a function declaration that returns some JSX. In fact, when we get right down to it, there is not really any other major difference. So, in our code, we are going to create a new function called **`App`**, and we are going to have it return a **`div`** with some **`Hello React`** code in it.

We will have a component that looks like this when the browser refreshes:

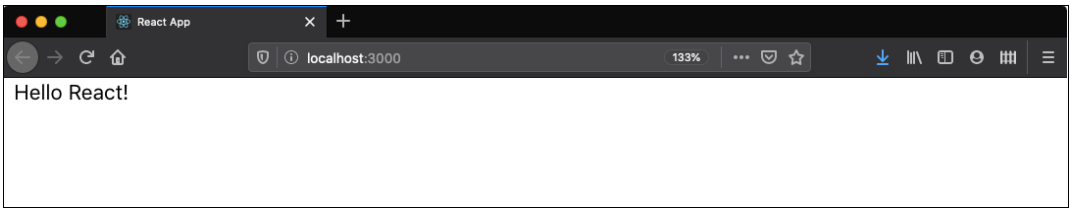

Figure 1.8: Simple component in React

By default, this gives us a pretty basic component with no frills, and if you are aiming to just create a quick and easy component used to display something, this is probably what you would use. There is also an option to pass in props to functional components by passing them in as a function argument, which would change our function to this.

3. Let's write the following code:

```
const App = props => <div>Hello React!</div>;
```

This would allow you to use this with a parent component, for example, that is passing details down to a child (but we will talk about that a little more later). You can also use standard function syntax if you prefer that.

This is the ES6 arrow function syntax:

```
const App = props => <div>Hello React!</div>
```

and it becomes:

```
function App(props) {
  return <div>Hello React!</div>;
}
```

The arrow syntax is much cleaner and is more preferable. In addition, if you have a component that spans multiple lines, you will need to wrap your JSX inside parentheses instead.

4. Let's expand our component from the previous example and add an **`h1`** header to the app, in addition to our **`Hello React!`** message:

```
import React from 'react';
const App = props => (
<div>
  <h1>My App</h1>
  <br />
  Hello React!
</div>
);
export default App;
```

The resulting component should look like this:

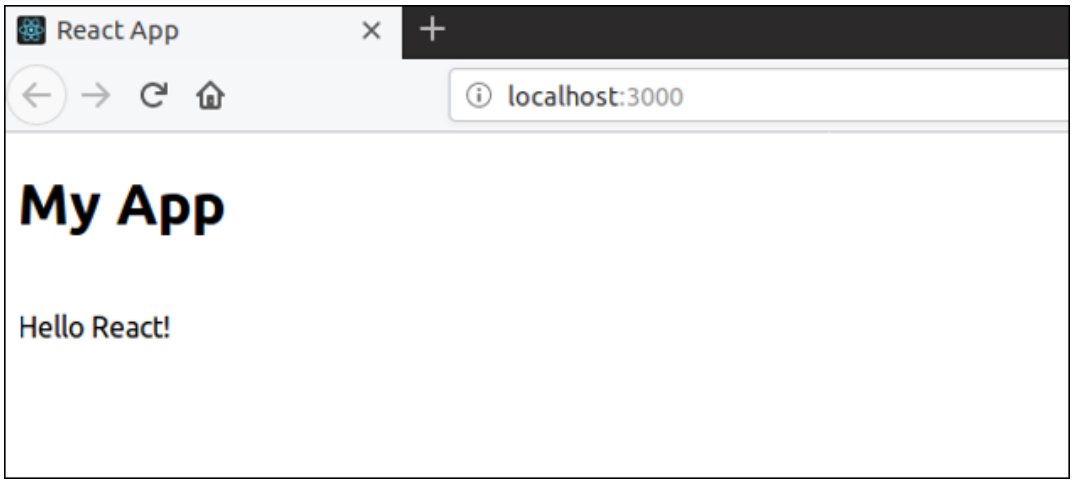

Figure 1.9: React component

With all this knowledge, let's move onto the next section.

UNDERSTANDING REACT RENDERING

We have created two separate React components here but only with a cursory understanding of what is happening behind the scenes, so it's worth spending a little time unpacking what is happening in these functional components, especially in relation to the concept of rendering in React.

The reason that just returning JSX in a functional component works is that ultimately, anything returned by a function is a valid target for React to render if it is called from React code. It is pretty easy to see in the **`Greeting`** function how this plays out: when you use the JSX tag for a function, it basically turns whatever the function returns into usable JSX code. If we return JSX in our function, then it all gets rendered appropriately.

Let's also explore what happens higher up in the React call stack when rendering is involved. It's one thing to understand rendering at the level of the components we write, but we should also understand how those get from the code we write to the actual browser page.

Back in **`src/index.js`**, one of the files that we did not modify, if you look through it, you will see a call to **`ReactDOM.render()`**:

```
ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

`ReactDOM` is a library that provides interaction with the DOM layer of the browser, and we use it specifically to place our React components into the browsers, available to be viewed and interacted with by the user.

In this case, we are taking our **`App`** component, represented by the JSX code **`<App />`**, and we are rendering the output of that (which must be valid JSX code) to the browser, converting everything from JSX into code the browser can understand and interpret.

This is why our functional component needs to return JSX, and why (as you will see later) our class-based components need to return JSX in their render function calls. If we return something JSX cannot interpret, then we will get an error message instead.

Verify the results of the preceding statement by changing the **`Greeting`** function to instead return anything else that's non-JSX and verify what happens for each.

> **NOTE**
>
> **`StrictMode`** helps to spot potential errors in advance. It performs code checks only during development time. It does not work in the production run.

BUILDING COMPONENTS WITH CLASSES

You can build your components using functional syntax, but there is another way to declare your React components that allows better state management in your components. You will be introduced to the basics of using classes. This topic will be covered in more detail in the later chapters.

In addition, the functional components of React have no real sense of utility functions or internal state; everything must be represented via props or not at all.

Instead, let's shift our focus to the other common method of building out React components: using classes. We will want to add one more addition to our React **`import`** statement at the top of our file because not only do we now want to import React, but we would also like to include the **`Component`** class from the React import.

EXERCISE 1.03: IMPLEMENTING A CLICK COUNTER IN REACT USING CLASSES

In this exercise, we will create a click counter for a web page. Here, we will convert the component we built previously as a functional component into a class component. Perform the following steps:

1. First, change the **`import`** statement to also include **`Component`** from the React library.

   We can import both at the same time with the following line:

   ```
   import React, { Component } from 'react';
   ```

   This tells us that we are going to import the React library as a whole, but that we also specifically want to pull **`Component`** out from the named exports.

   This gives us everything we need to start modifying our **`App`** component to instead be a class-based component. We will start off by writing out the class definition and then build it up as we go along.

2. Next, convert the functional declaration of the component to a class declaration that extends the **`Component`** import:

```
class App extends Component {
}
```

   Here, we define what was previously our functional **`App`** component as a class-based component (still called **`App`**).

   The next bit of this uses that **`Component`** named import that we specified at the top of the file. The extends keyword here tells JavaScript that our **`App`** component has a parent class, **`Component`**, and that its behavior should act as an extension to that parent class.

3. Next, add a **`render`** function, since we can't just return JSX in a class directly, and include what used to be the body of the component before:

```
render() {
return (
  <div>
  <h1>My App</h1>
  <br />
  <Greeting name="User" />
  </div>
 );
}
```

   This is mostly identical to our functional component, with the addition of needing to specifically declare the render function itself (without this, React doesn't know how specifically to deal with this class and render it out appropriately via JSX). We also need an explicit **`return`** statement since we are not using arrow syntax here, but otherwise this function remains identical.

   Now, what if we wanted to go deeper? As mentioned previously, a critical reason to use this style of component declaration is the ability to manage state, and if you look at our **`render`** function, we are also missing any references to props.

   The easiest way to define our class component in a way that allows us access to both state and props is to define a constructor function. The **`constructor`** function acts as the way for JavaScript to instantiate our new class (turn it from a class definition into an object) and set up any sort of default or initial values or state that we may need to think about.

4. Define this by creating a new function in our class called constructor and pass a **`props`** argument. You will also need to include a call to the constructor via **`super`** and pass in the **`props`** argument:

```
constructor(props) {
  super(props);
}
```

We can flesh this out even further. For example, maybe your class has some sort of code-configurable option to it that you do not necessarily want to expose to the user as part of its state.

5. In the **`constructor`** function, let's add a new variable to the object that will keep track of the title of the app:

```
this.title = 'React App';
```

6. Now change the **`render()`** function so that the header displays the value we just set instead of the hard-coded My App:

```
<h1>{this.title}</h1>
```

Finally, let's set up some internal state for our object. Do not worry too much about some of the specifics here (such as the **`onClick`** handler), as We will be going into these in much greater detail in later chapters.

We will start off by describing what it is that we want our component to do. In our case, we want to keep track of the number of times a particular button is clicked, and we want to display that count to the user.

> **NOTE**
>
> The button does not exist in the code. The **`clickCounter`** has been hard-coded.

7. Start off by adding an initial state declaration into our **`constructor`** function:

```
this.state = { clickCounter: 0 };
```

8. Next, use a helper function to render the click count display to the user.

   As mentioned previously, React and JSX can handle rendering any function that returns JSX. That means that writing helper methods inside a class is an incredibly simple thing to do.

```
renderClickCount() {
  return <p>I've been clicked {this.state.clickCounter} times!</p>;
}
```

   The even better news is that adding this display to our component is a simple matter of calling the function inside curly braces in our JSX.

9. Add these lines beneath the **`Greeting`** line in our **`render`** function:

```
<br />
{this.renderClickCount()}
```

   If you were following along, the final state of our code for the **`src/App.js`** function should look something like this:

`App.js`

```
3  const Greeting = props => <p>Hello {props.name}!</p>;
4
5  class App extends Component {
6  constructor(props) {
7    super(props);
8    this.title = 'React App';
9    this.state = { clickCounter: 0 };
10  }
```

The complete code can be found here: https://packt.live/2WsVTkA

The final display of the component should look like this:

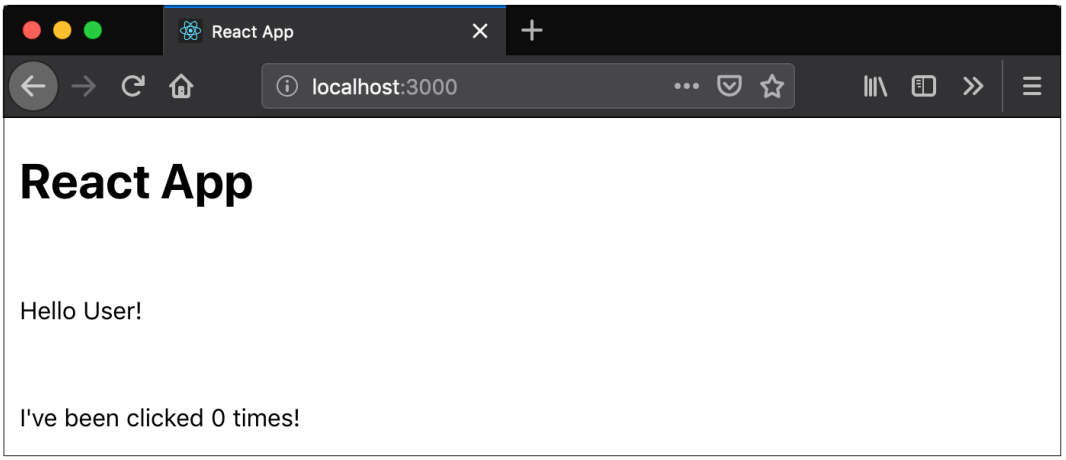

Figure 1.10: Click counter in a React component

With a firm grasp of the fundamentals of bootstrapping your first project in React, let's expand this and build a new project. Your project will allow you to demonstrate the early skills you have learned so far and put them into action while building a practical React example.

ACTIVITY 1.01: DESIGN A REACT COMPONENT

Let's say we want to start building a new e-commerce site, and we need to start off by building a skeleton for the main store page. We will call our project **`buystuff`**. In this project, you'll need to create a new project, create a header React component, create an inventory template React component, and then put everything together in our main React application project.

The following steps will help you to achieve the goal:

1. Verify that Node.js and Create React App are installed on your machine.
2. Create your project, called **`buystuff`**, via the **`Create React App CLI`**.
3. Delete all of the unnecessary files for our project.
4. Build the **`App`** React component as a class component but leave it blank.

5. Build the **`Header`** React component as a functional component. Its only prop should be **`title`**, which contains the store's name.
6. Build the **`InventoryItem`** React component as a functional component. It should contain props that consist of the item name and price.
7. Change the **`App`** component to have a constructor and a starting state with two items in it, and include your **`InventoryItem`** component in the **`render()`** function twice and your **`Header`** component once.

The end result should give us a React application that looks like this:

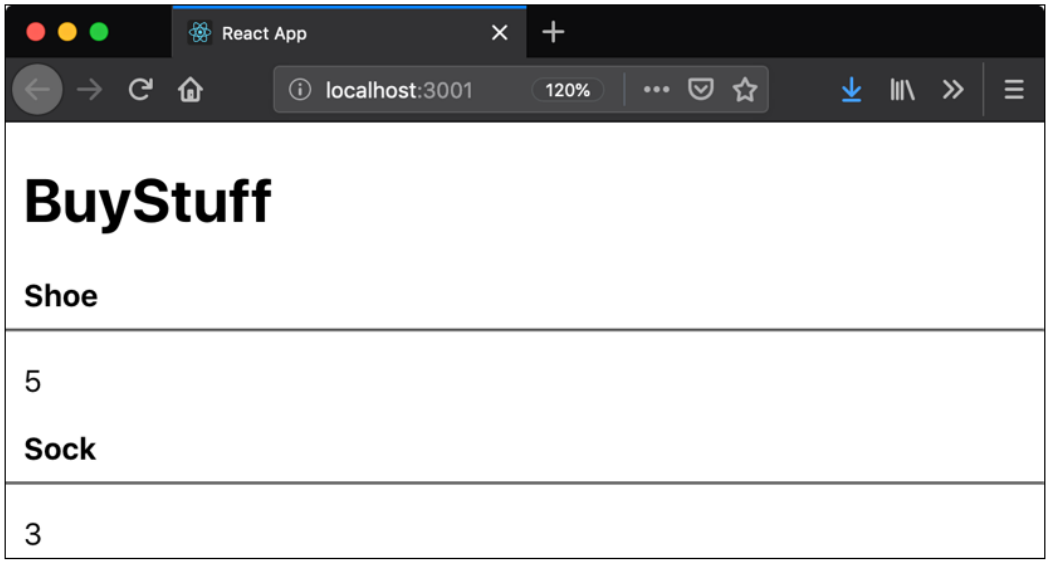

Figure 1.11: BuyStuff React component

> **NOTE**
>
> The solution of this activity can be found on page 606.

SUMMARY

It is important to understand the many moving pieces involved in building React projects, as each one lends itself well to the context and decisions made when building React. Understanding the historical context helps frame the questions around why implementing some things in React is the way it is. In addition, understanding how to use the Create React App framework helps you get moving faster and building new projects with less fuss and effort, making it a critical part of your development workflow.

We also now have a strong understanding of the basic building blocks of any React component, regardless of the level of complexity. We can define small, lightweight components using functional syntax, which is a very common idiom in modern React development. We can also define larger, stateful components using class syntax, which also helps us plan for and architect our more complicated React components in ways that are easy to develop on, maintain, and support over the long term.

Ultimately, the hardest part is just knowing when to use one or the other, and that decision just boils down to if you need helper functions or state. If so, you should probably use a class-based component and not a functional component, otherwise, use the lighter components. In the next chapter, we will build upon this base layer in much greater detail, creating a strong foundation on which we will build some very complex React projects from start to finish on top of the knowledge that you have gained here.

2

DEALING WITH REACT EVENTS

OVERVIEW

This chapter will provide you the knowledge about hooking JavaScript events in React applications. You will be able to identify best practices for wiring JavaScript events to component functions; you will get to practice binding JavaScript events to React framework through multiple hands-on approaches; and use alternate component declaration syntax to minimize bind issues.

INTRODUCTION – TALKING TO JAVASCRIPT WITH REACT

In the previous chapter, we designed our first standalone basic React application. The React framework allows you to create rich interactive web applications where users expect an interface that responds quickly to their actions. Often, when you are designing such interactive web applications, there is proper feedback expected within a fraction of a second for even a simple button click. Visualize this scenario:

you click a submit button after filling out a form on a web application, but nothing visible happens after you click the button.

From there, you might do one of two things, either sit and wait for some sort of feedback or furiously hit the button repeatedly, waiting for some sort of feedback or acknowledgment.

Now, you might get some feedback if the form redirects to a new page, or you might receive an error message with the form being submitted multiple times. In either of those scenarios, the developer who created that application has now created a user experience that feels nearly hostile to the end user, one that robs the user of valuable feedback and a way for them to understand how to properly use the application.

The good news is that these are all easily solvable problems in React. We can build our applications in a way that allows the users to be aware of each interaction they have with the site by providing some sort of feedback to the user so that they stay interested and use your application instead of giving up in frustration. You can start giving the user real-time feedback, gently guiding them along as they fill out the form and click the submit button or when they move the focus or cursor away from a field.

In this chapter, we are going to focus on how to intertwine our React components and JavaScript events in a way that really allows for the rich user experience on the web that most people expect nowadays. We will set up event listeners and handlers in our JSX code, and we will build functions in our components that will allow us to handle those events appropriately and change the state of our components. We will start off by designing how we want our component to function before we move into developing the code for the component.

DESIGNING THE STATE OF OUR APPLICATION

A typical pattern in modern web applications is that when there is a form with multiple input fields, one input field might rely on another input field. For example, a password field may affect a password confirmation field by requiring that the two fields match with each other. While this validation is being performed, there is another validation happening which will prevent the user from clicking the submit button if there are errors. In addition, there will be other fields that will affect each other. For example, a password field may require the password to be different from the username entered. For the purpose of our example, we are going to build a form where someone can sign up for an account. They will need to enter:

- A username
- A password
- A password confirmation
- An email address

In terms of validations, we will want to make sure that:

- The username is filled out.
- The password is filled out.
- The password matches the confirmation.
- The email address is at least in the format of (someusername)@(somedomain.com).
- The submit button is grayed out if there are any errors.

> **NOTE**
>
> We are intentionally using an overly simplistic method of validating email addresses here to avoid complexity.

QUICK UI EXAMPLES

Let's go through some User Interface (UI) examples to better understand the flows and events that will happen here. This is how the form will look when the user opens the page:

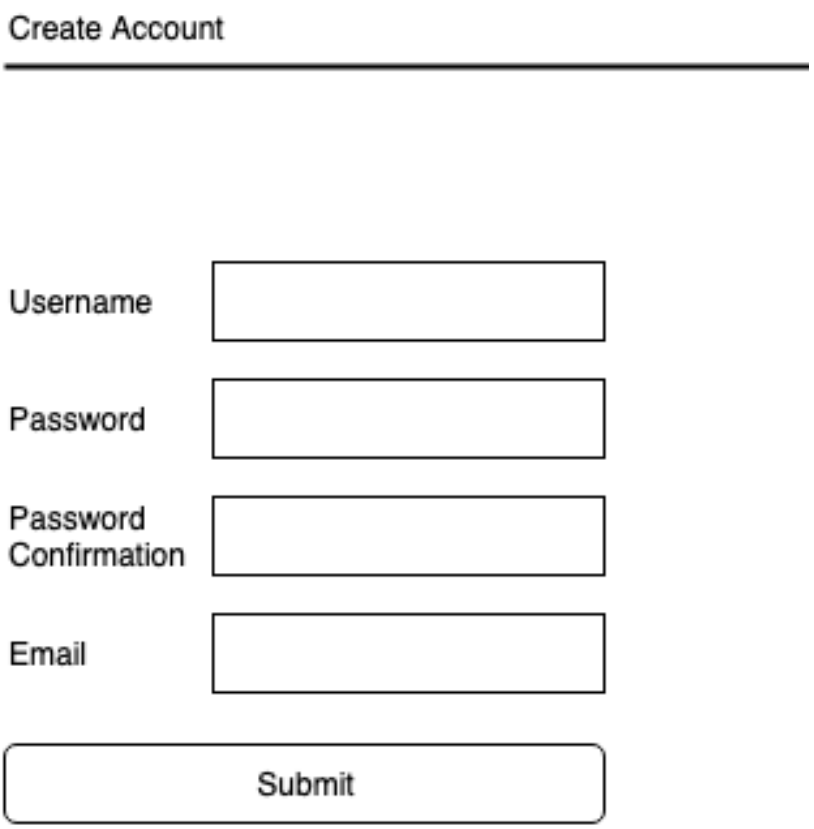

Figure 2.1: UI form example

If the **`username`** is blank, an error message shows up at the top of the page, as shown in the following figure:

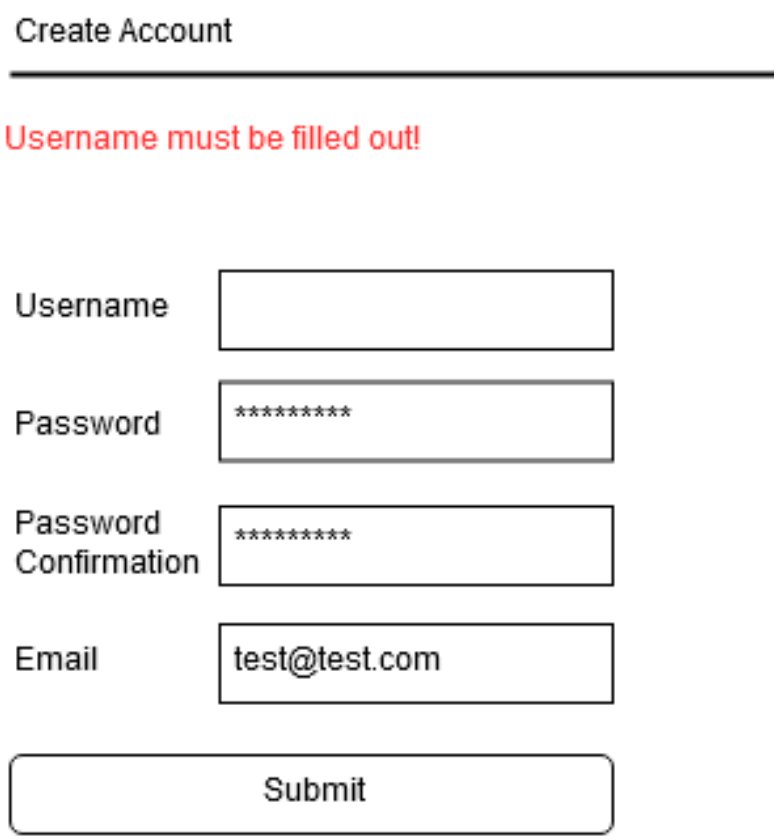

Figure 2.2: Error message while validating the username field

If the passwords don't match, the following error message will pop up:

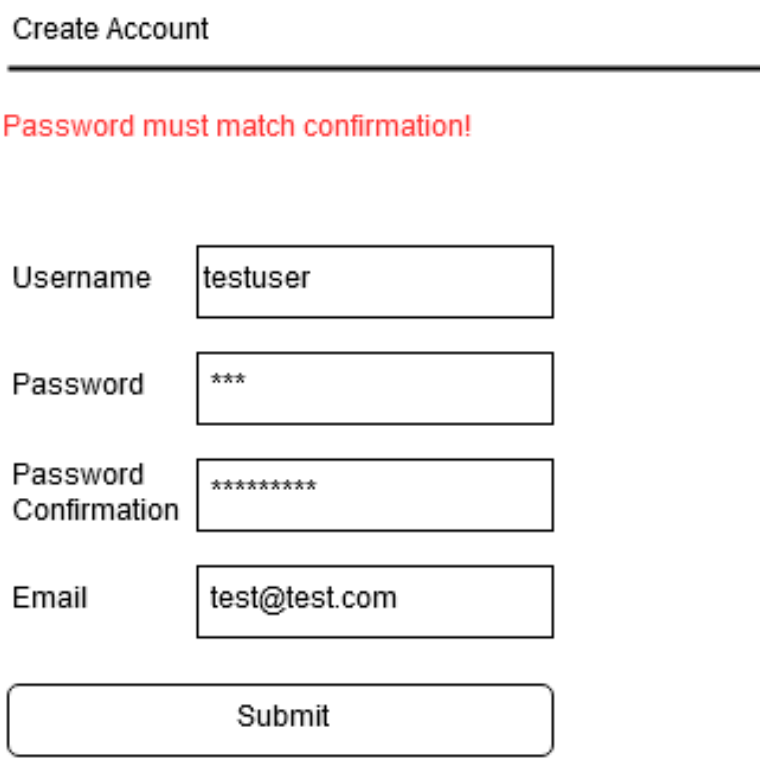

Figure 2.3: Error message while validating the password field

If there are multiple errors in the same form, both the error messages will pop up:

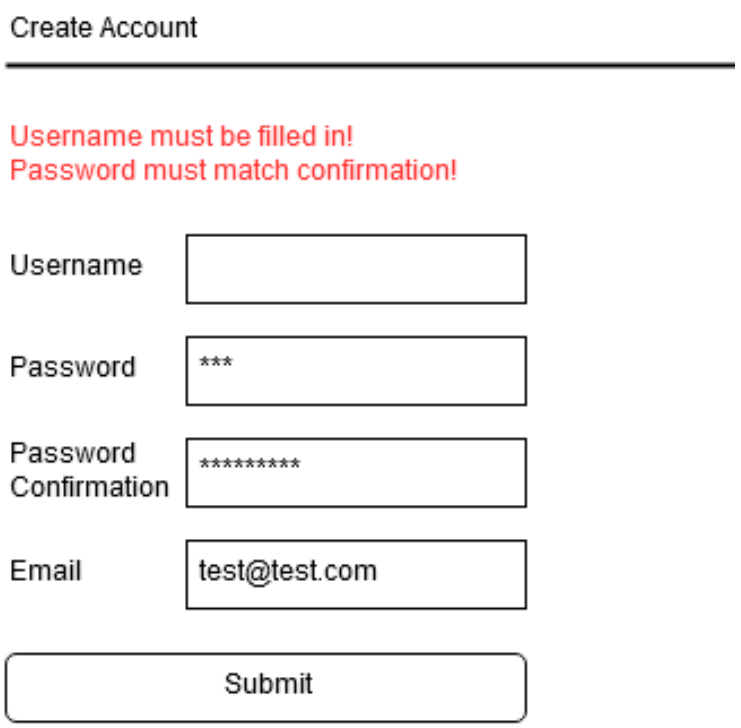

Figure 2.4: Error messages while validating the username and password fields

These are just a few of the examples that we will be dealing with along the way. The purpose of these examples was to illustrate the general flow for the forms that we are going to build in the following section and make it easier for us to visually relate to. Let's start by building the states and validations in the form.

GETTING STARTED – BUILDING OUR BASELINE COMPONENT

To get started with our baseline component, let's think about each of the pieces that we will build and eventually put together to create this form. We are going to build a small component to represent our form and slowly build onto this component to compose it into something more functional.

We will start off with a class-based component since, typically, these interactions will require a certain amount of state manipulation and it's easier if we build the component for that purpose from the start. As discussed in *Chapter 1*, *Getting Started with React*, let's go to the Node.js command prompt and start a new Create React App project for this to keep everything separate and easy to follow. We will start off by changing the contents of App.js to just be a simple boilerplate class component. We do this to make it easier for us to iterate on this project and establish a baseline for our UI:

```
import React, {Component} from 'react';
class App extends Component {
  render() {
    return (
      <div className="App">
        Test App
      </div>
    )
  }
};
export default App;
```

This should just give us our standard shell **App** display:

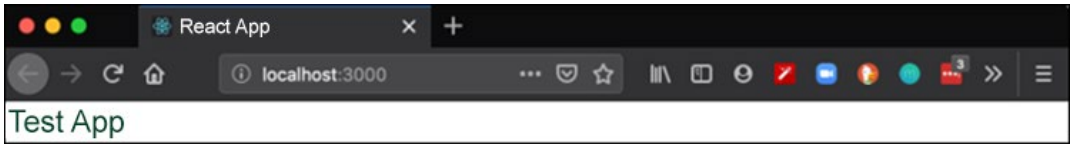

Figure 2.5: Our baseline component

Typically, for components such as these, we also want them to be able to interact with the user in some way, which typically necessitates some form of simple state management. Now, before we can jump into coding the constructor and the basic state representation, we need to first discuss what we are going to build.

To figure out what we will need to include as properties in our state, we will need to take a look at the design and do a little bit of noun extraction to figure out the state of our component.

> **NOTE**
>
> Noun extraction is the process of exploring the mockup, design, or documentation to find the nouns and use them to inform the general properties or attributes that an object has. It can also be used to identify the objects themselves.

In this case, our form has the following nouns attached to it:

- Username
- Password
- Password confirmation
- Email
- A list of errors

We can also make a few assumptions based on the design that will tell us the types for our nouns as well. For example, nearly all of those nouns are strings, with the exception of the list of errors, and even that is just a simple list of strings.

EXERCISE 2.01: WRITING THE STARTER STATE FOR OUR CODE

Now that we have the general flow designed, we can use that to extrapolate out how the state should be initialized and declared in our constructor. We need to build our initial form and create a scaffold before we can start doing anything more complex, so that will be the objective of this exercise:

1. Start off your component in **`src/App.js`** with the boilerplate code discussed in *Chapter 1*, *Getting Started with React* and build a constructor:

```
class App extends Component {
  constructor(props) {
    super(props);
  }
}
```

Build out an initial state for your code referencing the design from earlier. Define the state by setting a state property on the object inside of the constructor, and that state needs to be a JavaScript object with each of the appropriate fields in it:

```
this.state = {
  username: '',
  password: '',
  passwordConfirmation: '',
  email: '',
  errors: []
};
```

We will also write this into its own function to help us keep our render function nice and lean, so let's write our **`displayForm()`** function. It must include the **`username`**, **`password`**, **`passwordConfirmation`**, and **`email`** fields that we created in our initial state.

2. Create a text input for each of those strings and a button to submit the form:

`App.js`

```
<div>
  Username: <input type="text" /><br />
  Password: <input type="text" /><br />
  Password Confirmation: <input type="text" /><br />
  Email: <input type="text" /><br />
  <br />
  <button>Submit</button>
</div>
```

`The complete code for this step is available at:` https://packt.live/35qT2L3

3. Next, modify our **`render`** function to use this new form:

```
render() {
  return (
    <div className="App">
      Create Account
      <hr />
      {this.displayForm()}
    </div>
    )
  }
}
export default App;
```

When we save and reload from here, we should expect to see the form showing up on our page:

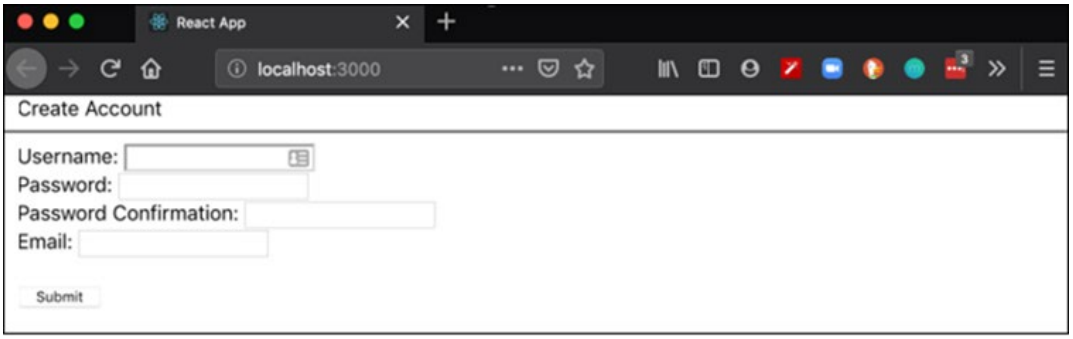

Figure 2.6: Starter Component

This gives us an initial form to work with, built entirely in React and ready to hook into the next functionality we need to start adding: our event handlers.

EVENT HANDLERS IN REACT

We will be adding a few validations that we will be attaching to our baseline component. We do not want these validations to happen after the user clicks **`Submit`** button, because at that point the user has already filled out much of the form and that can make for a bad user experience. It also should not return back to the user with the list of validation issues on form submission. Instead, we are going to have each of these triggers along the way and then we are going to listen to a button click event to simulate submitting the form. We will talk about a few different events, such as **`onClick`** and **`onBlur`**, and get comfortable with interacting with them. This will also give us a good amount of room to experiment and help us become comfortable with properly dealing with JavaScript events in React and some of the gotchas we will run into along the way. **Event handlers** in React manipulate DOM elements similarly to how JavaScript does. However, there are two major differences:

The naming convention for React events is camelCase and not lowercase.

As an event handler, since React uses JSX, you pass a function rather than a string.

For example, take the HTML for an **`onClick`** handler:

```
<button onClick="this.submitForm()">Submit</button>
```

It's slightly different in React with JSX:

```
<button onClick="{this.submitForm}">Submit</button>
```

Let's start off with event handlers in React.

ONCLICK

To begin understanding and working with event handlers in React, we will start off with the simplest and most common: the click event handler. The **`onClick`** handler is used to define what event fires off when a DOM object is clicked, so we are not going to do anything too crazy.

In React, this is the JSX code that sets up the event handler **`onClick`**:

```
<button onClick={this.submitForm}>Submit</button>
```

But how is the event argument being passed? Nothing here appears to be doing that. You must see this through the lens of treating functions like arguments as well and also visualize a little bit of the code magic that is happening behind the scenes. For example, when you set up an event handler in React, you are essentially seeing the end result of that getting translated to an equivalent **`call()`** statement:

```
(target element).addEventListener("click", function(event) {
  (target handler function).call((target handler context), event);
});
```

That means that, at runtime, the call before **`this.submitForm`** is getting roughly translated to this:

```
this.submitForm(event);
```

With this bit of knowledge, let's write our first event handler in React through an exercise.

EXERCISE 2.02: WRITING OUR FIRST EVENT HANDLER

The objective of this exercise is just to get moving with adding a basic event handler to our baseline component. We will use the form created in *Exercise 2.01*, *Writing the Starter State for Our Code* of this chapter.

1. Let's head back to the **`displayForm()`** function, built in the previous exercise, and find the button in the form. Here, add the **`onClick`** handler:

```
    displayForm() {
      return(
        <div>
          Username: <input type="text" /><br />
          Password: <input type="text" /><br />
```

```
          Password Confirmation: <input type="text" /><br />
          Email: <input type="text" /><br />
          <br />
          <button onClick={this.submitForm}>Submit</button>
        </div>

    );
  }
```

We will call this click handler **`submitForm`** and reference it inside our component class since this will be an event handler local to this component.

2. Next, write the **`submitForm()`** function:

```
submitForm(event) {
  console.log("Submitting the form now...");
}
```

Now when the button is clicked, we will get something showing up in our JavaScript logs telling us that the button was properly clicked. If you set everything up correctly, you should see some output in your web console indicating that the button was clicked:

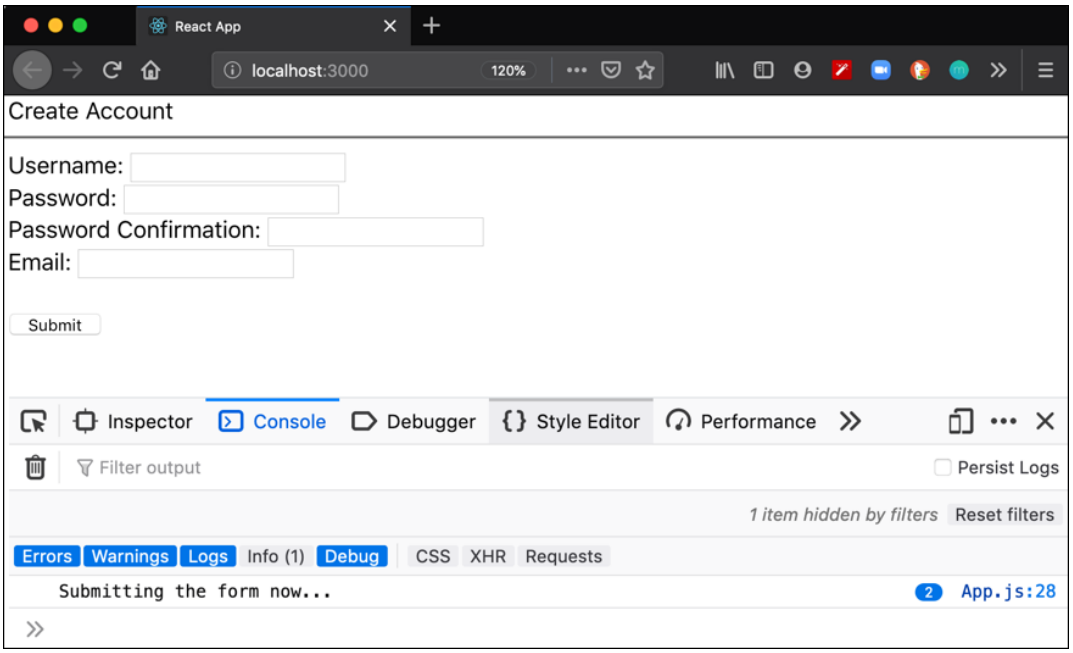

Figure 2.7: JavaScript Logs for the Starter Component

Since we wrapped our code in a form, we want to make sure that the **`Submit`** button doesn't automatically try to submit the form and reload the page, so we put a quick **`event.preventDefault()`** call at the top of the function to prevent that behavior.

3. Modify the **`submitForm`** function to include a **`console.log`** statement that will output the event:

```
submitForm(event) {
  console.log("Submitting the form now...");
  console.log(event);
}
```

When we click the button here, we will see a lot more input into the JavaScript console. Specifically, we should now be seeing details about the JavaScript event that our code is now listening for and reacting to:

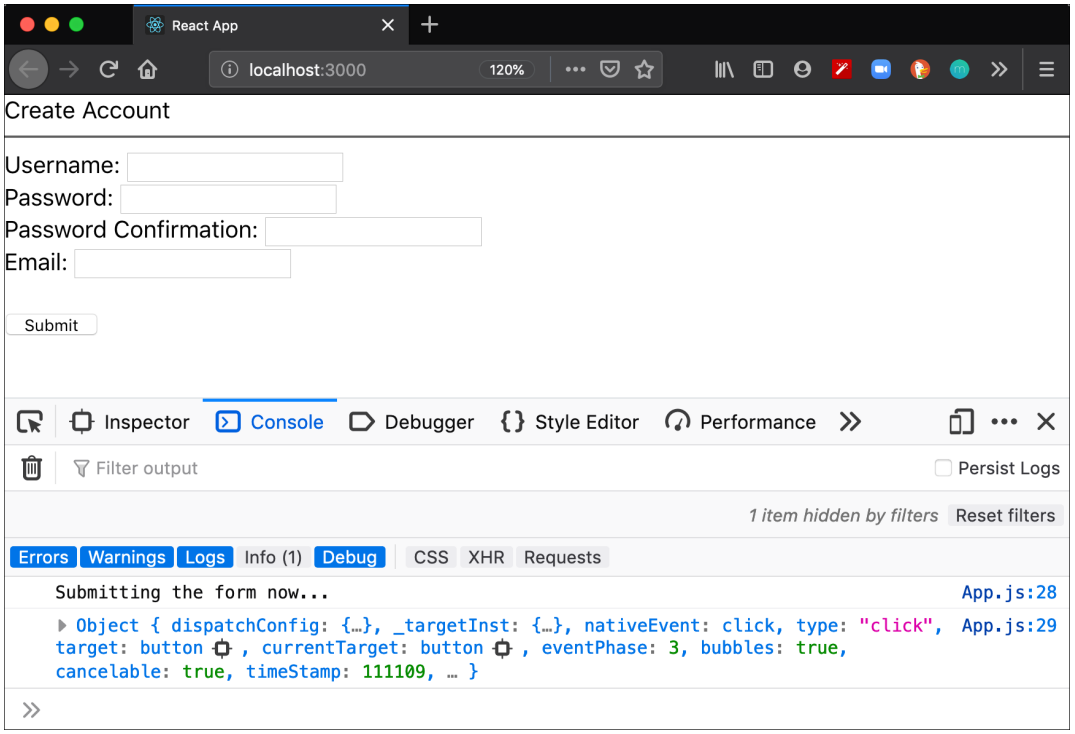

Figure 2.8: JavaScript for events

If you take a look at the output you can see a lot of details about the request and the event, but this, of course, begs the question: what do target and **`currentTarget`** refer to? The target and **`currentTarget`** properties of the event will always refer to the DOM elements, not the React components, so you will need a separate way to identify the element that triggered the event.

ONBLUR

Another event handler that is very frequently used in React is **`onBlur`**. This event performs the validation for each field as that field loses focus. This means when you tab out of that particular field for the form, the validation occurs because we are assuming at that point that the user is done editing that field. We will add the **`onBlur`** event handler to the form that we have built in the previous sections in a bit. But for now, let's look at how the **`onBlur`** event is written in HTML and in React with JSX.

Here it is in HTML:

```
Username: <input type="text" onblur="myFunction()" />
```

It's slightly different in React with JSX:

```
Username: <input type="text" onBlur={this.validateUsernameOnBlur} />
```

The form that we have built so far is broken up into multiple different fields. A lot of these fields will be repeating the previous implementations as we go along, so there will not be a great deal of unique code to write. Let's add the **`onBlur`** event in action through the following example.

We will start with modifying the **`Username`** text field of our form and add an **`onBlur`** event as shown below:

```
displayForm() {
  return (
    <div>
      Username: <input type="text" onBlur={this.validateUsernameOnBlur}
/><br />
      Password: <input type="text" /><br />
      Password Confirmation: <input type="text" /><br />
      Email: <input type="text" /><br />
      <br />
```

```
      <button onClick={this.submitForm}>Submit</button>
    </div>
  );
}
```

We will write our **`validateUsernameOnBlur`** function in our component by adding a quick validation on the input field's value, which you will be able to get through a property on the event's target object:

```
validateUsernameOnBlur(event) {
  console.log("I should validate whatever is in ", event.target.value);
}
```

It is always preferable to start off the event handlers with a quick **`console.log`** statement, as it helps in validating the events that are being fired off without having to trace through more complicated code later.

In this function, you can see that we are explicitly checking for the event argument, and then in our **`console.log`** statement, we are trying to get the value of the text input, which we can fetch by referencing the event's target object and pulling the value out of it:

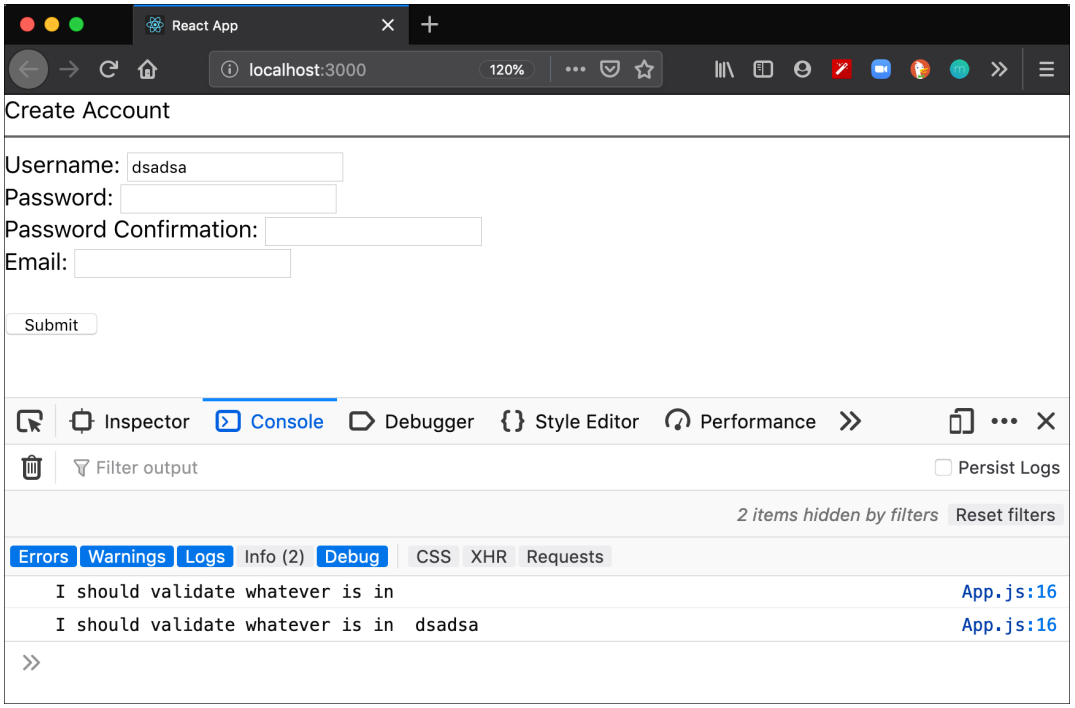

Figure 2.9: Console statement

As we can see in the preceding screenshot, the **`console.log`** statement gives us the form value we entered into the text box at the time of tabbing away from that field or clicking on another field, which is exactly what we want. We will see the implementation in much more detail in a bit but for now, let's go a step ahead and understand the context of these event handlers.

CONTEXT OF EVENT HANDLERS

While building a form, you will be interacting very heavily with event handlers, and understanding the context of **`this`** property will prevent you from scratching your head down the line when you get error messages such as **`this is undefined`**. Especially in the case of forms, you will be dealing very heavily with event handlers and event handlers will change the context of **`this`** property. Now, if we want to set the new state in our baseline component of the form we created earlier, we will have to call out **`this.setState`** in our **`validateUsernameOnBlur`** function. However, if you try to do that, you are going to hit a **`this is undefined`** error message. For example, take changing our **`validateUsernameOnBlur`** function to the following:

```
validateUsernameOnBlur(event) {
  console.log("I should validate whatever is in ", event.target.value);
  this.setState();
}
```

The preceding code results in the following error:

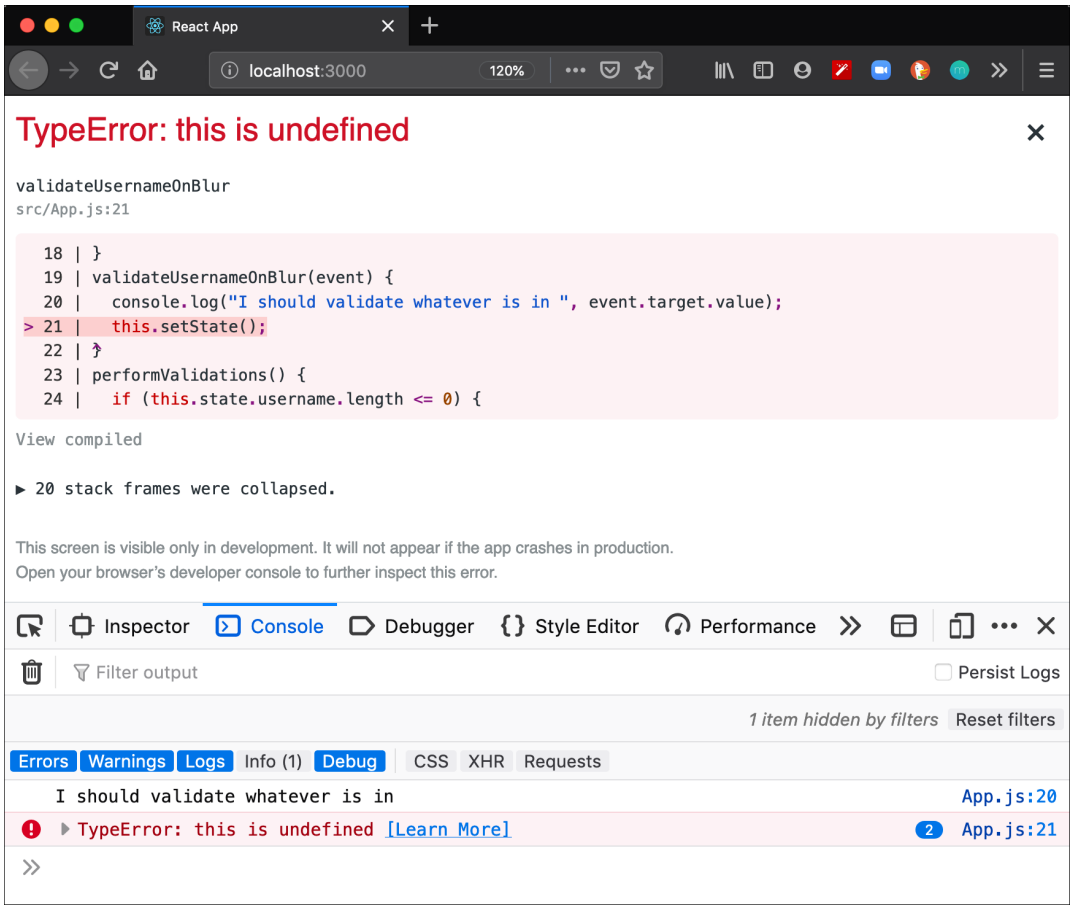

Figure 2.10: TypeError in the console statement

The reason is that because the event handler code is essentially wrapping up the call and calling our event handler function for us, it's losing the context of the function, which should essentially be the component. Instead, the context becomes undefined. The **`this is undefined`** error message can be hard to track down if you don't know what you are looking for. The error message is ambiguous as **`this`** keyword is not explained.

The good news is that this is an incredibly simple thing to solve. We can explicitly tell JavaScript to bind the context of **`this`** keyword to the component itself instead of allowing the event handler to show us the context. There are two common ways to address this:

- In-line bind statements
- Constructor bind statements

IN-LINE BIND STATEMENTS

The simplest method to add a context to our baseline component is to add **`bind(this)`** call to the end of our event handler declaration in the input field like so:

```
displayForm() {
  return (
    <div>
      Username: <input
      type="text" onBlur={this.validateUsernameOnBlur.bind(this)} /><br
/>
      Password: <input type="text" /><br />
      Password Confirmation: <input type="text" /><br />
      Email: <input type="text" /><br />
      <br />
      <button onClick={this.submitForm}>Submit</button>
    </div>
  );
}
```

Make that change and the code will start working again when you select a field other than the username field. This is a shortcut to solve this problem if you only need the bind in a single place, but is not a great strategy if we have to write in-line bind statements multiple times in the code, especially for repeat calls to the same functions. We will see another way to bind **`this`** concept to the component next.

CONSTRUCTOR BIND STATEMENTS

We use constructor bind statements to tell JavaScript explicitly that when we reference the **`this.validateUsernameOnBlur`** function in our component, it should always have the context of the component bound to it when this is referenced.

Since the **`constructor`** in the class-based components is used to declare the state of a component, when we are calling **`this.state()`** in the constructor, we should bind our event handlers explicitly inside of our constructor to avoid doing repetitive tasks, especially for the same functions, to save ourselves a little bit of extra time and effort. This requires us to add the following line to the **`constructor`**:

```
this.validateUsernameOnBlur = this.validateUsernameOnBlur.bind(this);
```

> **NOTE**
>
> Constructors have been discussed in detail in *Chapter 4*, *React Lifecycle Methods*.

Now we can go back to our **`displayForm()`** function and remove the in-line **`bind`** statement instead:

```
displayForm() {
  return (
    <div>
      Username: <input type="text" onBlur={this.validateUsernameOnBlur}
/><br />
      Password: <input type="text" /><br />
      Password Confirmation: <input type="text" /><br />
      Email: <input type="text" /><br />
      <br />
      <button onClick={this.submitForm}>Submit</button>
    </div>
  );
}
```

Our code will otherwise remain identical. If you try this again, you should again see the focus change work and not result in any additional errors. Let's practice this in the following exercise.

EXERCISE 2.03: BUILDING OUR USERNAME VALIDATOR

In this exercise, we will put the code we just talked about into our component that we created previously. We will add the **`bind`** statement to the constructor and call our **`validateUsernameOnBlur`** function from our **`displayForm`** function when the form input hits the **`onBlur`** event. To do so, let's go through the following steps:

1. Drop our constructor since we can specify the initial state in a different way. Instead, we will use JavaScript fields to define the state by setting a state field on the class:

```
class App extends Component {
  state = {
    username: '',
    password: '',
    passwordConfirmation: '',
    email: '',
    errors: []
  };
  validateUsernameOnBlur = this.validateUsernameOnBlur.bind(this);
```

2. Write the **`validateUsernameOnBlur()`** function. Nothing here should be new:

```
validateUsernameOnBlur(event) {
  console.log("I should validate whatever is in ", event.target.
value);
  this.setState();
}
```

3. Call the **`onBlur`** event handler inside the **`displayForm()`** function, without needing an in-line bind statement:

```
displayForm() {
  return (
    <div>
      Username: <input type="text" onBlur={this.
validateUsernameOnBlur} /><br />
      Password: <input type="text" /><br />
      Password Confirmation: <input type="text" /><br />
      Email: <input type="text" /><br />
      <br />
```

```
        <button onClick={this.submitForm}>Submit</button>
      </div>
    );
  }
```

4. Call the **`submitForm`** function, which is active while the **`Submit`** button is pressed:

```
  submitForm(event) {
        console.log("Submitting the form now...");
        console.log(event);
      }
```

5. Call **`displayForm`** from the **`render`** method:

```
  render() {
        return (
          <div className="App">
            Create Account
            <hr />
            {this.displayForm()}
          </div>
        )
      }
```

The resulting class structure looks like the following:

`App.js`

```
3  class App extends Component {
4    state = {
5      username: '',
6      password: '',
7      passwordConfirmation: '',
8      email: '',
9      errors: []
```

The complete code can be found here: https://packt.live/2PsyyMu

The output is as follows:

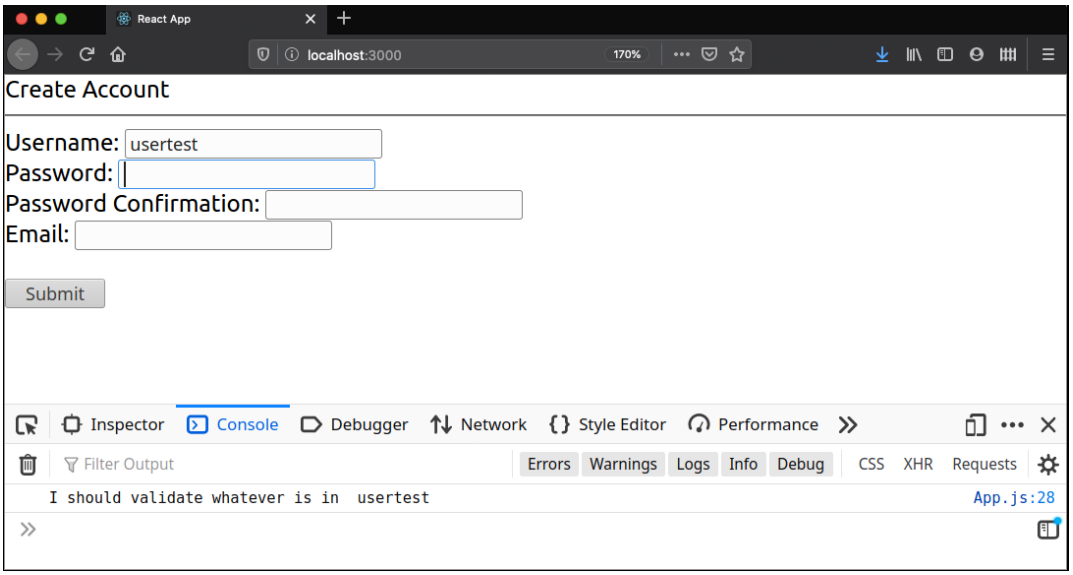

Figure 2.11: Form component

Another way to define our components is to use some relatively newer syntax (the public field syntax) to define our class component, the properties in the component, and the functions in the component. This allows us to define our functions in such a way that they remember the binding of **`this`** keyword regardless of how they are passed or called via event handlers.

EXERCISE 2.04: USING ALTERNATIVE CLASS DECLARATIONS TO AVOID BINDS

In this exercise, we will use an alternative class declaration so that we can avoid the bind statements altogether. We will use the **`displayForm`** component that we created previously. We will drop the **`constructor`** and we will see how to specify the initial state in a different way using arrow functions and declaring the fields inside it. To do so, let's go through the following steps:

1. Drop our **`constructor`** since we can specify the initial state in a different way. Instead, we will use JavaScript fields to define the state by setting a state field on the class:

```
class App extends Component {
  state = {
    username: '',
    password: '',
```

```
      passwordConfirmation: '',
      email: '',
      errors: []
    };
  }
```

2. Redefine the **`validateUsernameOnBlur()`** function by changing the function to instead be a field on the class as well. You will need to use the arrow function syntax, mentioned in *Chapter 1*, *Getting Started with React*, to make this work:

```
validateUsernameOnBlur = (event) => {
  console.log("I should validate whatever is in ", event.target.
value);
  this.setState();
}
```

   The only major difference here is that we are defining the function in a similar way to how we define other arrow functions. The advantage here is that this function now has this bound appropriately, so we don't need to worry about explicitly binding.

3. Call the **`onBlur`** event handler inside the **`displayForm()`** function:

```
displayForm() {
      return (
        <div>
          Username: <input type="text" onBlur={this.
validateUsernameOnBlur}
         /><br />
          Password: <input type="text" /><br />
          Password Confirmation: <input type="text" /><br />
          Email: <input type="text" /><br />
          <br />
          <button onClick={this.submitForm}>Submit</button>
        </div>
      );
    }
```

4. Call the **`submitForm`** function, which is active while the **`Submit`** button is pressed:

```
submitForm(event) {
      console.log("Submitting the form now...");
      console.log(event);
    }
```

5. Call **`displayForm`** from the **`render`** method:

```
render() {
      return (
        <div className="App">
          Create Account
          <hr />
          {this.displayForm()}
        </div>
      )
    }
```

The resulting class structure looks like the following:

`App.js`

```
1  class App extends Component {
2    this.state = {
3      username: '',
4      password: '',
5      passwordConfirmation: '',
6      email: '',
7      errors: []
8    };
```

The complete code can be found here: https://packt.live/2UXLBrr

The output is as follows:

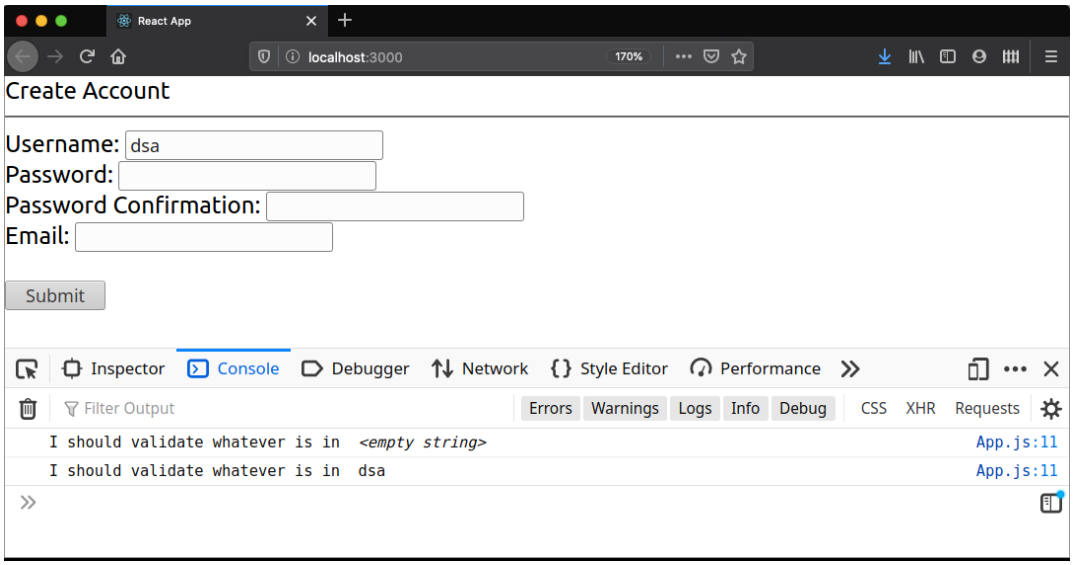

Figure 2.12: Validation in the form component

Unfortunately, this is still a syntax proposal and is not guaranteed to be supported in every environment that you may be working in, so for now, we will stick with the more widely available syntax. If you are working in a Create React App project, though, and feel more comfortable using the proposed fields syntax instead, that remains an option. Create React App will create a project with the appropriate **Babel** config to use the public class fields syntax by default.

Now let's look at the ways to handle our validation, but we are going to go with an approach that is more aligned with common React best practices.

FORM VALIDATION IN REACT

We need to finish creating the validation check to make sure the username is not empty when the mouse cursor moves away from the field. We will need to do a few things to achieve this functionality:

We will need to build a rendering function that displays the list of form errors. We will need to hook the **`validateUsernameOnBlur()`** function to validate the username and update the state where appropriate.

The first task is simple enough that we should be able to build it without really needing to update much else, so we will start there.

EXERCISE 2.05: CREATING A VALIDATION FOR INPUT FIELDS

We are still working with the same form that we have been working with this entire chapter. We will continue to iterate on this form now by adding a little more validation for the username field. The list of errors that we have is a relatively simple thing to display; all we need to do is iterate over the list of errors, and then for each of those, we need to just display a simple quick string. React has a little gotcha that we will run into when building out lists of elements dynamically: each individual item requires a separate entry for the key of the item. This allows React to quickly identify each item uniquely and update the DOM representing that item when it changes:

1. Write a **`displayErrors()`** function that, given a list of errors from our state, maps over each. You will need to set the key property and determine some way to uniquely identify each element as a value for the key property:

```
displayErrors() {
   return (
     <div className="errors">
       {this.state.errors.map((err, i) => <p key={`err-${i}`}>{err}</
p>)}
     </div>
   );
 }
```

   We start off with a simple-enough function signature; there are no arguments that we are passing along. Instead, we just have a simple map call where we are passing each error and the index of the iteration. This allows us to set a unique key for each item, which we then render with a simple **`<p>`** tag.

2. Add the **`displayErrors()`** call to our main **`render`** function:

```
render() {
  return (
    <div className="App">
      Create Account
      {this.displayErrors()}
      <hr />
      {this.displayForm()}
    </div>
  )
 }
```

3. Recreate **`src/App.css`** and add a single bit of CSS to it:

```
.errors {
  color: red;
}
```

4. Change our **`src/App.js`** file to import this CSS file near the top:

```
import "./App.css";
```

Since we don't want to write multiple validations every single time for each field that needs to be validated to ensure that it's not blank, we will start off by refactoring our code to move the not-empty check into a new function.

5. Create a new function to validate whether the field is blank or not:

```
validateNotEmpty(fieldName, value) {
  if (value.length <= 0) {
    return `${fieldName} must be filled out.`;
  }
 }
```

We will want to include not just the value that we need to validate, but also the field name to be able to generate the appropriate error message. We check the value supplied to make sure it's not blank, and then if it is, we will return a string back with the appropriate error message.

6. Modify our **`validateUsernameOnBlur()`** function call from *Exercise 2.04*, *Using Alternative Class Declarations to Avoid Binds* to a new helper function to perform the validation:

```
validateUsernameOnBlur(event) {
  const username = event.target.value;
  const errors = this.state.errors;
  errors.push(this.validateNotEmpty("Username", username));
  this.setState({ username, errors });
}
```

The bulk of the function stays the same, but now writing a **`validatePasswordOnBlur`** function becomes significantly easier for us.

7. Copy our **`validateUsernameOnBlur`** from the previous step and change username where appropriate to password:

```
validatePasswordOnBlur(event) {
  const password = event.target.value;
```

```
    const errors = this.state.errors;
    errors.push(this.validateNotEmpty("Password", password));
    this.setState({ password, errors });
  }
```

8. We will add the constructor again in this step. Use the **`bind`** statement in the constructor:

```
  constructor(props) {
    super(props);
    this.state = {
      username: '',
      password: '',
      passwordConfirmation: '',
      email: '',
      errors: []
    };
    this.validateUsernameOnBlur = this.validateUsernameOnBlur.
  bind(this);
    this.validatePasswordOnBlur = this.validatePasswordOnBlur.
  bind(this);
  }
```

9. Change our **`render`** function as well to modify the password field to use this new validation function:

```
  displayForm() {
    return (
      <div>
        Username: <input type="text" onBlur={this.
  validateUsernameOnBlur}
      /><br />
        Password: <input type="text" onBlur={this.
  validatePasswordOnBlur}
        /><br />
        Password Confirmation: <input type="text" /><br />
        Email: <input type="text" /><br />
        <br />
        <button onClick={this.submitForm}>Submit</button>
      </div>
    );
  }
```

10. Write our **`validateEmailOnBlur()`** function. The code in **`validateEmailOnBlur()`** is simple enough and follows the same format we have been using:

```
validateEmailOnBlur(event) {
  const email = event.target.value;
  const errors = this.state.errors;
  errors.push(this.validateEmailFormat("Email", email));
  this.setState({ email, errors });
}
```

11. Split the field's value on a `@` character and verify that both sides have at least one character in them:

```
validateEmailFormat(fieldName, value) {
  let [lhs, rhs] = value.split('@');
  lhs = lhs || '';
  rhs = rhs || '';
  if (lhs.length <= 0 || rhs.length <= 0) {
    return `${fieldName} must be in a standard email format.`;
  }
}
```

12. Modify the email text field in the **`displayForm()`** function:

```
Email: <input type="text" onBlur={this.validateEmailOnBlur} /><br />
```

13. Add validations for our password confirmation to ensure it matches the password by writing a **`validatePasswordConfirmationOnBlur()`** function, adding it to **`render`**, and adding the binds for the last two validation functions we wrote. First, let's write the **`validatePasswordConfirmationOnBlur()`** function:

```
validatePasswordConfirmationOnBlur(event) {
  const passwordConfirmation = event.target.value;
  const errors = this.state.errors;
  if (passwordConfirmation !== this.state.password) {
    errors.push("Password must match password confirmation.");
  }
  this.setState({ passwordConfirmation, errors });
}
```

14. Add the call to our new function for **`displayForm()`** to our password confirmation field:

```
Password Confirmation: <input type="text" onBlur={this.
validatePasswordConfirmationOnBlur} /><br />
```

15. Finally, make sure all these functions are appropriately bound in our constructor (we have added two of these already, but we will include all four for the sake of completeness):

```
this.validateUsernameOnBlur = this.validateUsernameOnBlur.bind(this);
this.validatePasswordOnBlur = this.validatePasswordOnBlur.bind(this);
this.validatePasswordConfirmationOnBlur =
     this.validatePasswordConfirmationOnBlur.bind(this);
this.validateEmailOnBlur = this.validateEmailOnBlur.bind(this);
```

Now, when you run through the form and break all the rules we have established here, you should see all of the error messages show up at the top of the form:

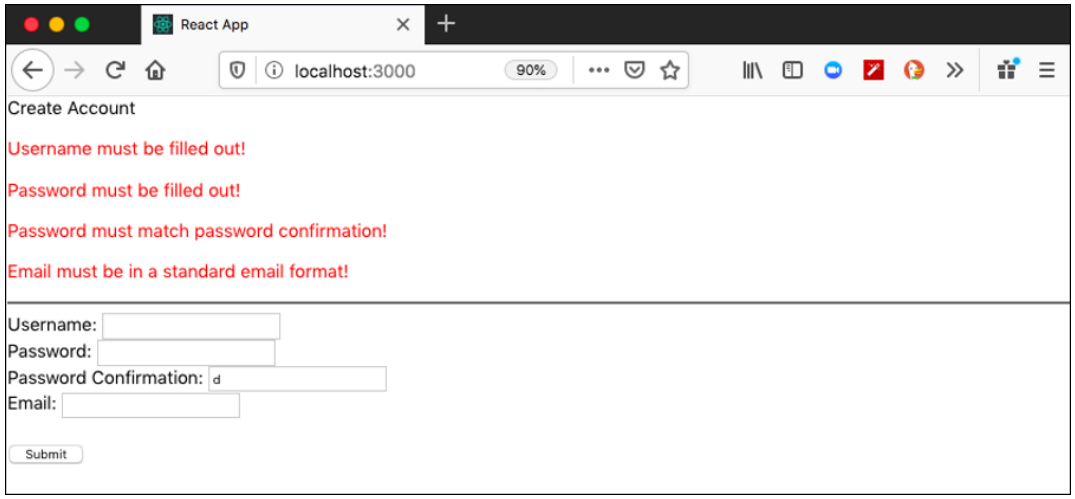

Figure 2.13: Form component with the error messages

The great news is that any other event handlers you may want to write along the way are going to hinge on the same rules that you have just learned about here. It is important to become comfortable with writing event listeners and with knowing how and when to bind functions to avoid weird issues as you write more complicated code.

Now that we have explored how to build React event handlers and to dynamically modify what we are rendering; we need to put it into practice the concepts we have learned so far.

ACTIVITY 2.01: CREATE A BLOG POST USING REACT EVENT HANDLERS

In this activity, we are going to build out an application that keeps track of the number of characters entered into a text area. This will require us to hook into a new event that we have not used yet and change the text and element rendered to the page to display the total length of the field. Specifically, we are going to make a text area that, unless the user has entered at least **`100`** characters, will not allow you to submit the form and post the text.

Here, we will build a new React application with Create React App and then build a text area, adding the length of the field next to it:

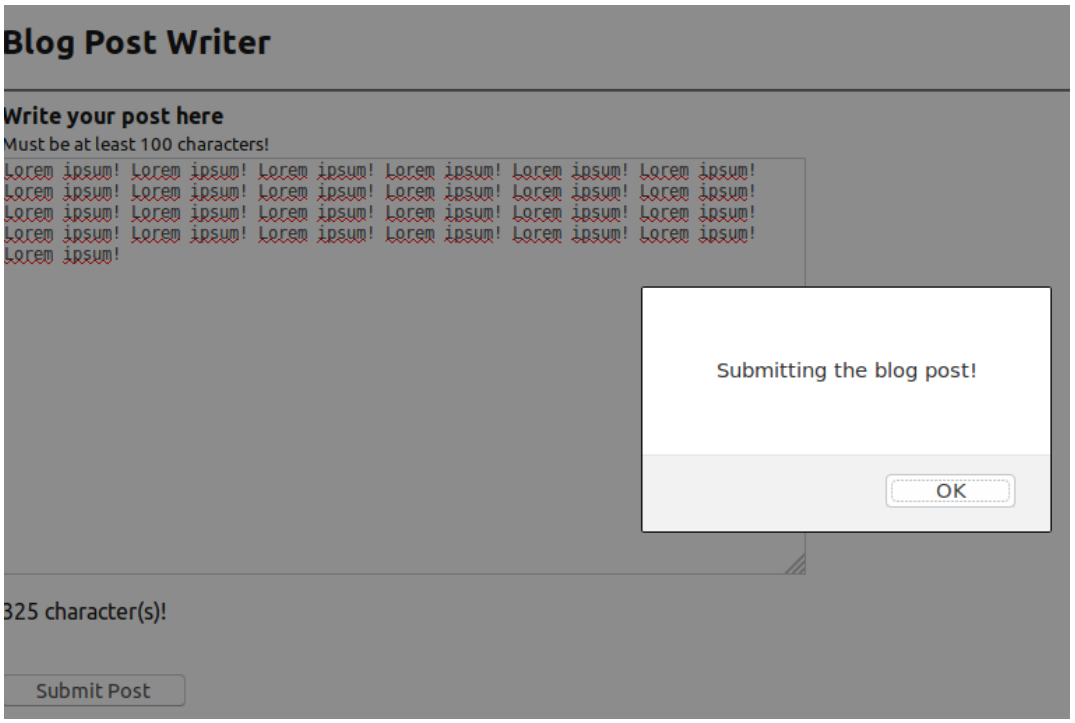

Figure 2.14: Blog Post

The following steps will help to complete the activity:

1. Create your project, called **`fieldlength`**, via the Create React App CLI.
2. Delete all the unnecessary files for our project.
3. Build the **`App`** React component as a class component but leave it blank.
4. Give the component an initial state with a single element in the state; this will store the input from the **`textarea`**.

5. Add a **`textarea`** to the component.
6. Add a function that will act as the event handler. This function will need to accept an event as an argument and should update the state of the component by setting the input from the **`textarea`**. Add this to the **`textarea`**.
7. The event handler you need to use here is up to you; if you need a hint, look for one that can react to changes to your form or input to the form.
8. Add a function that will return **`N`** characters, wrapped inside JSX, where **`N`** is the length of the input in the **`textarea`**.
9. Add the function above to the display of your component.
10. Add a **`submit`** button to make the app look like a blog post editor.
11. Include some instructions for the user that the post must be **`100`** characters.
12. Write a validation function to monitor the length of the field.

    Hint: Try to use a callback with **`setState`** for the best results.
13. Create an alert box to visually notify the user before submitting a blog post.
14. Verify that as you type text into the **`textarea`**; the display is now updated.

> **NOTE**
>
> The solution for this activity can be found on page 609.

SUMMARY

Over the course of this chapter, we have covered everything you need to be able to confidently use JavaScript events to their fullest in your React applications. While we only covered two event handlers, the reality is that all the event handlers you write will function by the same rules and be subject to the same general usage guidelines. Using this information, you will be able to write code that is efficient, dynamic, and provides the kind of rich user experience that anyone using your web application will truly appreciate. In the next chapter, we will learn how to achieve conditional rendering in React.

3

CONDITIONAL RENDERING AND FOR LOOPS

OVERVIEW

In this chapter, we will implement conditional rendering and various looping techniques in React. You will be able to use conditional rendering in your React application via inline ternaries in JSX and guard clauses. Furthermore, you will be able to programmatically design new components using for loops to create lists of components efficiently. Eventually, you will develop an application to handle complex states in React.

INTRODUCTION

In the previous chapter, we have built applications with relatively complex forms and all of that featured information that always got displayed; as in, there was no hidden data involved. In many applications, however, there tends to be portions of the app where the data is hidden and isn't loaded until you perform some action. You can do this in a few ways using other web development languages that are not typically the best choices. For example, you could hide elements on a web page with CSS using **`display:none;`** but then those sections would still show up if people viewed the source or overrode those CSS rules in your browser. Therefore, this would not be a great way to tackle the situation. Instead, we can rely on writing good JavaScript and React code to only display certain elements or components when it is appropriate, using conditional rendering.

In addition, you will frequently find yourself in a position where you need to display the same elements multiple times on a web page. Sure, you could copy and paste a bunch of times and hope you have gotten the number of elements correct, but it is far cleaner and easier to use loops in React to display multiple repetitive elements.

There are, of course, a few things you will need to be mindful of as you implement these features, but ultimately, they are pretty simple little things you need to remember, and we will cover them throughout the course of this chapter.

CONDITIONAL RENDERING

Conditional rendering is rendering elements that may or may not be displayed on a web page depending on certain conditions or events. These conditionals are decision paths in your code, something like an **`if`** statement. We use conditional rendering to render elements when a particular decision path is hit. A good example of conditional rendering is a credit card information form that only appears when you choose credit card as your payment option. Conditional rendering allows us to use standard JavaScript conditional statements (such as **`if`** and **`else`**) to dynamically choose which components to display in our application and which ones to hide.

For example, let's look at this example:

```
import React, { Component } from 'react';
import './App.css';
class App extends Component {
render() {
 return (<div className="App">
    <button>Click me to show the rest!</button>
    <div>
```

```
      I am the content that should be hidden by default!
    </div>
  </div>
 );
}
}
 export default App;
```

This should just give us a button on a page and a completely visible **`div`** element below that we want to be hidden by default when the user visits this app, as shown in the screenshot that follows:

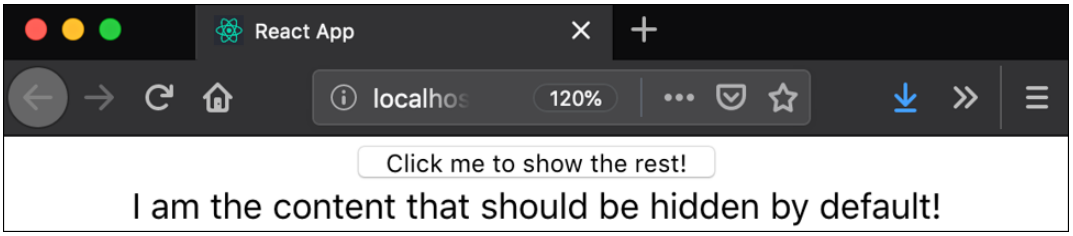

Figure 3.1: Basic app

In React, there are two primary ways to tackle conditional renders:

- Inline in JSX
- Via a dedicated **`subrender`** function.

With *Inline in JSX* option, you might see something like this:

```
import React, { Component } from 'react';
import './App.css';
class App extends Component {
render() {
  const showMessage = true;
  return (
    <div className="App">
      { showMessage && <p>Hello World</p>}
    </div>
  );
}
}
export default App;
```

Here, you can see that we are just adding a conditional statement and only displaying the **`Hello World`** message when **`showMessage`** is **`true`**, literally in the same line as our JSX.

An example of a **`subrender`** function, that is, a dedicated function that checks a conditional and either returns the JSX code or **`null`**, would be this instead as shown below:

```
import React, { Component } from 'react';
import './App.css';
class App extends Component {
renderMessage(show) {
if (show) {
   return <p>Hello World</p>;
 } else {
   return null;
 }
 }
 render() {
 const showMessage = true;
 return (
  <div className="App">
    { this.renderMessage(showMessage) }
  </div>
 );
 }
}
export default App;
```

In the following exercise, we will use these concepts to flesh out a more complex application using conditional rendering.

EXERCISE 3.01: BUILDING OUR FIRST APP USING CONDITIONAL RENDERING

In this exercise, we are going to build a React application that has a button that shows and hides the content on the page on click, very similar to the preceding example. We will create a page with a button and some content in a **`div`**, but we will use conditional rendering to show and hide the content. This will also hinge on your learnings from the last chapter, as we will want to use state to appropriately show or hide the content.

1. Start off by creating a new React application via the same **`npx create-react-app`** command we've run a few times now on the command line:

```
$ npx create-react-app exercise1
$ cd ./exercise1
$ yarn start
```

2. Delete the contents of **`src/App.css`** but keep the file. In **`src/App.css`**, let's add some CSS to make the button a little larger and easier to click:

```
button {
  margin: 0.25em;
  height: 10ex;
  font-weight: bold;
}
```

3. Delete **`src/logo.svg`** since we will not be using that in this exercise.

4. In **`src/App.js`**, start off by changing the React **`import`** statement at the top to also include **`{Component }`** as we will use a class component and will have an internal state to keep track of. Remove the reference to **`src/logo.svg`** since we will not need that logo either. Your **`import`** statements at the top now should just be:

```
import React, { Component } from 'react';
import "./App.css";
```

5. Delete the existing **`App`** component and, instead, we will just want a single **`div`** and a button that just says **`Click to show the secret message!`**. Our new **`App`** component is as follows:

```
class App extends Component {
  render() {
  return (
    <div className="App">
      <button>Click to show the secret message!</button>
    </div>
    );
  }
}
```

This should leave us with an application that looks like this:

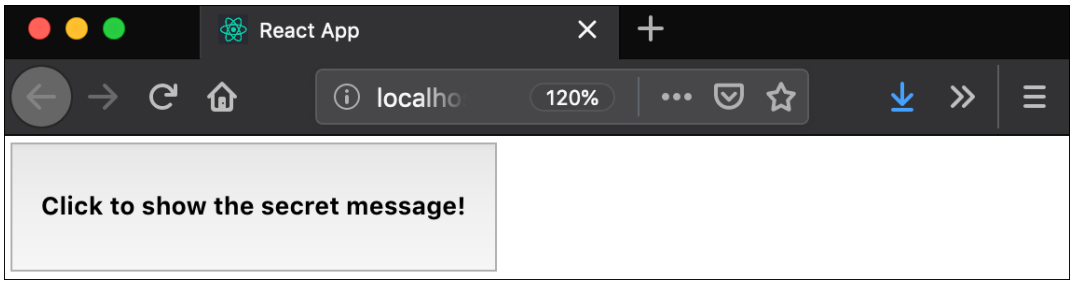

Figure 3.2: App showing the button

6. Build up the state for our application to keep track of whether we're showing or hiding the secret message.

   We'll build out a constructor, as shown in *Chapter 1*, *Getting Started with React*, taking in the **`props`** argument and passing it in a call to the parent class's constructor, and then we'll build up a state that has an initial property in it called **`showSecret`**, which will default to **`false`**:

```
constructor(props) {
   super(props);
   this.state = {
    showSecret: false
};
  }
```

7. Let's write up a conditional renderer now. You can choose to be as creative as you'd like with it:

```
secretMessage() {
 return (
  <div className="secret-message">
   I am the secret message!
  </div>
 )
}
```

8. Add the call to **`secretMessage()`** to your **`render()`** function's JSX so that the text will actually show up somewhere on the app:

```
render() {
  return (
   <div className="App">
```

```
        <button>Click to show the secret message!</button>
        { this.secretMessage() }
      </div>
    );
  }
```

This should give us an app that resembles the following screenshot (depending on what you chose for your secret message, of course):

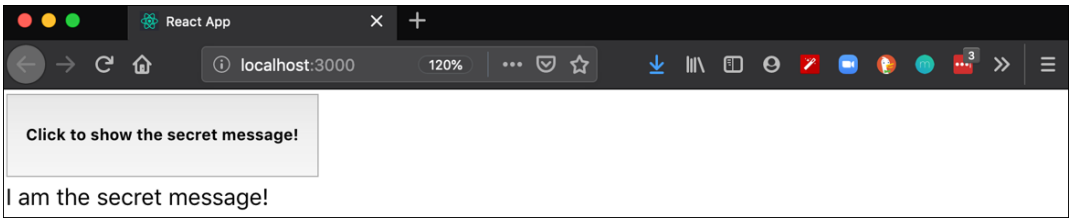

Figure 3.3: App showing the secret message

9. Let's add a new function called **`toggleSecretMessage()`** to start modifying the state.

   This function will be responsible for toggling the state of the secret message display, so this should be relatively simple to write:

```
  toggleSecretMessage() {
    this.setState({
      showSecret: !this.state.showSecret
    });
  }
```

   We will want to call this function from our button since that will act as the trigger to show and hide the secret message. Since we'll be using an event handler in a button to call a function that relies on setting the state of the component, we'll also need to make sure we're appropriately binding this to the context of the component and not the event, so we'll need to include a bind statement.

10. Let's do this by calling **`bind`** on the **`toggleSecretMessage`** function. Therefore, **`this.toggleSecretMessage`** (the function itself) becomes

    `this.toggleSecretMessage.bind(this)`

    We will throw **`bind`** into the **`onClick`** handler, as an inline **`bind`** statement is fine for any function where the **`bind`** only needs to happen once.

11. Let's create our new **`render`** function as follows:

```
render() {
  return (
    <div className="App">
      <button onClick={this.toggleSecretMessage.bind(this)}>
    Click to show the secret message!
    </button>
    { this.secretMessage() }
    </div>
  );
}
```

    This allows React to know that when the button is clicked, the code being executed remembers that it is inside of the component we are building. That means that any references to **`this`** in our code will be replaced with the component instead of whatever called that function, like our **`onClick`** handler function.

12. Render the secret message.

    There are two approaches to this problem, and we will discuss the pros and cons of each. We will start with an inline conditional in JSX with a ternary operator associated with it:

```
{ this.state.showSecret ? this.secretMessage() : null }
```

    This gives us a conditionally rendered secret message (when the browser refreshes or you navigate to the page, it will now be hidden by default), which is the correct behavior. We are using the ternary operator here to make the code a little cleaner and because **`if`** statements inside of JSX are not even syntactically correct. We return a **`null`** in case there are no values so that React knows not to attempt to add anything to the DOM. This is fine for very simple examples and can arguably help to keep your code clean, but there is another way for us to write this conditional rendering statement.

13. Return back to our **`secretMessage()`** function definition. Add a **`guard clause`** to our function.

> **NOTE**
>
> A **`guard clause`** is a simple statement at the top of your function that verifies inputs, properties, or state, and returns early to avoid unnecessary work.

The new code for this function is as follows:

```
  secretMessage() {
    if (!this.state.showSecret) { return; }
    return (
    <div className="secret-message">
      I am the secret message!
    </div>
    )
  }
```

Note that at the top of that function, we now have a quick conditional that checks the state for **`showSecret`**. If it's **`false`**, we return nothing out of the function and call it a day. If it's **`true`**, we continue on with rendering the secret message.

14. Call our **`render`** function and modify the JSX to instead just call **`this.secretMessage()`** without the inline conditional:

```
render() {
  return (
    <div className="App">
    <button onClick={this.toggleSecretMessage.bind(this)}>
    Click to show the secret message!
    </button>
    { this.secretMessage() }
  </div>
  );
}
```

When you visit the page now (or when the browser refreshes), you should no longer see the secret message. This is *nearly* perfect, but there is a little more we can do. If you look at the text on the button, it just always displays **`Click to show the secret message!`**. If we are making elements on the page state-aware, we should apply the same rule throughout to maintain the consistency of our app.

15. In the code for the button, we will use an inline conditional rendering statement to change what text is displayed to the user based on the component's state:

```
<button onClick={this.toggleSecretMessage.bind(this)}>
  Click to { this.state.showSecret ? "hide" : "show" } the secret
message!
</button>
```

    Now when we visit the page, by default, we get the click to show button text and the hidden secret message, and when we click the button, we instead get the secret message and a click to hide text on the button instead. When we first load the page, we will see this:

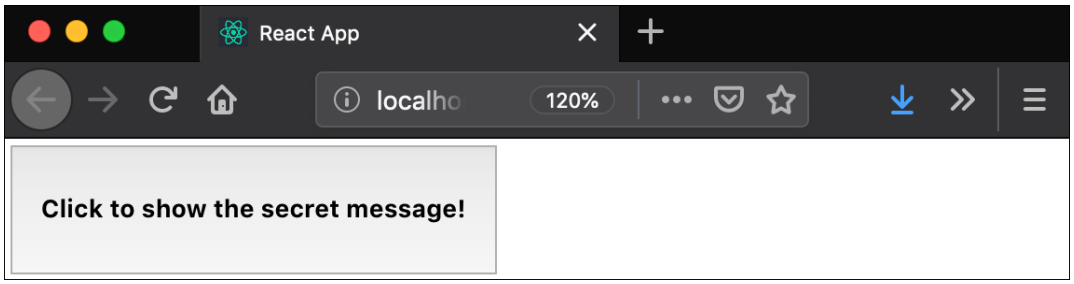

Figure 3.4: App showing the button

    And when we click on the button, we will get this:

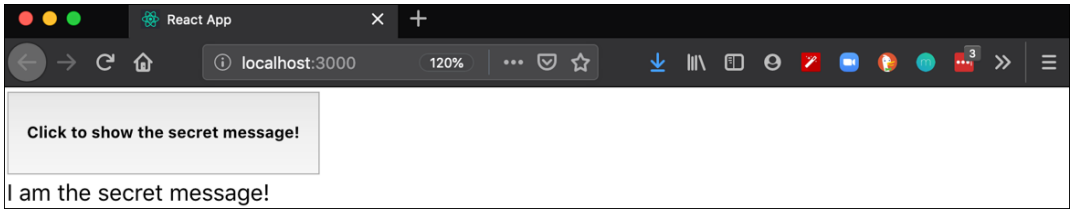

Figure 3.5: App showing the secret message

With that, we have completed writing our first exercise with conditional rendering.

NESTED CONDITIONAL RENDERING

We have implemented our first example of a dynamic app with conditional rendering, but there is a lot more we can do. In the preceding exercise, we just have a single state and a single portion of the code that is conditionally rendered, but what if we wanted to expand that further or have even more complex application states that we want to be represented in the UI?

Conditional rendering is very useful because we can actually use it at multiple nesting levels to have an app structure that is entirely dynamic.

In the next exercise, we are going to build out a quiz app where different questions will appear depending on your answers. We will be recording the state and modifying it as the user answers each question along the way.

EXERCISE 3.02: BUILDING A CONDITIONAL QUIZ APP

Now we need to think about how we want to build up this *quiz app* for the user. We will display each question and the choices for each; when the correct answer is chosen, we will display a message to the user indicating that they chose correctly, and when they pick the wrong choice, we will show a message stating that they chose incorrectly.

We will have a list of questions that we are going to use for our *quiz show*. To do that, we will need to set up a data structure for those questions in the following format:

- **question**: This is a string that will be presented to the user.
- **possibleAnswers**: This will be an array of strings. These are the two possible choices we will present to the user for them to choose from.
- **rightAnswer**: This represents the correct answer for the user.
- **playerChoice**: This represents the answer the player chose.

1. Start off by creating our app. Let's name it **`exercise2`**:

```
$ npx create-react-app exercise2
$ cd ./exercise2
$ yarn start
```

2. Delete **`src/logo.svg`** since we won't need it.
3. Clear out the contents of **`src/App.css`** (but keep the file). We will get back to this later.
4. Remove the **`import`** for **`src/logo.svg`** since we removed that, and add the **`{ Component} import`** to the React **`import`** at the top:

```
import React, { Component } from 'react';
import './App.css';
```

5. Replace the **App** component with a **class** component. This can just include a header with the name **Quiz Show** instead:

```
class App extends Component {
  render() {
  return (
    <div className="App">
    <h1>Quiz Show!</h1>
    </div>
  );
  }
}
```

6. Let's build our initial state with the list of questions with the appropriate answers first:

App.js

```
13  {
14    question: "What animal barks?",
15    possibleAnswers: ["Dog", "Cat"],
16    rightAnswer: "Dog",
17    playerChoice: null
18    },
19    {
20    question: "What animal is more closely related to a tiger?",
21    possibleAnswers: ["Dog", "Cat"],
22    rightAnswer: "Cat",
23    playerChoice: null
24    },
```

The complete code can be found here: https://packt.live/3fCGJA4

We start off with the **constructor**, passing in the **props** and handing those off in a call to **super()**. Next, we set up an initial state where we keep track of the player's score (how many questions they have answered correctly) and initialize it to zero.

7. Add a function to display the questions to the user. Start by writing a utility function to display a single question; it should take an index to the right question in our state as its only argument:

```
displayQuestion(index) {
  const question = this.state.questions[index];
  return (
  <div className="question-display">
    <p className="question">
    {question.question}
```

```
      </p>
      <br />
      <button className="question-choice">
      {question.possibleAnswers[0]}
      </button>
      <button className="question-choice">
      {question.possibleAnswers[1]}
      </button>
      <br />
      <p className="result-correct">
      Your answer is correct!
      </p>
      <p className="result-incorrect">
      Your answer is incorrect!
     </p>
    </div>
    );
  }
```

We try to keep each of the portions of the display separate, complete with distinct CSS classes, to make it easier to change the look and feel later without making major modifications to the structure. It's easier to scope out the component by verifying everything displays correctly before moving on to the phase of separating the displays out depending on how the user answers.

8. Let's put the calls to **`displayQuestion()`** into our **`render`** function next. For now, we will just have four calls to **`displayQuestion`** in our **`render`** function, each passing in a different index:

```
  render() {
    return (
    <div className="App">
      <h1>Quiz Show!</h1>
      <hr/>
      {this.displayQuestion(0)}
      {this.displayQuestion(1)}
      {this.displayQuestion(2)}
      {this.displayQuestion(3)}
    </div>
    );
  }
```

This should give us our initial quiz show display, as in the following screenshot, after the browser refreshes:

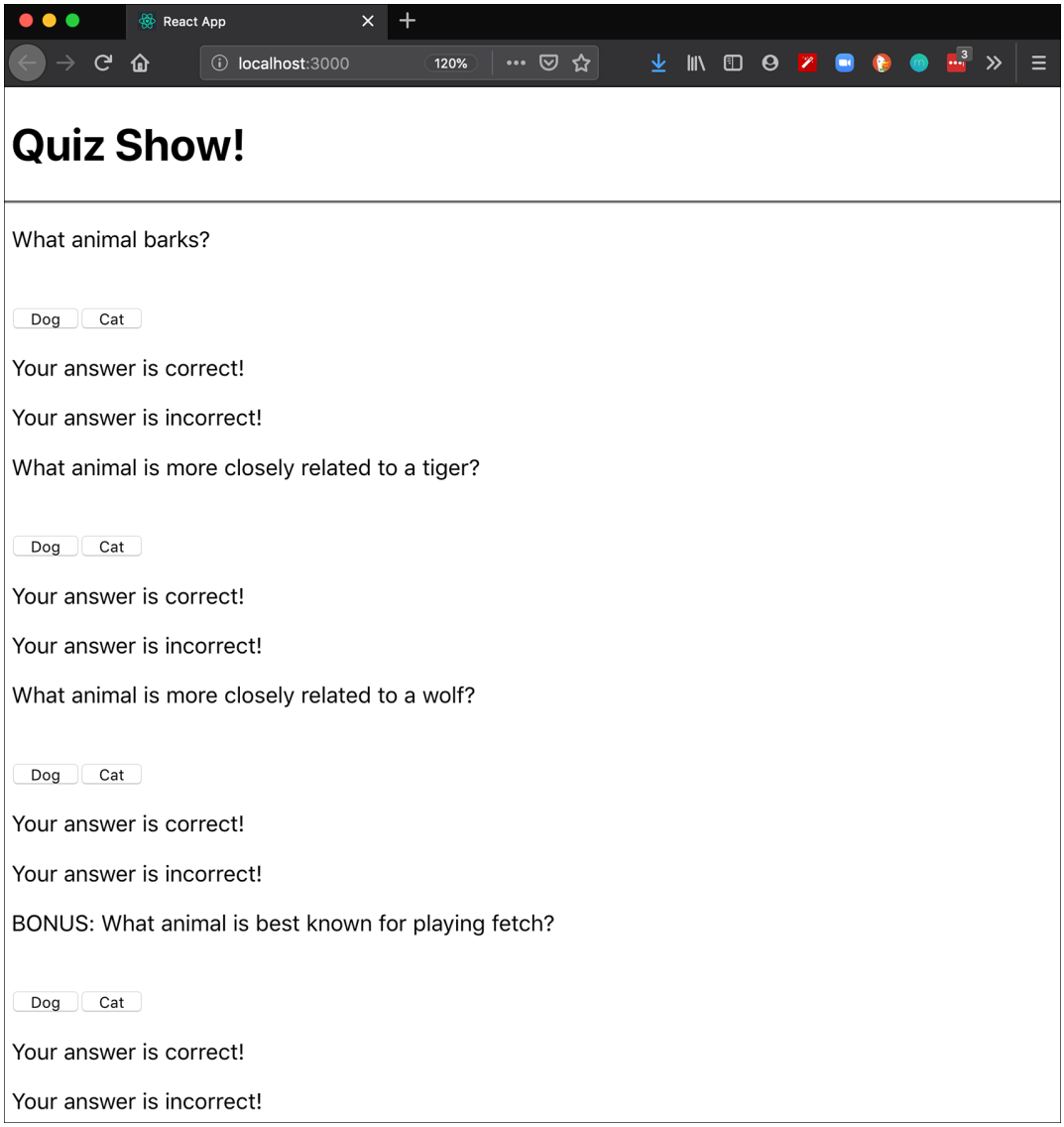

Figure 3.6: Quiz show app

9. Clean up the style of the page a little bit to make the elements more easily distinguishable from each other.
10. Open **`src/App.css`** and let's start adding classes for each of the elements.

Each question display should have its own background and be separate from each other question display. The questions themselves should stand out and be bold, and the buttons should be made larger and easier to interact with. Finally, the right and wrong answers should have appropriately matching color schemes. We will use green and red colors here for these elements. You can use the following style sheet as the contents for **`src/App.css`** or instead choose your own design:

```
.question-display { background: #ddd; border: 2px solid #ccc;
margin: 20px; padding: 10px; }
.question { font-weight: bold; }
.question-choice { height: 5em; width: 20em; margin: 10px; }
.result-correct { color: #272; }
.result-incorrect { color: #922; }
```

This should give us a much cleaner UI, as per the following screenshot:

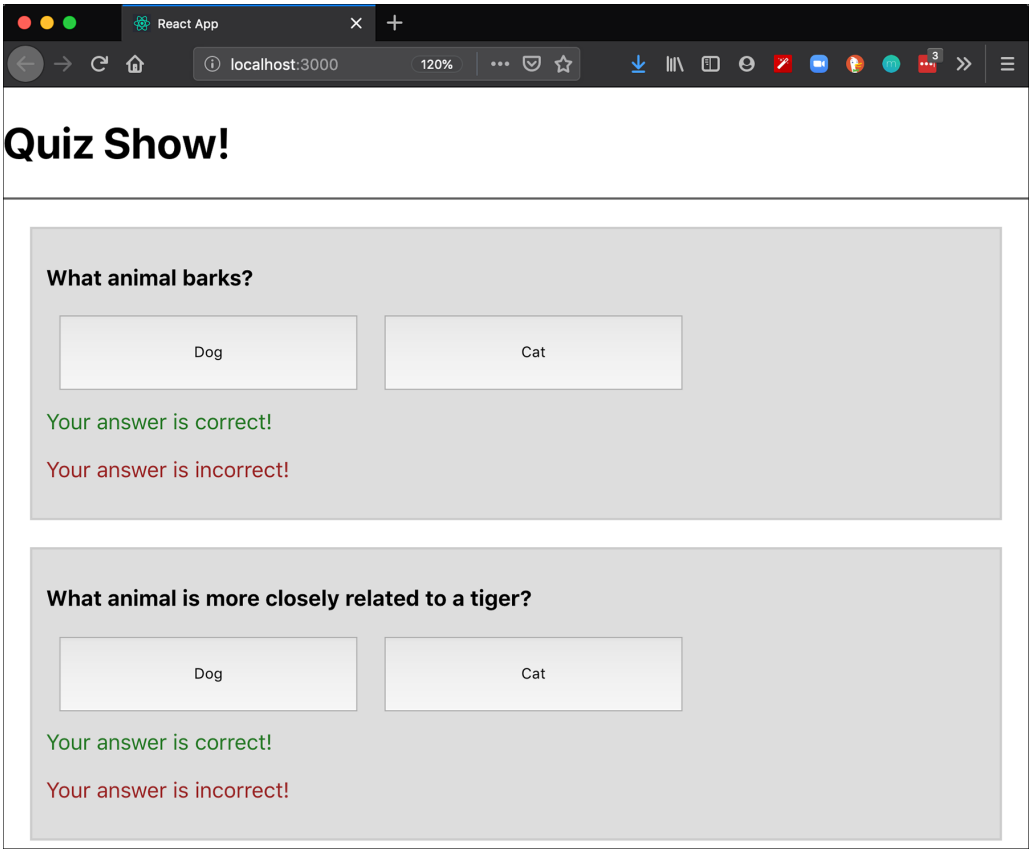

Figure 3.7: Quiz show app with the CSS

11. Start off by writing out an **`answerQuestion`** utility function to answer each question, which will take an index and the answer chosen as its arguments.

    In this function, we should find the question that the user answered, update the **`playerChoice`** property of that question, update the question list in the state, and then update the cached player score as a callback to update the state's question list:

```
answerQuestion(index, choice) {
 const answeredQuestion = this.state.questions[index];
 answeredQuestion.playerChoice = choice;
 const allQuestions = this.state.questions;
 allQuestions[index] = answeredQuestion;
 this.setState({
  questions: allQuestions
 }, () => {
  this.updatePlayerScore();
 });
}
```

> **NOTE**
>
> We need to use a callback here where we recalculate a player's score per question, otherwise, if we just try to increment/decrement based on the player's response, they could click the same answer multiple times and rack up an infinite number of points.

    We have not written our **`updatePlayerScore`** function yet, but we will do that soon.

12. Add a call to **`answerQuestion`** to the buttons as well. Back in **`displayQuestion()`**, we will update the button displays to the following code:

```
<button className="question-choice" onClick={() =>
this.answerQuestion(index, question.possibleAnswers[0])}>
  {question.possibleAnswers[0]}
</button>
<button className="question-choice" onClick={() =>
this.answerQuestion(index, question.possibleAnswers[1])}>
  {question.possibleAnswers[1]}
</button>
```

13. Write that **`updatePlayerScore`** function.

    This should just filter the list of questions down to the questions that have been answered correctly and assign a point for each. So, for the player score, we can just set that to the length of the correctly answered questions.

    To ensure the logic is working correctly, use a **`console.log`** statement and ensure that the score matches the number of correctly answered questions:

```
updatePlayerScore() {
  const playerScore = this.state.questions.filter(q => q.rightAnswer
===
     q.playerChoice).length;
  this.setState({ playerScore });
  console.log("New player score:", playerScore);
}
```

If we play around with this a little bit, we should be able to get points for correctly answered questions, lose points for incorrectly answered questions, and not somehow end up with more than four answered questions. The output is as follows:

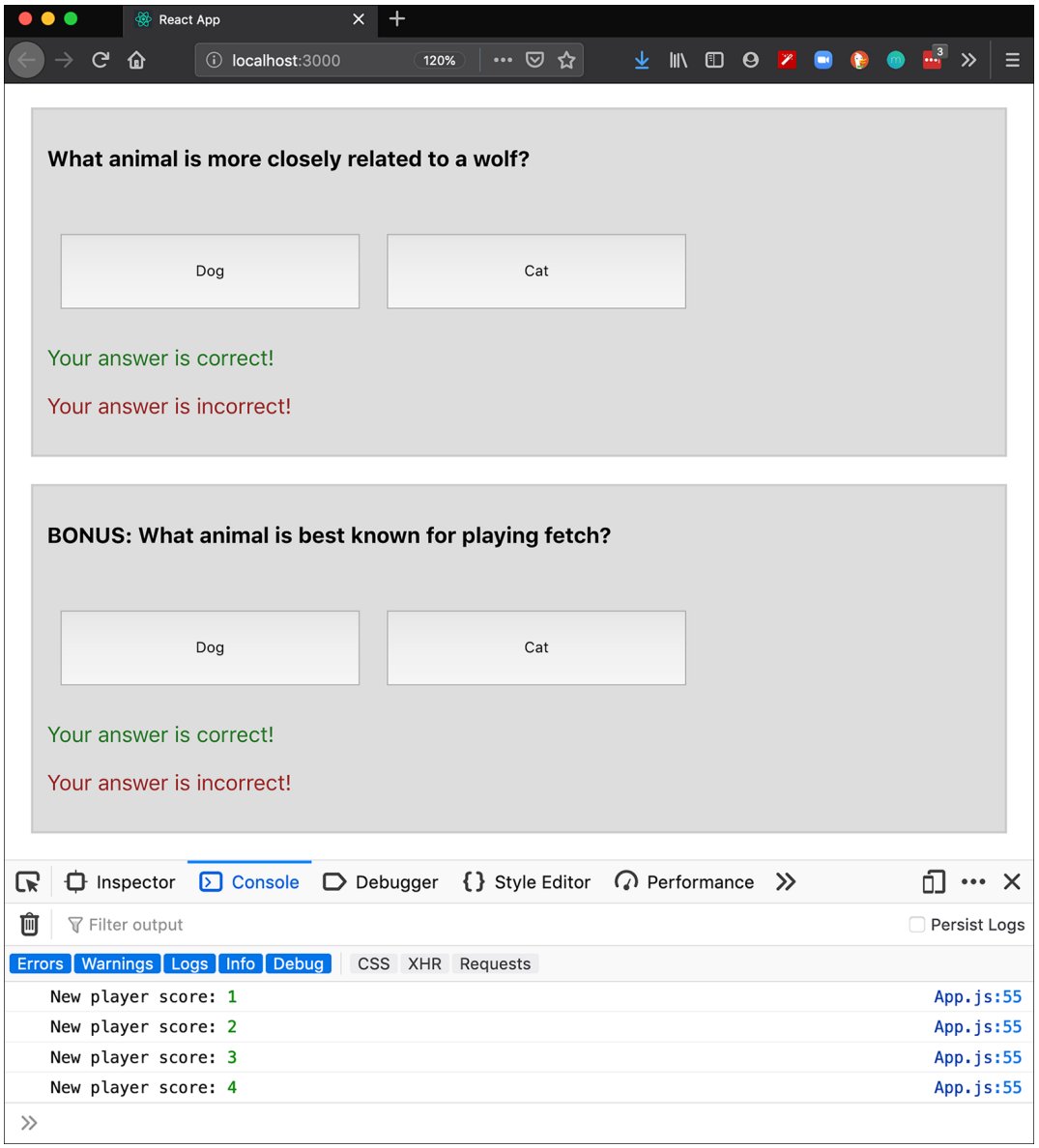

Figure 3.8: Console of the Quiz show app

14. Extract the portion of the template that displays the right/wrong answer result into its own helper function, which we will call **`displayResult`**.

    It should take the index of the question as the only argument. If the player has not answered yet (basically, if the **`playerChoice`** property is null), we will display nothing. If the answer is correct, we will display the correct result display, and if it is incorrect, we will display the incorrect result display:

```
displayResult(index) {
  const question = this.state.questions[index];
  if (!question.playerChoice) { return; }
  if (question.playerChoice === question.rightAnswer) {
    return (
    <p className="result-correct">
     Your answer is correct!
    </p>
   );
  } else {
  return (
    <p className="result-incorrect">
    Your answer is incorrect!
   </p>
  );
  }
}
```

    Finally, let's make it so we only display the question if the player has answered everything up to that point correctly.

15. Modify our **`displayQuestion`** function to return nothing if the player's score is less than the index, preventing the display of further questions. This is pretty easy for us to write as a single conditional and return at the top:

```
if (this.state.playerScore < index) { return; }
```

If all has gone well, as you answer the questions, you should see each question being loaded in appropriately, as in the screenshot that follows:

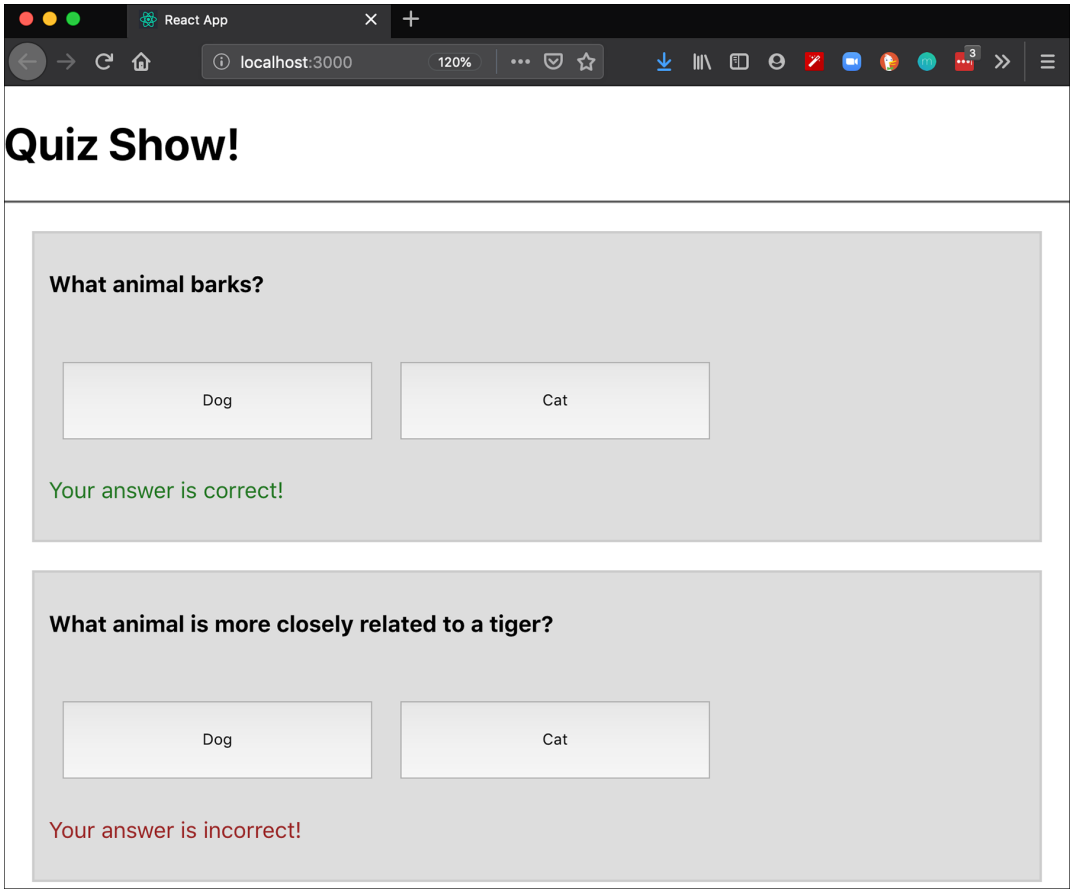

Figure 3.9: Quiz app

And that's it; we now have a working quiz system. Now, let's dive right into how the looping technique works in React.

RENDERING LOOPS WITH REACT

Another critical component of building dynamic apps with React is not just relying on conditional statements but also building upon standard loops in JavaScript to render multiple elements in simple ways. Think back to some of the work that we did as part of the last exercise: in our render call, we had this bit of code:

```
  render() {
    return (
```

```
      <div className="App">
      <h1>Quiz Show!</h1>
      <hr/>
      {this.displayQuestion(0)}
      {this.displayQuestion(1)}
      {this.displayQuestion(2)}
      {this.displayQuestion(3)}
      </div>
    );
  }
```

This is unnecessarily repetitive, and we can clean this up a lot with just a loop. Similar to how writing conditional rendering statements went, we can also write these either via inline statements in our JSX or through functions. For example, the preceding snippet could be written inline via the following:

```
  render() {
    return (
      <div className="App">
    <h1>Quiz Show!</h1>
    <hr/>
    {this.state.questions.map((question, index) => this.
displayQuestion(index))}
      </div>
    );
  }
```

Now we are getting a warning message that we should fix:

> **NOTE**
>
> Each child in a list should have a unique *key* prop.

We should absolutely fix that. Before that, though, let's dive a little deeper into rendering loops, because understanding what's going on will help us understand why we need to fix this in the first place.

RENDERING LOOPS

We have used the phrase rendering loops a lot, but what does it actually mean? Essentially, we are telling React that rendering an element is the result of rendering a list of items, instead of just rendering a single item. So, let's say we have a tree structure like this:

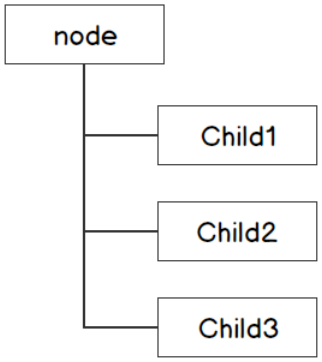

Figure 3.10: Tree structure of a node

If we were to write this out in JSX, it would look something like this:

```
<node>
  <child1 />
  <child2 />
  <child3 />
</node>
```

With the preceding JSX snippet as our example, the tree shown in *Figure 3.10* is simple enough that React can figure out which element needs to change if **`child2`** needs to be updated and not the rest of the tree. For example, it's easy for us to look at the tree example in *Figure 3.10*, point to **`child2`**, and say *this needs to update.* Now, if we were to build that list of children where the children nodes would change dynamically, it would be more abstract and difficult for React to interpret. Say, instead, our tree is as follows:

```
node
 |-- children (length: 3)
```

Our JSX is as follows (this is just pseudocode):

```
<node>
 {elements.map(child => <child />)}
</node>
```

Can you look at this and say *update child 2*? For that matter, how can React handle that? This scenario is where you would typically get the following error message in the console:

```
Warning: Each child in a list should have a unique "key" prop.
```

Let's instead say that each child has a special property on it called **`key`**, and that can tell React exactly where to look to update something. Now our graph is as follows:

```
node
 |-- children (length: 3) [key: 0, key: 1, key: 2]
```

And our JSX pseudocode is as follows:

```
<node>
 {elements.map((child, index) => <child key={index} />)}
</node>
```

`Map` is being used here specifically because we want to return the list of elements as we modify each of them; if we used a construct such as **`forEach`**, then we would not be returning a modified list.

Now, React can tell precisely where the second child is and it can update that without having to bother with the rest of the list of elements. Adding a key property to an element is just as simple as the preceding example: you provide a unique identifier called **`key`**, and it needs to be unique to all elements displayed on the page (not just in that particular subtree of rendered components).

Working with the same example, if we were to go into the **`displayQuestion()`** function, we could add a simple **`key`** property to the **`div`** at the top of the returned JSX with a unique identifier and it would remove that error message – something like this:

```
<div className="question-display" key={`q-${index}`}>
```

This would eliminate the error message completely.

Beyond that, everything else is just applying JavaScript techniques and rules to JSX syntax, something which, at this point, you should be very comfortable with.

EXERCISE 3.03: REFINING OUR QUIZ DISPLAY WITH LOOPS

Using the app that we started to build in the previous exercise, we are going to clean up the code significantly using rendered loops to refactor and refine our exercise. This will be a little bit of a shorter exercise.

1. Make sure the development server is running already, and if it is not, do so via **`yarn start`**:

```
$ yarn start
```

2. To clean up our quiz display, we need to only render the questions from a list instead of us adding each question in manually. We will start off with that same example. Where we had multiple calls to **`this.displayQuestion(index)`**, we will instead have a single-utility **`render`** function call:

```
  render() {
    return (
      <div className="App">
        <h1>Quiz Show!</h1>
        <hr/>
        {this.renderQuestions()}
      </div>
    );
  }
```

   We will also need to write the **`renderQuestions`** function, so let's take care of that while we're at it. In this function, we just need to iterate over each question and call **`displayQuestion`** for it, so we will use a **`map`** function call and pass in the index of the **`map`** call.

3. Let's create the **`map`** function call:

```
  renderQuestions() {
    return this.state.questions.map((question, index) =>
      this.displayQuestion(index)
    );
  }
```

4. Reload the browser window (if it hasn't already happened) and verify that, so far, nothing has changed apart from an error message about there not being a unique key for each item rendered (which we can fix).

5. Let's fix that error message by jumping to the top line of the return statement for **`displayQuestion()`** and adding a **`key`** property:

```
<div className="question-display" key={`q-${index}`}>
```

We still have some non-loop code that we can optimize, specifically in how we present the quiz choices to the user. In that same return statement, if you find the code that presents the choices, we can instead turn this into an inline call to map on **`question.possibleAnswers`**. Don't forget to give each answer choice its own unique key property.

6. Place the following code inside the **`div`** you created previously:

```
    { question.possibleAnswers.map((answer, answerIndex) => (
    <button key={`q-${index}-a-${answerIndex}`} className="question-
choice"
  onClick={() => this.answerQuestion(index, answer)}>
      {answer}
    </button>
   ))}
```

Save the code and everything should still be functioning identically to before and should look identical as well. Our code is now much saner and cleaner, but the functionality remains intact.

The output is as follows:

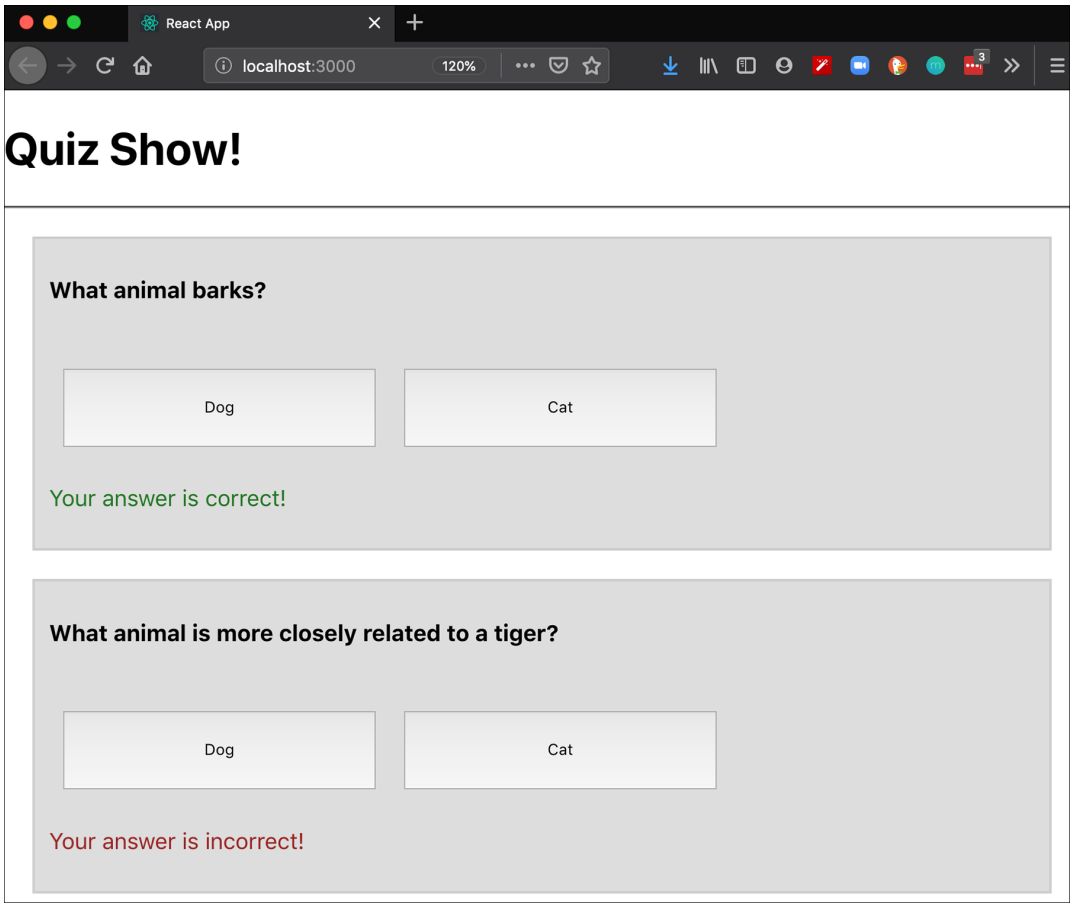

Figure 3.11: Quiz show App

With all the knowledge gained from the chapter, it's time to put everything together that we have learned so far and build a fancy game board in the activity.

ACTIVITY 3.01: BUILDING A BOARD GAME USING CONDITIONAL RENDERING

We are going to build a simple memory match game in React. We will create a list of tiles, randomize them, and assign simple number values to each. The idea is to match up different pairs. The rules of the game are simple:

- When a card is flipped, a number will be displayed, but otherwise will just be a plain backing.
- As we click on each card, we should "flip" the card, and only allow two cards to be flipped at a time.
- When a card is matched, it should be highlighted in green.
- When two cards do not match, the cards are flipped over again.
- Each attempt made by the player should be tracked; the player's score at the end is how many flips they made to match all the cards.
- The game is over when all matches have been made.

The CSS to use for building the app is provided here:

```
.Tile {
width: 200px;
height: 200px;
font-size: 64pt;
border: 2px solid #aaa;
text-align: center;
margin: 10px;
padding: 10px;
float: left;
cursor: pointer;
}
```

The following steps will help you achieve the goal:

1. Set up the game board to display the cards. We will start with having it display **`12`**.
2. Create your React app with its constructor and define the tile count.
3. Write the function that will generate the app when the **`New Game`** button is clicked. Several steps will be required to complete this. Steps *4-9* will help you to write the required function.

4. Start off with a blank list of tiles.
5. Generate a pair of numbered tiles per player, but only up to the max tile constant you created earlier.
6. Each tile should have properties for whether the tile has been flipped over or not, whether the tile has been matched or not, and what number to display as the *value* of the tile.
7. After building the tile, the tile should get added to the list of tiles.
8. After building the list, the tiles should be randomized.
9. Finally, the state of the application should be updated after this functionality has been built.
10. Set up a click handler for each card that will flip the card when triggered.
11. Set up some basic styling for each card to allow the player to see whether a card is already matched or not.
12. Set up logic to check the current card flipped against a previous card if a card is already flipped over; if they match, add those to the matches.
13. Keep track of every time the player attempts to make a match.
14. Add a condition for when all the cards have been flipped over and the game is done.

There should be a **`New Game`** button to allow the player to set up and start a new game.

The output should look like this:

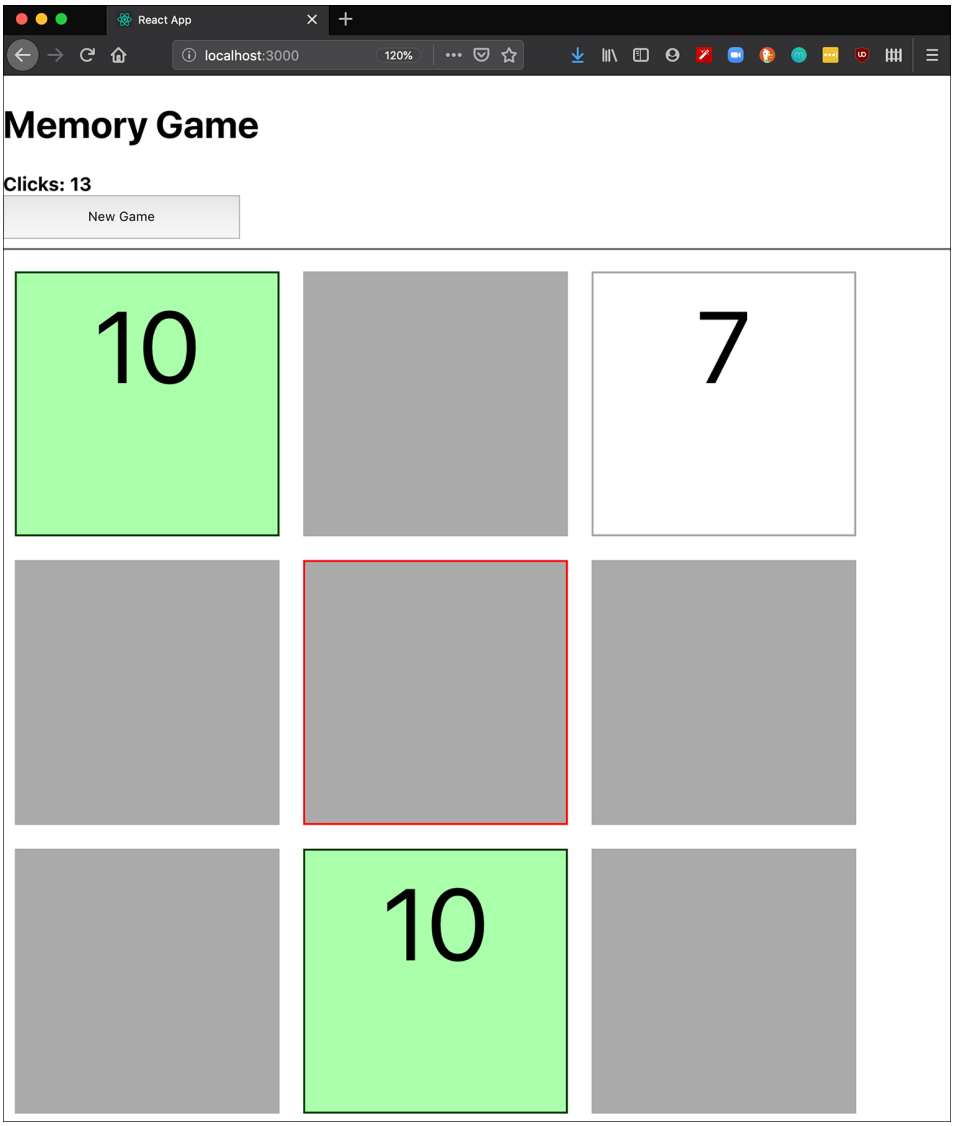

Figure 3.12: Memory Game app

And with that, we have built a memory game using conditional rendering in React.

> **NOTE**
>
> The solution of this activity can be found on page 615

SUMMARY

Ultimately, like so many other parts of React, understanding and mastering the framework hinges on being able to apply the rules and syntax of JavaScript that you are already familiar with. It's part of the strength of React; nothing you do is so intensely foreign that it's difficult to reason about. You can almost always take something you know in JavaScript and apply it, with few modifications. In the span of this chapter, we focused on applying conditionals and loops, two very basic building blocks of all JavaScript code bases and applied them to React and JSX.

These are some of the primary scenarios that you will run into when building React applications. Your applications will hinge on their ability to, ironically enough, react to dynamic states. You will rarely write static components and code in modern web applications and React not only knows this but discourages it. In the next chapter, we will learn how to use lifecycle methods to manipulate the state in class components.

4

REACT LIFECYCLE METHODS

OVERVIEW

In this chapter, we will implement various types of lifecycle methods in React. You will have a solid understanding over the entire React component lifecycle and how to use each of the lifecycle functions to their fullest. You will be able to implement lifecycle methods to set the state of the component when the component is initialized the first time. We will implement the mount lifecycle (**`constructor`**, **`getDerivedStateFromProps`**, **`render`**, and **`componentDidMount`**), the update lifecycle (**`getDerivedStateFromProps`**, **`shouldComponentUpdate`**, **`render`**, **`getSnapshotBeforeUpdate`**, and **`componentDidUpdate`**) and the **`unmount`** lifecycle (**`componentWillUnmount`**) through several exercises and use them to create a messaging application.

INTRODUCTION

In the previous chapters, while building our components, we have used lifecycle methods multiple times without knowing it. For example, we have used the **`constructor()`** and **`render()`** methods in every class component we have built so far. We did not call or overwrite these methods specifically but React has behind-the-scenes functionality for performing this.

In this chapter, we will talk a little bit about what that means and explore the different methods, where they fire off during building a component, mounting them onto the Document Object Model(DOM), rendering them, and then updating them beyond that. You can also implement events that need to occur when a component is removed from the DOM using the unmount lifecycle method.

It's worth noting, however, that you cannot overwrite or call lifecycle methods explicitly in functional components; only class-based components have this functionality. You cannot override them because, being functions, they lack the ability that classes in JavaScript have; classes can define functions on each individual instance. There is no inheritance or properties; the functional component just runs the code on render.

Let's start off the chapter with an overview of the component lifecycle methods in class components.

OVERVIEW OF THE COMPONENT LIFECYCLE

The React component lifecycle broadly comprises of two different segments:

- The initial **mount** lifecycle (when React attempts to build the component for the first time and inserts it into the DOM)
- The **update** lifecycle (after the component has been loaded and rendered for the first time and now is in a state of watching and updating for changes), which loops back on itself over and over

Let's have a look at the various methods that make up each of these component lifecycle stages in the following diagram:

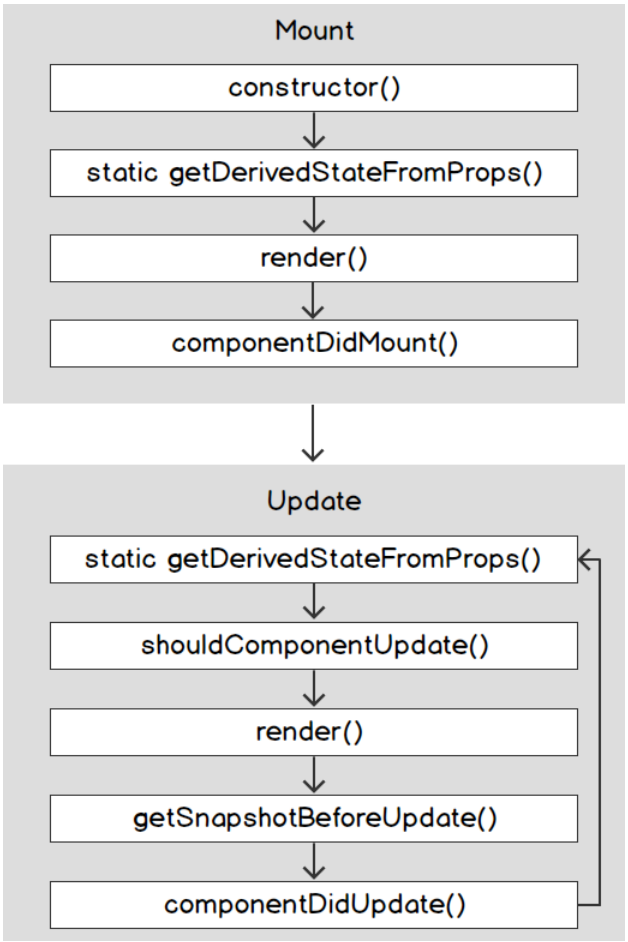

Figure 4.1: Flow of React lifecycle methods

All methods comprising the mount lifecycle are executed first. Then, the program flow proceeds to the update life cycle methods, which are executed repeatedly as long as the component is alive. Let's take a look at the lifecycle itself in this diagram:

In *Figure 4.1*, we will focus almost entirely on the following pointers:

- **`constructor()`**: The constructor of our component, the first thing that gets called (which you have seen quite a bit in the previous chapters).
- **`render()`**: This is called when the component is rendered to the DOM; again, something you have had already seen.

- **`componentDidMount()`**: This is when the component has been rendered and included in the DOM; this one is new.
- **`componentDidUpdate()`**: This is when the component had a change that triggered an update and was re-rendered to the DOM (for example, if a prop or the state changed); another new one.

We will rarely use the following methods:

- **`static getDerivedStateFromProps(props, state)`**
- **`shouldComponentUpdate(nextProps, nextState)`**
- **`getSnapshotBeforeUpdate(prevProps, prevState)`**

The lifecycle methods in the preceding diagram are specifically executed when the component is alive; that is, included in the browser's DOM and being rendered. When a component unmounts, the **`componentDidUnmount()`** method is called. That call happens right before the component is totally removed from the DOM.

As mentioned previously, everything in **`mount`** method happens first, and then everything in **`update`** method happens repeatedly in a loop. Basically, when a change happens, the loop retriggers, first by getting the new state, including any state that needs to be derived from **`props`**, and then it triggers the **`shouldComponentUpdate`** method. From there, the rest of the lifecycle fires off sequentially until the program flow encounters the **`componentDidUpdate`** method, at which point the React component will essentially pause in the cycle until another update happens and a new event needs to be triggered.

It's important to note that all of this is only a lifecycle in class components. Functional components have no internal functions that they can use since they essentially just render the JSX syntax and have no other structure to them.

Ultimately, you should remember the following basic flow of implementing the lifecycle methods as the following:

MOUNT: (constructor -> render -> componentDidMount) -> UPDATE: (render, componentDidUpdate) -> UPDATE (repeat until component removed) -> UNMOUNT: (componentWillUnmount)

EXERCISE 4.01: IMPLEMENTING LIFECYCLE METHODS

In this exercise, we will spend a little time implementing everything that we just learned about how the lifecycle flows. We will avoid the rarely used lifecycle methods and stick to the main lifecycle methods. Since the primary motive of the exercise is to implement lifecycle methods, the exercise in itself is simple, and we will be rendering the lifecycle stage using **`console.log`** statements. Let's see how:

1. Start off by creating a new React project, which we will call **`lifecycle`**. Start the project, and take a look at the browser window that should have opened for you automatically:

```
$ npx create-react-app lifecycle
$ cd lifecycle
$ yarn start
```

2. Delete **`src/logo.svg`** and delete the contents of **`src/App.css`**.

3. In order to disable strict mode, change the render part in **`src/index.js`** as follows:

```
ReactDOM.render(<App />, document.getElementById('root'));
```

4. Clean out the contents of the **`App`** component and replace them with a class component instead. Since we are using a class component here, we will need to add the following code:

```
import React, { Component } from 'react';
import "./App.css";
class App extends Component {
  render() {
  return (
    <div className="App">Hello Lifecycle</div>
  )
  }
}
export default App;
```

This should give us an initial component that looks like the following:

Figure 4.2: Initial component

5. Add our first lifecycle **`console log`** statement by adding **`console.log()`** to a constructor (you will need to add the constructor, too):

```
constructor(props) {
super(props);
  console.log("Constructor");
}
```

When your browser reloads, if you open up the JavaScript console, you should see the **`Constructor`** statement appear in your log:

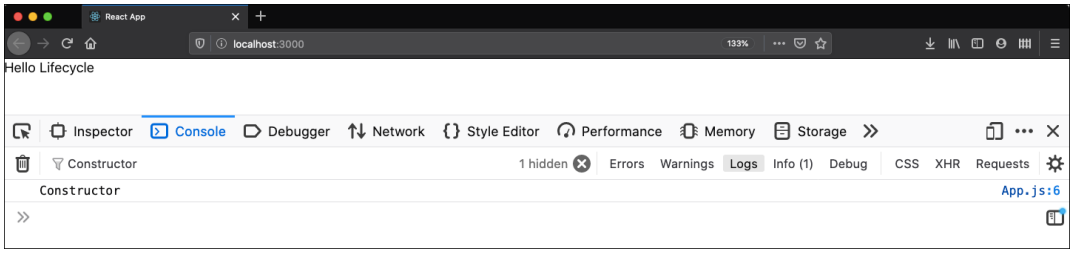

Figure 4.3: Console log of the component

6. Add a **`componentDidMount()`** function to the component. In that, you are going to add a **`console.log()`** call that outputs **`Component Did Mount`** to the log:

```
componentDidMount() {
console.log("Component Did Mount");
}
```

7. Add a **`console.log()`** call to your **`render()`** function before you return the JSX. The output will just be **`Render`**. Remember that **`render()`** is earlier in the component lifecycle than **`componentDidMount()`**, so this log statement will actually happen before the one we wrote earlier:

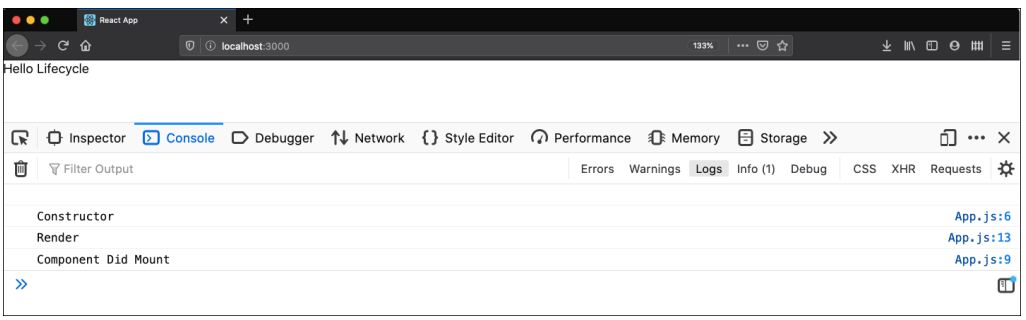

Figure 4.4: Console to show the messages for the lifecycle methods

After *steps 1 through 5*, we have implemented all of the mount lifecycle methods, so now we can move into the **`update`** loop lifecycle methods. To do this, we will need to set up a particular state that will allow us to watch for updates appropriately.

8. First, go into the **`constructor`**. Add a new state here with a **`state`** property called **`cycle`** that will start with a value of **`0`**.

9. Add a **`setInterval()`** statement that will update the component's cycle by **`1`** every second (**`1000 milliseconds`**):

```
constructor(props) {
  super(props);
  console.log("Constructor");
  this.state = { cycle: 0 };
  setInterval(
    () => this.setState({ cycle: this.state.cycle + 1 }),
    1000
  );
}
```

While this is technically enough now to see the updates in the log, we are going to quickly add something to the **`render`** function that will tell us the current value of the **`cycle`** state.

10. Update the **`render`** function to show the current value of the **`cycle`** state:

```
render() {
  console.log("Render");
  return (
      <div className="App">Hello Lifecycle: Cycle {this.state.
cycle}</div>
  );
}
```

11. Add the **`componentDidUpdate()`** method to your class component that will print **`Component Did Update`** in the console

With all of this done, we should see each of the **`update`** cycle methods looping in our console after the initial mount cycle methods:

```
componentDidUpdate() {
  console.log("Component Did Update");
}
```

The output is as follows:

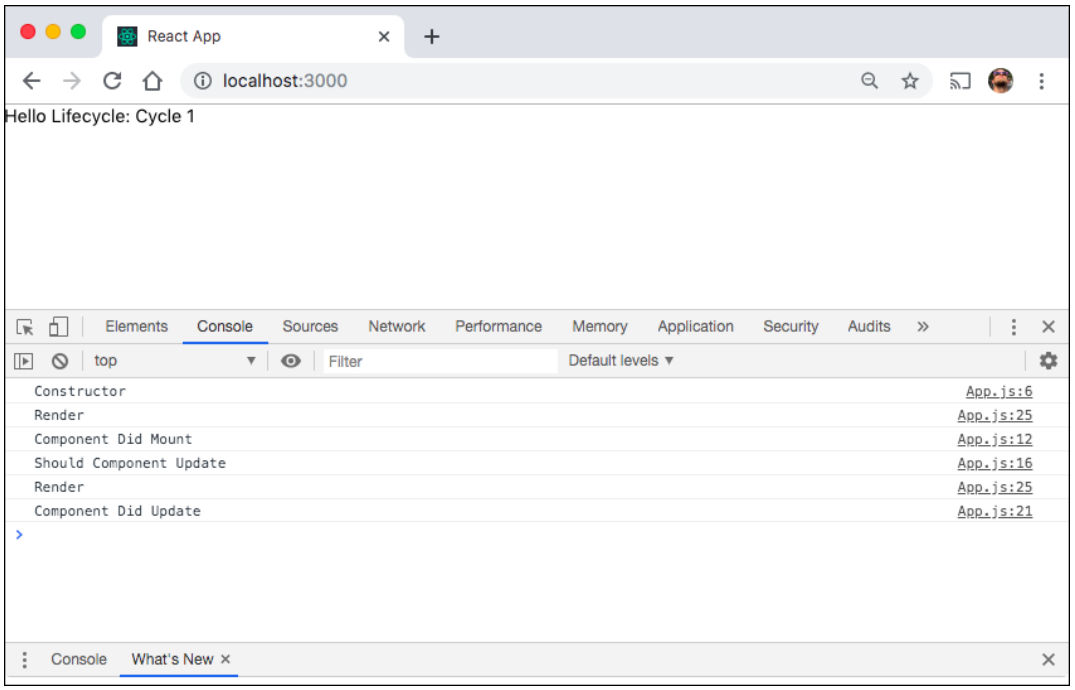

Figure 4.5: Console to show the output

As you can see, the **`console.log`** statement shows the output as expected. You can see just how the initial mount lifecycle leads into the looping **`update`** lifecycle. Now, let's explore each lifecycle method individually to give us a greater insight into what each one is responsible for.

THE MOUNT LIFECYCLE

The **mount** lifecycle is called twice: before and immediately after React renders the component into DOM. Note that *mounting* alone happens only once; there is no scope in a React world where you can *remount* a React component into the DOM. When we say that React renders to the DOM, it means that this is when React processes the JSX, converts it to HTML, and shows it on the browser.

Mounting is where a lot of the functionality will take place specific to initializing a component's state at the time of loading. Mounting happens when your app loads up for the first time, when you navigate to a particular component using something like React router, or it could be something like when you add a component to a page dynamically, like through conditional renders or loops. The first one of these functionalities is one you should be very comfortable with at this point: the constructor.

CONSTRUCTOR()

We have used the **`contructor(props)`** method in previous exercises throughout this book to initialize the overall state of our React component and to set up the call to the parent constructor with the **`props`** arguments that were passed in. Previously, we have used the constructor, but never with a focus on what was actually happening as part of the lifecycle.

The call to the constructor happens almost immediately before the component is first rendered to a page. Essentially, when our code is loaded, JavaScript will look for the first React component that gets rendered (the **`root`** component). Then, each child component below the root is rendered. React starts off each component with the **`constructor`** statement which initializes the **`state`** and the **`props`** for each component along the way and tells React precisely how it needs to handle rendering each of those components.

EXERCISE 4.02: CONDITIONAL RENDERING AND THE MOUNT LIFECYCLE

In this exercise, we'll demonstrate how the mount lifecycle in our components is affected through the conditional rendering techniques we learned in the previous chapter. Let's see how to do that:

1. Start off by creating a new React project; call it **`conditional`**.
2. Start the project and take a look at the browser window that should have opened for you automatically:

```
$ npx create-react-app conditional
$ cd conditional
$ yarn start
```

3. Delete **`src/logo.svg`** and delete the contents of **`src/App.css`**. Strip out the **`import`** statements and code we don't need.

4. Clean out the contents of the **App** component and replace it with a **class** component instead. Since we are using a **class** component here, we will need to change the import statement at the top to also import **{Component} from react**:

```
import React, { Component } from 'react';
import "./App.css";
    class App extends Component {
    render() {
    return (
              <div className="App">Hello Conditional</div>
          );
        }
      }
export default App;
```

By running the app (via **yarn start**) and opening up our browser to http://localhost:3000, we will see the initial component, like the following screenshot:

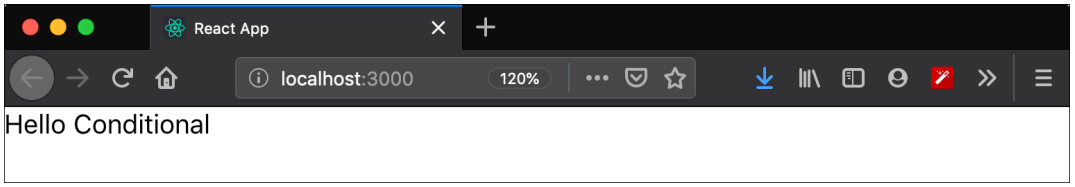

Figure 4.6: Initial component

5. Add a new class component to a new file, **src/LifecycleTest.js**, called **LifecycleTest**. We'll need the standard React **import** statements and an **export default** statement as well.

   The component will just have a simple constructor that outputs a log statement and a simple **return** statement that just returns any text wrapped in a paragraph tag:

```
import React, { Component } from 'react';
class LifecycleTest extends Component {
  constructor(props) {
  super(props);
  console.log('LifecycleTest Constructor');
}
render() {
  return <p>I only show up if the conditional is true!</p>;
```

```
    }
  }
export default LifecycleTest;
```

6. Add the **`LifecycleTest`** import to **`src/App.js`**:

```
import LifecycleTest from "./LifecycleTest";
```

7. Change our **`App`** component to be able to conditionally render this new component of ours. Add a single line to our **`return`** statement using a **`boolean`** value, the and operator, and a reference to the **`LifecycleTest`** JSX component:

```
  render() {
    return (
      <div className="App">
        Hello Conditional
          {true && <LifecycleTest />}
      </div>
  );
}
```

When we take a look at the JavaScript console, we should see the **`console.log`** statement from the constructor of the **`LifecycleTest`** call:

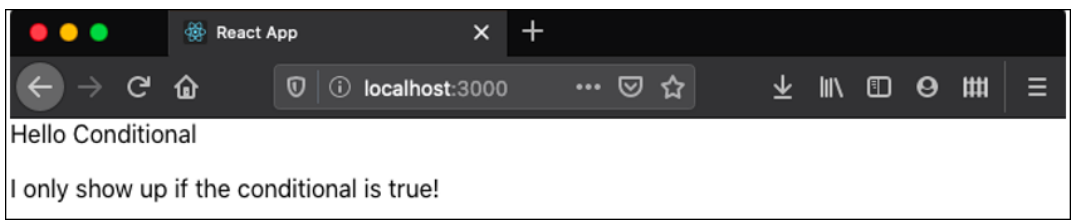

Figure 4.7: App showing the conditional message

8. Change the **`boolean`** statement to **`false`** instead, and take a look at the **`log`** statement; now you should not see the constructor's **`console.log`** statement:

The output is as follows:

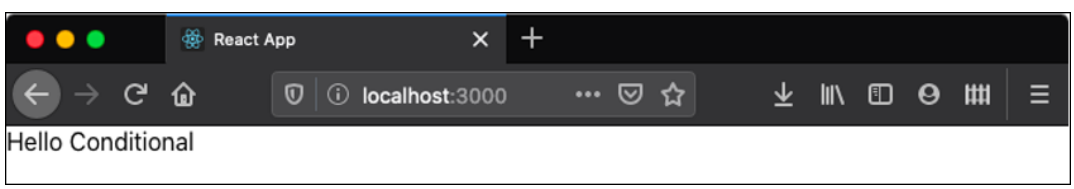

Figure 4.8: Conditional statement not shown

With that, you can see how the mount lifecycle method is affected by conditional rendering. It's important to see how conditional rendering can affect lifecycle methods, since you may need to rely on knowing when components are getting mounted and rendered into your DOM.

RENDER()

The **`render`** function is another lifecycle method that we should be very comfortable with at this point. **`render`** is the point where React converts the JSX that represents our component to the DOM. But up until now, we have not really used it much for anything outside of rendering JSX with a **`return`** statement. The reality is that **`render()`** is just a normal function and can actually have a good amount of other logic inside of it. For example, you can set locally scoped variables for use in your final JSX that gets rendered. There is a lot that you can do, but remember that this function, in terms of the React lifecycle, is called when the component actually gets rendered, and that this function must return JSX.

Now, where this gets a little odd is when you are talking about functions and events that need to happen during or around the time when you render your component. For example, what if your component needs to call out to some external service? You don't know how long that external service will take to respond to you, so what do you do?

For example, let's say we take a call to an external service that takes **`5`** to **`10 seconds`** to return data back to us, and we have that inside of our **`render()`** call:

```
render() {
  const data = fetchDataFromExternalService(); // This takes 5-10 seconds
to return
  return (<div className="App">Response from service: {data}</div>);
}
```

When React goes to render the component onto the page, you will see a huge delay before everything else shows up on your page. This creates a really awful user experience, so it really underscores the importance of keeping your **`render()`** calls light and not dependent on anything that you can't control the performance of. The good news, however, is that there is a component lifecycle method that is perfect for dealing with these sorts of situations.

COMPONENTDIDMOUNT()

The **`componentDidMount()`** lifecycle method is better suited for functions that take an indeterminate amount of time to load. **`componentDidMount()`** is called after **`render()`** is called, so you don't get any weird behavior where the component takes a long time before it actually shows up on your page. As a result, this lifecycle method has become the standard for when you want to load anything into the state that is the result of an AJAX call or anything that involves any sort of long-running code.

One thing worth noting, however, is that your calls that modify the state might take a very short amount of time. If this happens, you might sometimes see a *flicker* effect that occurs when you display a placeholder for things that are only conditionally rendered when the state is updated with data. This is generally considered to be bad practice and you should instead use a placeholder **`render`** (such as a little bit of text that says **`Loading...`** or displays a spinner when the page is loading).

Another common best practice is to render an element that informs the reader that the page or content is still loading, rather than having the initializing state to be empty. If you do this, it will give the impression that the component has no data, and then when the AJAX call returns, it suddenly has data. This can be confusing for the end user, and generally it's better to be clear about the transitional states in your component rather than having a binary loaded/not loaded state. For example, think about when you use an email client. If it displays **`No emails received!`** when you first load the page, and then the load happens, and suddenly you have over a thousand emails in your inbox. Such occurrences can be misleading for the user and they might not trust the information they see. Instead, picture that same email client, but when it is loading, we have rendered an element that informs us that the data is loading. This leads to a much better user experience. Now, the user can trust the information that you are displaying to them.

THE UPDATE LIFECYCLE

The **`update`** loop segment of the lifecycle is something that lives for the entire lifetime of the component (where the lifetime of a component is if it needs to be displayed in the DOM). This loop repeats every time that the component needs to be updated, which is basically whenever any data that affects what needs to be loaded and displayed as the DOM changes which will then start a lifecycle method specifically to determine what changes need to be displayed. For example, if you modify the **`shouldComponentUpdate()`** lifecycle function to just always return **`false`**, then as far as React is concerned, no changes should ever retrigger the update loop. Similarly, if you always return **`true`**, like we did in our first exercise, then React will assume every state or props change needs to retrigger re-rendering.

RENDER()

We have already discussed the **`render`** function, but it is worth mentioning that this is when the **`render`** function gets called each time the component updates. So, after the initial render that happens in the **`mount`** lifecycle, this gets called in the **`update`** lifecycle.

COMPONENTDIDUPDATE()

The **`componentDidUpdate()`** lifecycle method, despite having a similar function signature to **`getSnapshotBeforeUpdate`**, is used a little more frequently. The function definition is as follows:

```
componentDidUpdate(prevProps, prevState, snapshot)
```

It allows you to react to a component updating from changes in **`props`** and/or state, which can be helpful if you need to update some state management code or to make calls to remote services, for example.

Also, there are a few arguments here to deal with. The **`prevProps`** parameter is the shape of the component's props before the update happened and **`prevState`** is the shape of the component's state before the component updated. The **`snapshot`** argument is a bit trickier; picture it as a virtual representation of the component in the update. Also, the **`snapshot`** argument is only needed if your component also implements the **`getSnapshotBeforeUpdate()`** function, which is incredibly rare. If you are not using that, you can just leave the snapshot argument out completely.

One word of caution, however, is that if you are doing anything that would then update **`props`** or **`state`**, you need to wrap those calls in some sort of conditional. If you don't, you'd update the **`props`** or **`state`**, which would trigger the update lifecycle, leading to this function, where you'd update the props or state, which would trigger...well, you get the picture.

In the next exercise, we will perform an exercise that will simulate this example in greater detail.

EXERCISE 4.03: SIMULATING A SLOW AJAX REQUEST AND PRERENDERING YOUR COMPONENT

As previously mentioned, in this exercise, we will simulate a long loading AJAX call and help demonstrate the importance of using the right lifecycle method to ensure the best possible UI experience for your app.

1. Start off by creating a new React project, which we will call **`profile`**, start the project, and take a look at the browser window that should have opened for you automatically:

```
$ npx create-react-app profile
$ cd profile
$ yarn start
```

2. Clean out the contents of the **`App`** component and replace it with a **`class`** component instead. Since we're using a class component here, we will need to change the **`import`** statement at the top to also **`import Component from React`**:

```
import React, { Component } from 'react';
import "./App.css";
    class App extends Component {
    render() {
     return (
      <div className="App">User Profile</div>
    );
  }
 }

export default App;
```

3. Create four lifecycle methods in your **`App`** component: **`constructor`**, **`componentDidUpdate`**, **`componentDidMount`**, and **`render`** (which we already have):

```
class App extends Component {
  constructor(props) {
    super(props);
  }
componentDidUpdate() {
  }
  componentDidMount() {
  }
  render() {
  return (<div className="App">User Profile</div>);
  }
}
```

> **NOTE**
>
> You may receive a **`useless constructor`** warning message if you refresh the browser at this point. That is fine and expected; we'll add to the constructor soon.

4. Set up an initial state of a dummy list of messages in the constructor that you built in the previous step:

```
// State will be messages: ["Hello World", "How are you"]
this.state = { messages: [] };
```

5. Build a **`renderProfile()`** function:

```
renderProfile() {
  if (this.state.messages && this.state.messages.length > 0) {
    return (
<ul>
  {this.state.messages.map((msg, index) => <li key={`msg-
${index}`}>{msg}</li>)}
</ul>
  );
```

```
    } else {
   return (<div>No messages!</div>);
   }
}
```

6. Next, we will fill in our **`componentDidUpdate`** lifecycle method: **`componentDidUpdate(prevProps, prevState)`**. Remember, we are not implementing **`getSnapshotBeforeUpdate`** so we do not need to include the snapshot argument here. We will add two **`console.log`** statements, one for each of the passed-in arguments:

```
  componentDidUpdate(prevProps, prevState) {
  console.log('prevProps:', prevProps);
  console.log('prevState:', prevState);
  }
```

7. In **`componentDidMount`**, simulate a long-loading profile:

```
  componentDidMount() {
   setTimeout(() => this.setState({ messages: ["Hello World", "How
are
   you?"] }),
   10000 // 10 seconds
);
  }
```

When the page reloads, you should see **`No messages`** until the timeout completes, which is a bad user experience and makes the user distrust the information that this component displays to them. Let's update this to instead follow best practices and display a "**`loading`**" state instead.

8. Go back to the state and add a new flag for a **`loading`** state:

```
this.state = { messages: [], loading: true };
```

9. In **`renderProfile`**, change the **`render`** to also include a check for whether we are loading:

```
renderProfile() {
  if (this.state.loading) {
   return (<div>Loading...</div>);
 }
  if (this.state.messages && this.state.messages.length > 0) {
   return (
    <div>
```

```
        <ul>
          {this.state.messages.map((msg, index) => <li key={`msg-
          ${index}`}>{msg}</li>)}
          </ul>
        </div>
      );
      } else {
      return (<div>No messages!</div>);
      }
    }
```

10. Now add a **`loading`** state update to the **`componentDidMount`** function to update our new **`loading`** flag:

```
    componentDidMount() {
      setTimeout(() => this.setState({ messages: ["Hello World", "How
are
     you?"], loading: false }),
      10000 // 10 seconds
    );
  }
```

11. Finally, go back to the **`render()`** method and add a call to **`renderProfile`**:

```
  render() {
    return (
      <div className="App">
        User Profile
      <hr />
        {this.renderProfile()}
      </div>
    );
  }
```

Now when the page loads, we should see the **`loading message`** until the browser returns data. If you were to change the function inside of the **`setTimeout`** to instead return an empty list (if the user had no messages), it would instead display the **`no messages`** text to the user after loading, which means the user can trust the state that the component is returning to the user!

The final UI should look like this:

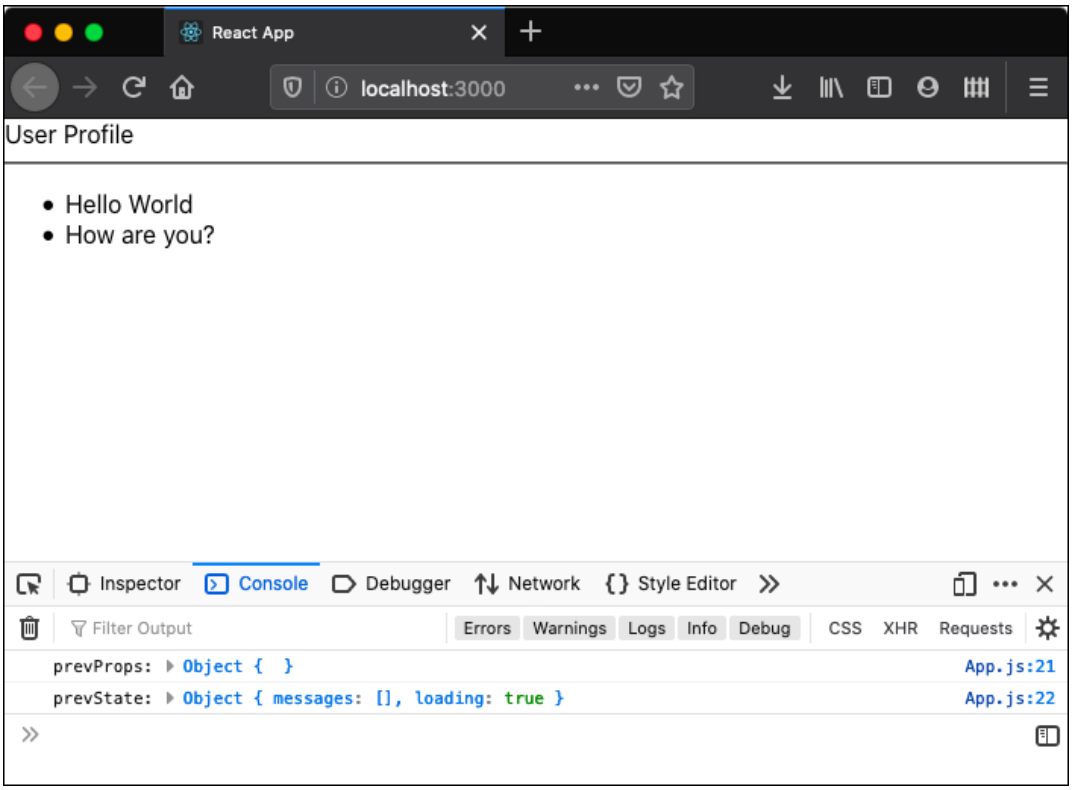

Figure 4.9: App component

With that, we have built a good simulation of something you, as a React developer, will absolutely have to account for at some point in your career: dealing with a slow-loading component. It's most likely will be due to some AJAX call, but regardless of the cause you are now well equipped to be able to handle it and know how to properly structure your code to prevent scenarios like that from affecting your overall page load performance.

THE UNMOUNT LIFECYCLE

The **`unmount`** lifecycle lives outside of the main lifecycle, as it only comes into play when a component is removed from the DOM. This lifecycle is also far more limited, as the only lifecycle method it contains is **`componentWillUnmount()`**.

COMPONENTWILLUNMOUNT()

`componentWillUnmount()` exists solely to perform and react to the component being removed from the DOM, and then is eventually removed entirely from existence. While it sounds a bit dramatic, it's important to note that this is what happens when a component is otherwise completely destroyed. It takes no arguments, and there are some caveats about using it.

The code for **`componentWillUnmount()`** might look something like this:

```
class App extends Component {
  componentWillUnmount() {
    alert("I've been removed!");
  }
  render() {
    return (<div className="App">Hello World!</div>);
  }
}
```

Now if something were to happen that would cause that component to no longer be displayed, we'd see a browser alert box pop up with the message **`I've been removed!`**.

When can a component be removed from the DOM, though? This can happen if you're building a list of components and one of the items in the list gets deleted in JavaScript and that component's JSX is no longer included in the rest of the JSX that is getting written to the browser. It can also happen if you are using conditional rendering and the condition that previously displayed the component is no longer true.

Since this is called only when a component is about to be removed from the DOM and tossed into the void, you probably shouldn't call any state or prop modification logic on it, nor set any new event handlers or anything that you'd want to do to a long-lived component.

> **NOTE**
>
> The only time this is called is before the component is unmounted, instead of after the operation, unlike **`componentDidMount`** and **`componentDidUpdate`**.

EXERCISE 4.04: MESSAGING ON UNMOUNT

We'll work through a quick exercise to quickly demonstrate the usage of **`componentWillUnmount`**. We'll build up a list of items, display them on the page, and then demonstrate what happens when each item is removed from the list, and thus the JSX is removed from the DOM.

1. Start off by creating a new React project, which we will call **`will-unmount`**, start the project, and take a look at the browser window that should have opened for you automatically:

```
$ npx create-react-app will-unmount
$ cd will-unmount
$ yarn start
```

2. Next, strip out the stuff we don't need. Delete **`src/logo.svg`** and delete the contents of **`src/App.css`**. Clean out the contents of the **`App`** component and replace it with a class component instead. Since we are using a class component here, we will need to change the **`import`** statement at the top to also **`import Component from React`**:

```
import React, { Component } from 'react';
import "./App.css";
    class App extends Component {
      render() {
        return (
         <div className="App">
      <h1>My List</h1>
    </div>
   );
  }
}
export default App;
```

   Visit the app in your browser by going to **`http://localhost:3000`**.

3. Next, set up the state in the constructor of the **`App`** component to be a list of items. Each item should have an **`ID`** and a **`message`** that we'll display to the user:

```
constructor(props) {
  super(props);
  this.state = {
    list: [
```

```
      { id: 1, message: 'Hello' },
      { id: 2, message: 'Everyone' },
      { id: 3, message: 'What' },
      { id: 4, message: 'Is' },
      { id: 5, message: 'Up' }
    ]
    };
  }
```

4. We will also need the component that our **`componentWillUnmount`** function will live in, so we will create a new component to display each message. Create this as **`src/Message.js`**. The message will be passed in via **`props`**, and we will also need a **`remove me`** button that we will flesh out later:

```
import React, { Component } from "react";

class Message extends Component {
  render() {
  return (
    <div className="Message">
      <p>{this.props.message}</p>
      <button>Remove me</button>
    </div>
  );
  }
}
export default Message;
```

5. Next, return to our **`App`** component and import the **`Message`** component:

```
import Message from "./Message";
```

6. In the **`render`** function, we'll display each of the components via a **`map`** statement:

```
render() {
  return (
  <div className="App">
    <h1>My Items</h1>
    {this.state.list.map(item => (
      <Message
      key={item.id}
      id={item.id}
```

```
        message={item.message}
      />
      ))}
    </div>
    );
    }
```

7. Now add a function that will remove the item from the list in state. It will need to accept an ID as its only argument and filter the list of items on the **`filter`** statement:

```
removeItem(id) {
const newList = this.state.list.filter(item => item.id !== id);
this.setState({ list: newList });
}
```

8. Pass that **`removeMe`** function down to the **`Message`** component (in the **`render()`** function body) as one of its props:

```
{this.state.list.map(item => (
  <Message
      key={item.id}
      id={item.id}
      message={item.message}
      removeItem={this.removeItem.bind(this)}
    />
  ))}
```

9. Add a **`componentWillUnmount()`** function to our **`Message`** component that will just produce a **`console.log()`** statement indicating what component has been removed:

```
componentWillUnmount() {
console.log('Removing item', this.props);
}
```

10. And finally, add the call to **`this.props.removeMe`** to the **`onClick`** handler for the button we added earlier:

```
<button onClick={() => this.props.removeItem(this.props.id)}>
  Remove me
</button>
```

11. Finally, confirm that everything is working, and messages are flowing into the console each time you remove a component, like in the following screenshot. The output is as follows:

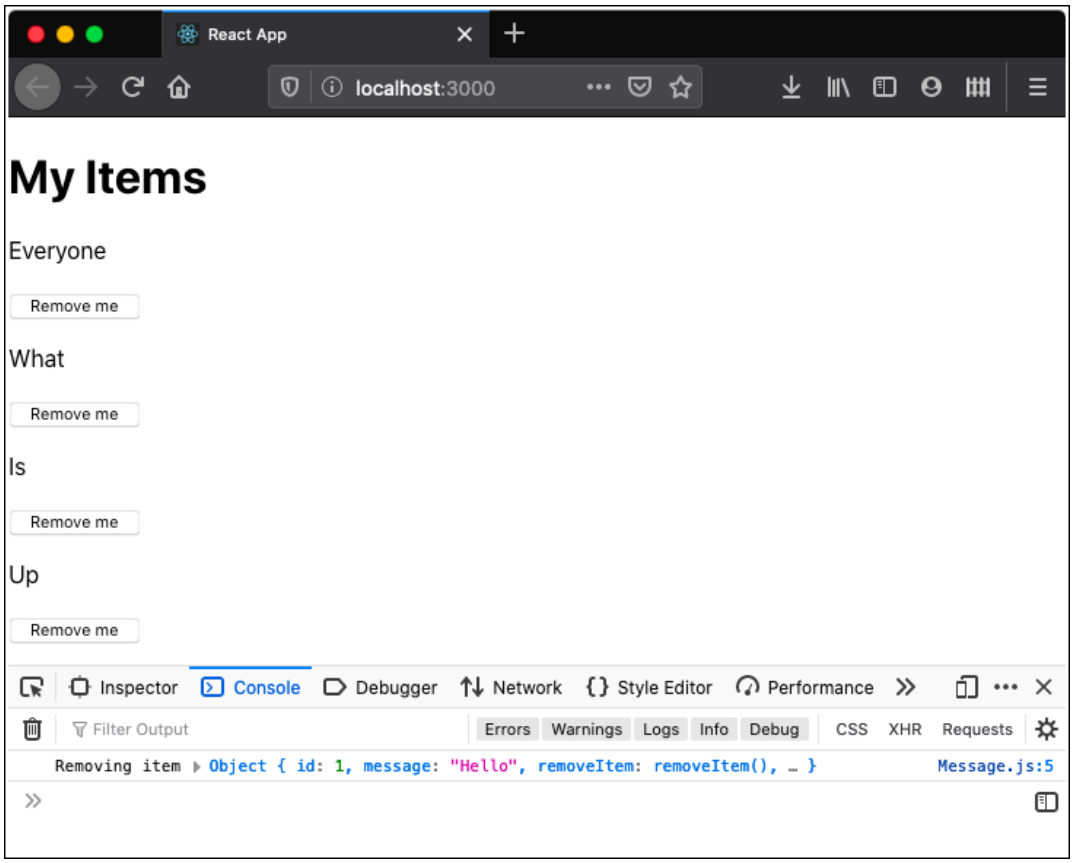

Figure 4.10: React app component

Now, it's time to put into practice all the lifecycle method skills you have gained in the chapter with the upcoming activity.

ACTIVITY 4.01: CREATING A MESSAGING APP

In this activity, you're going to write an app that will simulate logging into a messaging application, loading your messages in the app based on what username you enter, and logging out, all of which should simulate a real app with AJAX calls but without the actual logic. This will help you get a feel for the sort of decisions you would need to make around the lifecycle in a real-world messaging application, where the user must be able to trust the behavior of the application. React's lifecycle methods give us a good way to provide that sort of seamless feel.

The following steps will help you achieve the activity:

1. Create your React application.
2. Simulate an Ajax request to export a few different utilities; a library has been provided for this **`AjaxLibrary.js`**. It can be found inside the **`src`** folder: https://packt.live/2N3BEoA.
3. Create the base file **`UserCountDisplay.js`** where you'll need to use the **`constructor()`**, **`componentDidMount()`**, and **`render()`** lifecycle methods; you will have to meet a few requirements such as: the initial user count should be zero; a message displaying the user count is loading; after the message is loaded, it should show the number of users.
4. Update the **`render`** function in the **`UserCountdisplay.js`** file.
5. Create a basic form inside called **`LoginDisplay.js`**; import the **`AjaxLibrary.js`** here as well.
6. Add an **`onChange`** event handler to the username in the form and create an **`eventHandler`** for the **`fetchUser`** function from the **`Ajaxlibrary.js`** file.
7. Create the **`Logout`** button with the required functionality.
8. Add the **`LoginDisplay`** component to your **`App`** component.
9. Pass the user prop into the **`LoginDisplay`** component by importing the **`UserDisplay`** component in the **`LoginDisplay`**.
10. Update the **`async`** function in order to display the loading message.

The final output should look like this:

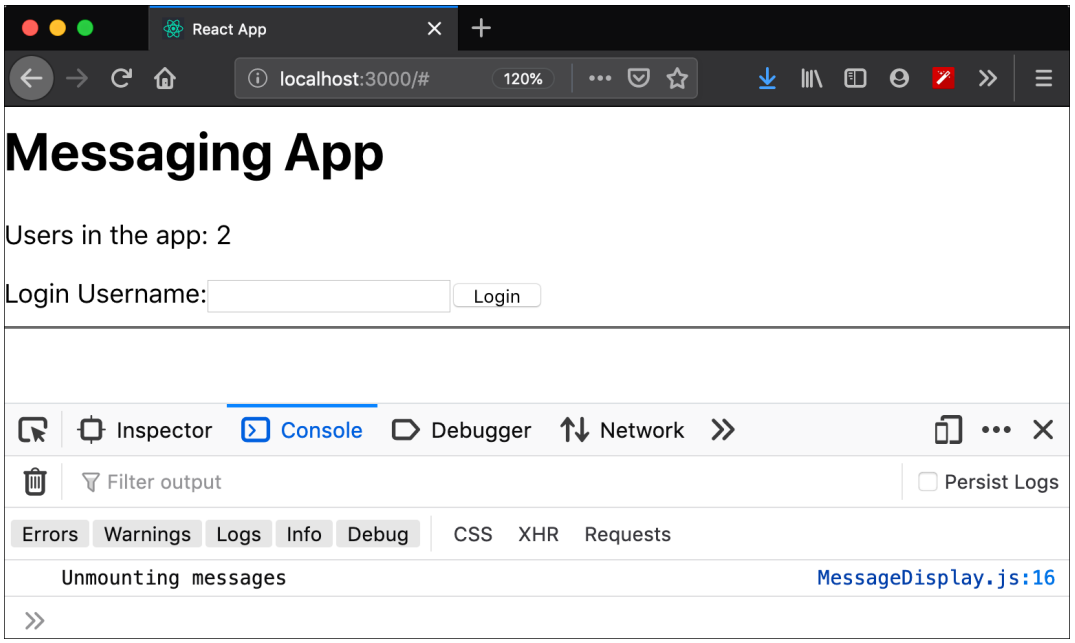

Figure 4.11: Final output

> **NOTE**
>
> The solution of this activity can be found on page 623

With this activity, you have successfully created the Messaging App.

SUMMARY

In this chapter, we have described and implemented various React component lifecycle methods. Knowing how the lifecycle perpetuates itself is critical to architecting React components that are anything larger and more complex than a couple of static boilerplate components. Moreover, it is worth understanding how the lifecycle flows adding additional functionality of events as appropriate. For example, knowing about the lifecycle allows us to determine the best place to add in functionality such as making external AJAX calls that should modify the state without doing something that would affect the rendering pipeline for the component. This is something that is borderline required for mastery of writing smart React components.

If you remember the basic flow of implementing lifecycle methods, as mentioned in the chapter, you will have enough information to get you through most of the React components that you write in your lifetime. In the next chapter, we will look at class and functional components in React.

5

CLASS AND FUNCTION COMPONENTS

OVERVIEW

This chapter will introduce you to the two components types in React, class components and function components. With the knowledge acquired from this chapter, you will be able to use industry best practices to identify the component hierarchy and break the UI down into logical components. This chapter aims to form the basis of creating UIs in React, be they simple or complex, and provides you with the basic tools required to build React applications.

INTRODUCTION

In React, code is written in the form of entities called components that help us organize code modularly. This modular structure allows apps to be scalable and maintainable. React.js, as a JavaScript library, has achieved a lot of popularity in the web domain and has rightly been enjoying this success because it allows developers to build scalable apps rapidly while employing industry best practices.

In the previous chapters, we have largely worked with a specific type of component in React called class components. When the React framework emerged in the scene, the use of class-based components increased. During the early days of React, even before classes were available to us in JavaScript (ES6), React implemented its own take on classes with **`React.createClass`**. However, starting with *React 0.14.0*, ES6 component classes extended **`React.Component`**. And so, components can now be defined as **Class components** that use JavaScript classes to extend **`React.Component`** and **Function components** that are written as JavaScript functions. React works with React DOM, which renders these components in a chosen root element in the HTML DOM to render the React app. Let's start with the structure of React components.

INTRODUCTION TO THINKING WITH REACT COMPONENTS

React provides an excellent way to build scalable and highly performant applications. We have to carefully and efficiently structure these applications while we are building them. We will look at a few design principles that will help us to logically structure our app into components.

ATOMIC DESIGN

While building web and mobile applications, which is a common use case for React, we tend to create a design system that consists of elements, along with the design and interaction that make up the application. Now, the more reusable the components are, the more the design becomes consistent and cohesive, and speed improves. Developers are able to work with components, which provides a good foundation to grow and make further modifications when required.

One way to compose a design system would be to use the principles of atomic design, which dictate how components should look and feel when the user interacts with them. Atomic design, as coined by Brad Frost, takes its cues from chemistry and uses it as a metaphor to explain the structure of UIs:

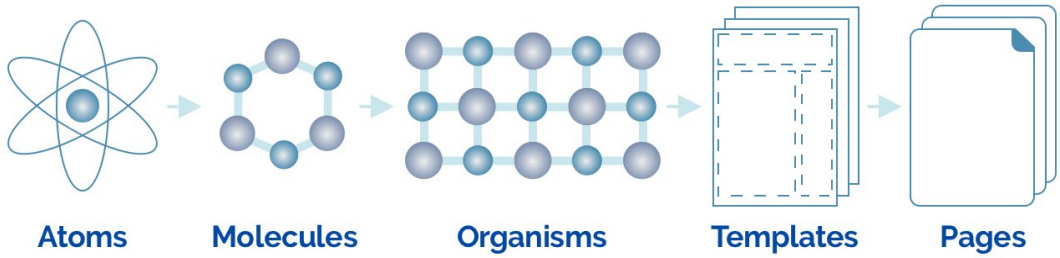

Figure 5.1: Atomic design, as coined by Brad Frost

According to modular atomic design, web components should be versatile and reusable. We begin with atoms, which are the smallest basic building blocks. In UIs, these are elements such as buttons, inputs, form controls, and labels.

Molecules are a combination of multiple atoms that, in UIs, would be grouped elements that work together, such as a search module that is a combination of a form label and a text input with a button or a simple styled list of links.

Combining various molecules gives us organisms, which are relatively complex UI components, such as a website header or navigation bar, which is made up of smaller elements, such as lists of links, dropdowns, and hamburgers for mobile navigation.

Templates that are made up of organisms begin to take shape, showing us the final design layout. These are page-level objects and articulate the desired content structure. An example would be to create a dashboard template or a home page template using the elements mentioned above.

Pages are created from templates and are higher fidelity, the final level of our interfaces. They have all the realistic content, assets, and media added and are a true reflection of what the end result will be.

With this knowledge of how an application should be designed and can be broken down into several UI elements, let's take a look at the wireframe of a shopping cart.

PRAGMATIC APPROACH

Here is a wireframe of a shopping cart:

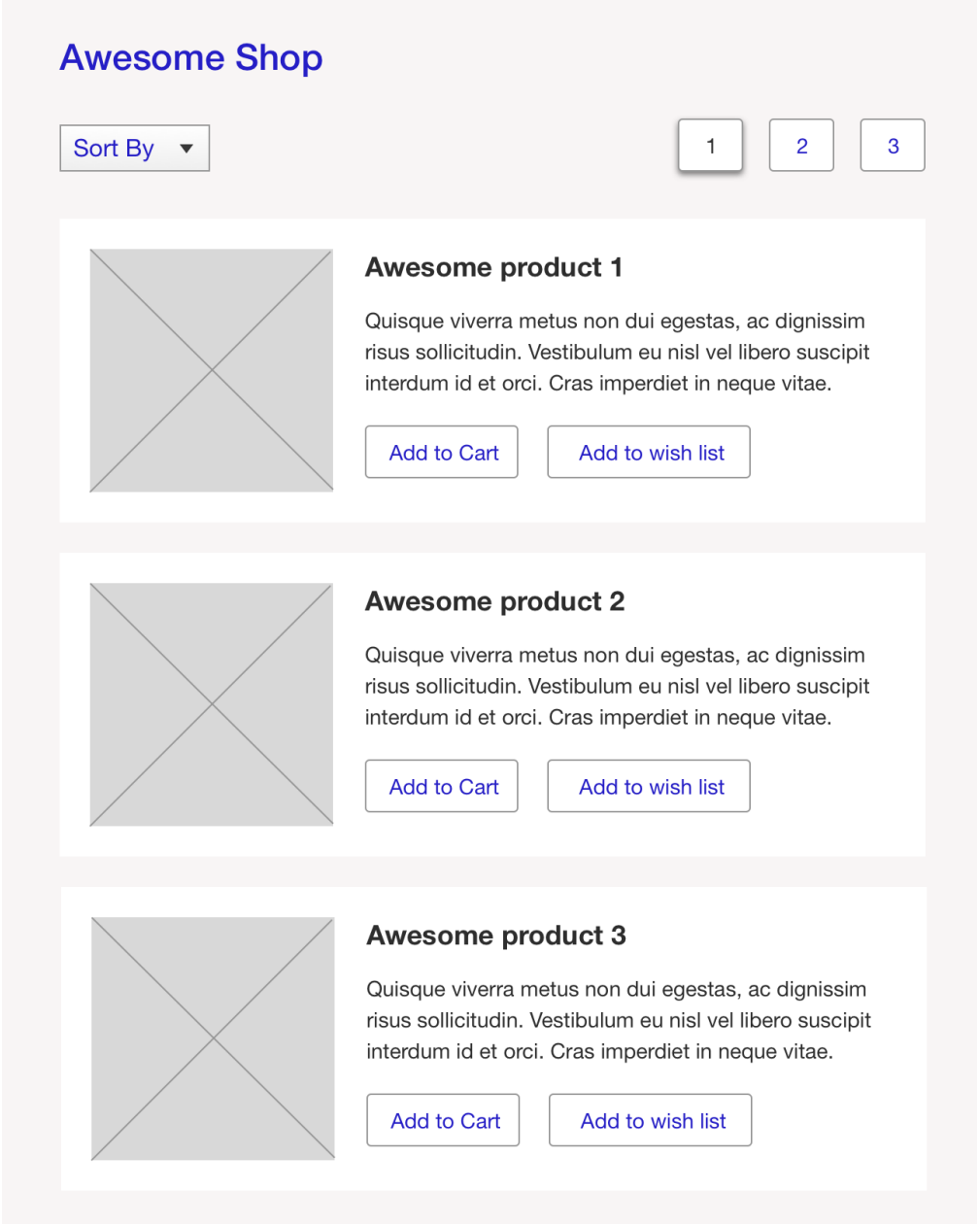

Figure 5.2: Wireframe of a shopping cart

We can see the following elements on screen:

- Page header
- Sort field
- Pagination
- Product list

The product list comprises product elements, which includes the following:

- Product title
- Product image
- Product description
- Add to Cart
- Add to wish list

We shall next break down the UI into the elements we have identified and mark them so that we are able to build reusable components. Atomic design principles help us to think about how the components should be broken down and structured. We also need to name them, which may be a common name for the UI element.

The following is the wireframe of the same shop with marked components:

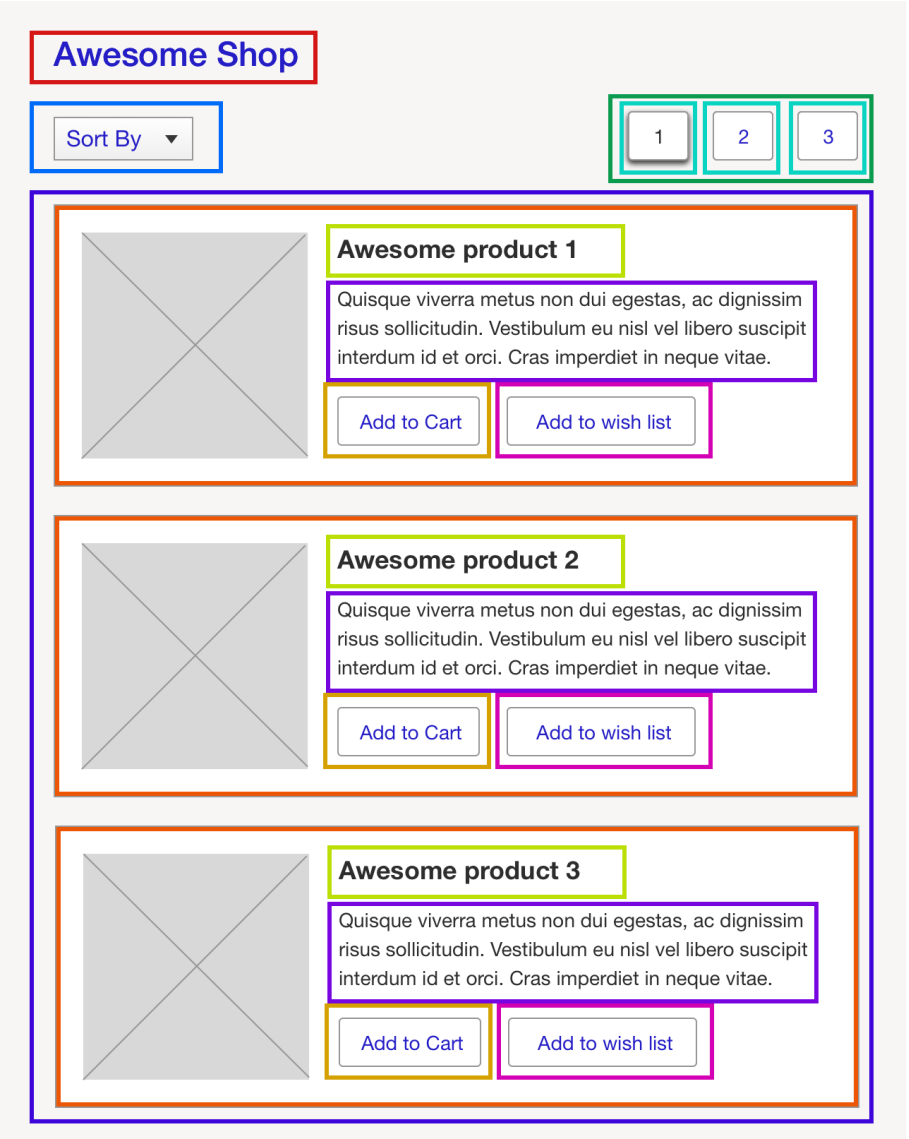

Figure 5.3: Elements marked as components

> **NOTE**
>
> Components are written in pascal case, which is a naming convention where the first letter of every word in a compound word is capitalized.

The component names would be:

- PageHeader
- SortField
- Pagination
- Button
- ProductList
- Product
- ProductImage

Now, we are able to correctly identify the elements that could be transformed into a component. In the next section, let's focus on building React components.

BUILDING REACT COMPONENTS

Since presentation and data can be abstracted, breaking the UI down into components reduces clutter and gives more meaning to the components, which can now perform specialized tasks. Writing components as container and presentational components, also referred to as smart and dumb components, is a pattern that is commonly employed by developers in React.

Container components usually have a local state and are concerned with the implementation of data and how things work. They have minimum to no markup. These components deal with data, be it loading it, calling actions, or mutating them. When creating a shopping cart application with the wireframe shown in *Figure 5.3*, it would suffice to have container components at the root level, which will then retrieve and hold the data and just pass it down to its children. The child components, which are typically the presentation components, receive values to be displayed and show the required JSX on screen as output. Presentational components are usually stateless components that are concerned with the way things essentially look in the UI. They usually have some DOM markup and associated styles and could encompass both presentational and container components. They do not load or mutate data and are usually written as function components.

DATA FLOW

In React, the data typically flows from the root to the child components and is *unidirectional*. When a user interacts with the UI, for example, they click on a sort field to sort by price, an action is triggered that causes sorted data to be requested from the server and updated. A stateful container component updates the data and, as it changes, it reciprocates through all the child components. In *Figure 5.3*, the root-level app component holds data pertaining to products and passes it to the **`ProductList`** component, which renders the product information. When the user performs an action to fetch new data, the data is fetched and updated and then gets passed down to **`ProductList`**, which updates the view.

We have so far seen references to the terms **`state`** and **`props`** and, before going any further, we should demystify what these terms are because they are such an intrinsic part of what we do in components.

STATE AND PROPS

Container components have local mutable variables called **state** that change when a user interacts with the component. Since data flow is unidirectional in React, the change in state is passed down to the presentation components as immutable **props**.

Simply put, **state** holds local variables that can then be edited or mutated by the component, and **props** are immutable variables that the component receives.

The preceding example shows a simple way to use components and we will learn in detail each aspect of the components.

CLASS COMPONENTS

React allows us to define components as class or function components. Class components that are stateful can be created by extending **`React.Component`** and are more feature-rich than function components and provide us with a number of predefined methods that make managing data and components easy. We can also write our own methods that can be used at different stages of the component.

Let's see an example of a simple class component that shows a **`Hello World`** message:

```
import React, { Component } from "react";
class HelloMessage extends Component {
  render() {
    return <h1>Hello World!</h1>;
  }
}
export default HelloMessage;
```

This can be used in a file as:

```
import HelloMessage from "./HelloMessage";
```

If you've noticed, we have used a syntax that may seem unfamiliar to developers who are new to React. React code is usually written in ECMAScript 2015 or later. ECMAScript 2015 is also commonly referred to as ES6, and we will continue to use ES6 for all our code in the chapter.

Let's further try and dissect each line of our code to understand it and what it does:

```
import React from "react";
```

The first line uses the ES6 **`import`** command to **`import React from react`**. This is essential when you have JSX syntax in the code.

Next, we create a React component, **`HelloMessage`**, as a class by extending **`React Component`**:

```
class HelloMessage extends React.Component
```

Though classes have been around in most programming languages for a long time, JavaScript classes were introduced in ECMAScript 2015. **`React.Component`** is what makes the features of a React component available to us:

```
render() {}
```

A **`render`** method is defined in the class as a built-in method in React. In React, it is called a life cycle method, which has a hook that renders the component. JSX content is returned by the **`render`** function:

```
return <h1>Hello World!</h1>;
```

The content returned by the **`render`** function is what gets displayed on the page. The **`HelloMessage`** component uses the default export available in ES6. Without affecting the output in any way, this component can also be written as a named export while also using a named import to import **`React.Component`**. We can also combine a class definition with an export, as shown in the following code:

```
import React, { Component } from "react";
export class HelloMessage extends Component {
  render() {
    return <h1>Hello World!</h1>;
  }
}
```

Multiple named exports can exist in a module, but only one default export can be used. Developers can choose to either use named exports or default exports.

In our examples, moving on, we will use named exports and imports. The preceding component can be imported inside **`index.js`**, which uses **`ReactDom`** to render this in an HTML **`div`** element with the ID **`root`**:

```
import React from "react";
import "./App.css";
import { HelloMessage } from "./HelloMessage";
function App() {
  return (
    <div className="App">
      <HelloMessage />
    </div>
  );
}
export default App;
```

The **`css`** file that could be used along with the preceding component is as follows:

```
.App {
  text-align: center;
  font-family: sans-serif;
}
```

The following is the output of this component on a browser:

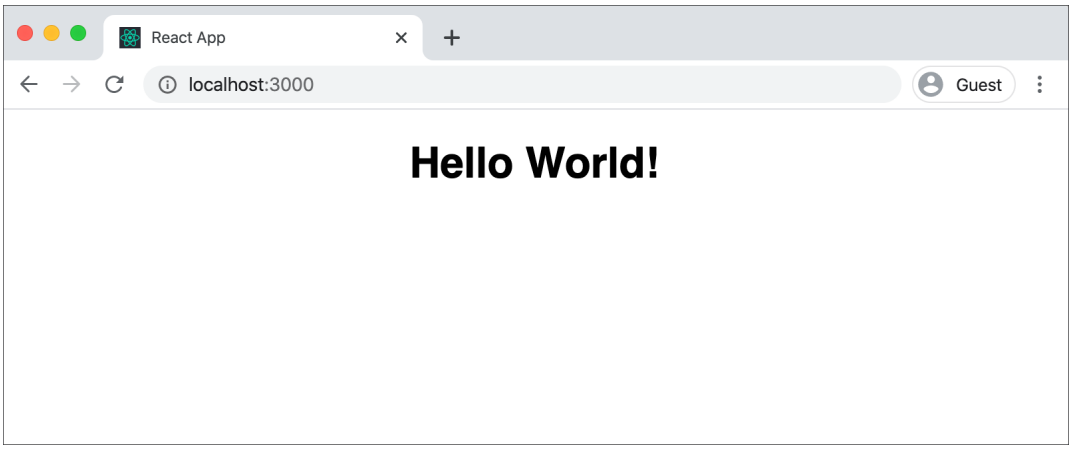

Figure 5.4: Hello World app

Now that we have seen how to create a class component, let's practice this in the following exercise.

EXERCISE 5.01: CREATING PROFILE COMPONENT AS A CLASS COMPONENT

In this exercise, we will create a class component called **`profile class component`** that can receive data pertaining to a person and display it according to design.

1. Open the command line to create our app.
2. Run the **`npx`** command to create the react app. We have chosen the name **`profile-class-component`**:

```
$ npx create-react-app profile-class-component
```

3. Navigate to the folder and run our app in development mode:

```
cd profile-class-component/
yarn start
```

This should start running your app and render the initial react app screen:

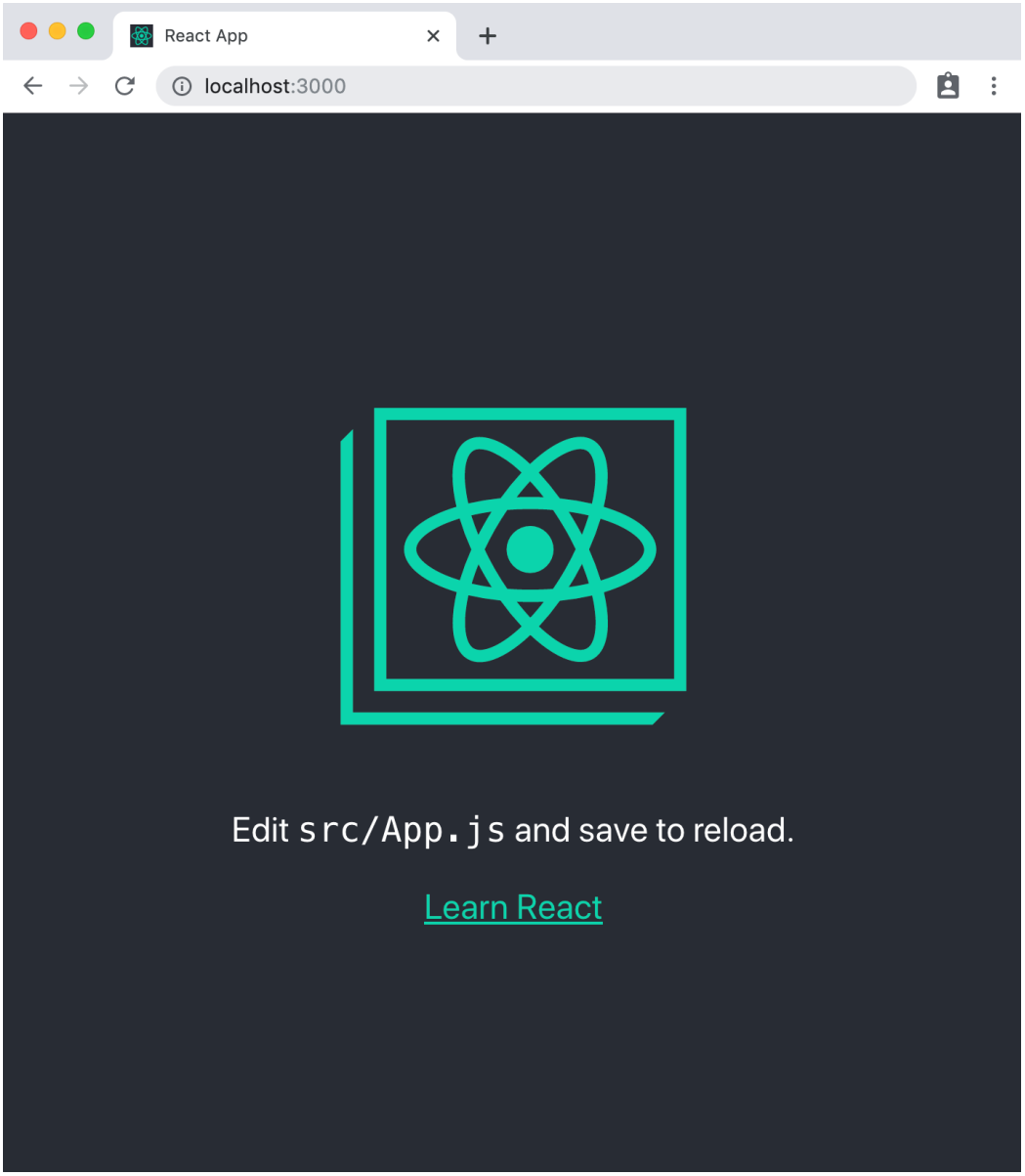

Figure 5.5: Initial screen

4. Now, open the code in our code editor. We will use Visual Studio Code in our example here, but it can be any editor of your choice. We will now edit the code in the **`src/App.js`** file:

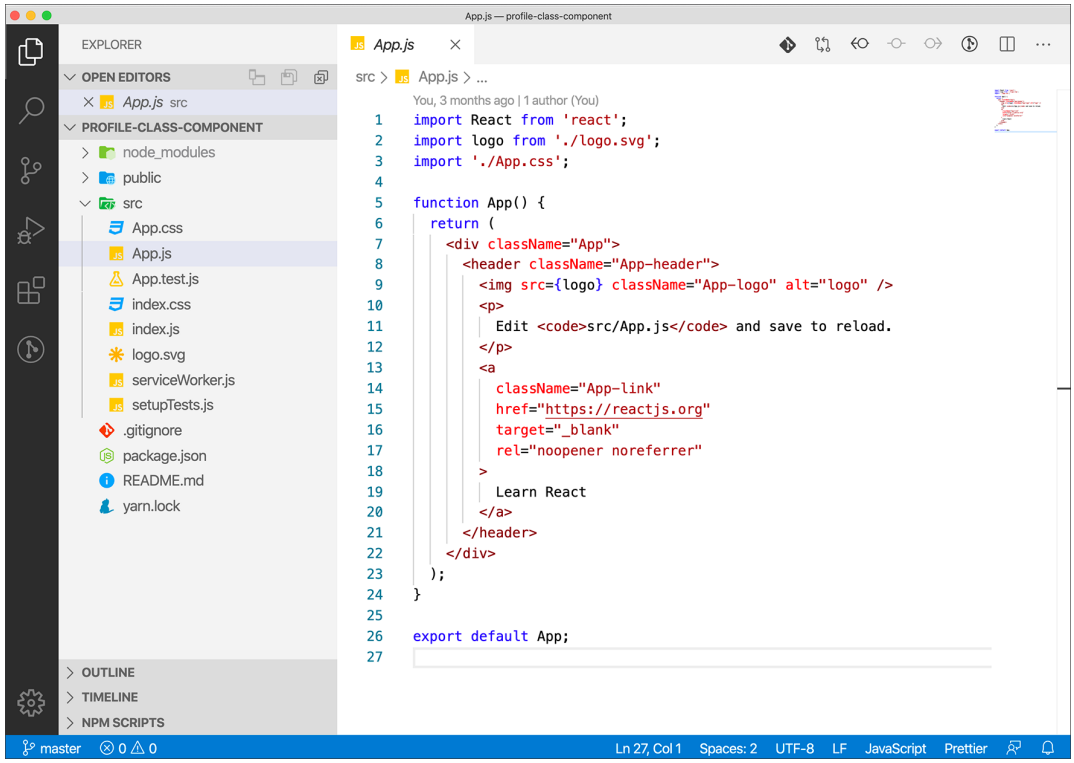

Figure 5.6: Modifying code in the src folder

5. Delete the **`src/logo.svg`** files.

6. Create a file named **`Profile.js`** and add a class component that returns some JSX in the render function:

```
import React, { Component } from "react";
class Profile extends Component {
  render() {
    return (
      <section className="profile">
        Profile
```

```
        </section>
      );
    }
}
export default Profile;
```

7. The **`App.js`** file can now include and render **`Profile.js`**, which then changes to the following:

```
import React from "react";
import "./App.css";
import Profile from "./Profile";

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

export default App;
```

8. Replace the css in **`App.css`** with basic styling:

```
.App {
  text-align: left;
  margin: 50px;
}
```

   Doing this renders the newly created **`Profile`** component on screen, but we do not have any relevant information displayed.

9. In **`App.js`**, declare a user variable with the data in our app. In reality, we would usually get this information from an API.

   For our example, we are using an image from UnSplash, https://unsplash.com/photos/g4PLFkpUf4Q, which provides us with images that are free to use:

```
const user = {
  name: "Brian",
  interests: "Reading, Swimming, Technology",
  age: 9,
  image: "https://images.unsplash.com/photo-1470071131384-
          001b85755536?auto=format&fit=crop&w=200&q=80",
```

```
  color: "Red",
  movie: "Star Wars"
};
```

10. Pass this data to the **`Profile`** component as an attribute:

```
const user = {
  name: "Brian",
  interests: "Reading, Swimming, Technology",
  age: 9,
  image: "https://i.pravatar.cc/200?img=4",
  color: "Red",
  movie: "Star Wars"
};

function App() {
  return (
    <div className="App">
      <Profile user={user} />
    </div>
  );
}

export default App;
```

11. Let's access this data using **`this.props`**. Hence, the **`user`** object is available as **`this.props.user`**:

```
import React, { Component } from "react";
class Profile extends Component {
  render() {
    return (
      <section className="profile">
        <header>
          <h1>{this.props.user.name}</h1>
        </header>
        <div className="profile-content">
          <div className="profile-image">
            <img src={this.props.user.image} alt={this.props.user.
name} />
          </div>
          <div>
            <p>
```

```
                <strong>Age:</strong> {this.props.user.age}
              </p>
            </div>
          </div>
        </section>
      );
    }
  }
  export default Profile;
```

When you start and run the application, it will begin to show the information on screen:

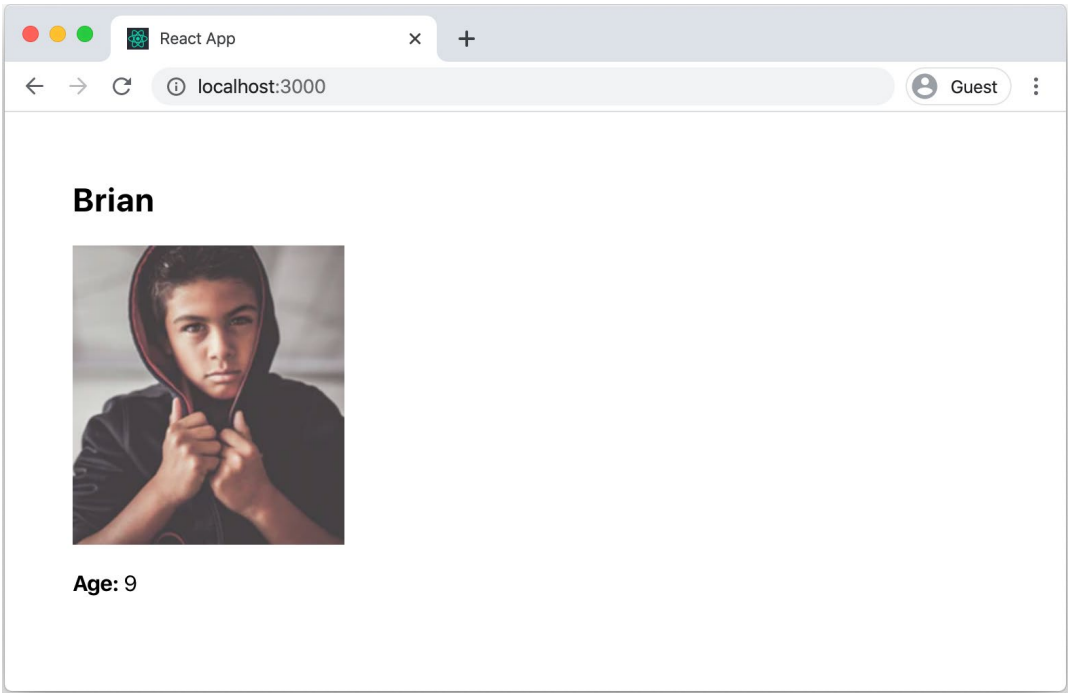

Figure 5.7: Initial user profile displayed

However, accessing all properties using **`this.props.user`** seems repetitive, and we can use the ES6 spread syntax instead.

12. Use the ES6 spread syntax and add this in **`Profile.js`** inside the **`render`** function:

```
const { name, image, age, interests, color, movie } = this.props.
user;
```

> **NOTE**
>
> In ES6, the de-structuring assignment allows us to unpack arrays and objects. If you would like to know more about this, you can visit MDN for more information: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

This allows us to use these variables in our JSX:

Profile.js

```
5  class Profile extends Component {
6    render() {
7     const { name, image, age, interests, color, movie }
      = this.props.user;
8    return (
9    <section className="profile">
10     <header>
11        <h1>{name}</h1>
12        </header>
18            <p>
19              <strong>Age:</strong> {age}
20            </p>
21            <p>
21              <strong>Interests:</strong> {interests}
```

The complete code can be found here: https://packt.live/2T3UqPH

13. Add the styling for our **Profile** component and then import it. We will add some styling in a new file, **profile.css**:

```
.profile {
  max-width: 500px;
  margin: 10%;
  padding: 20px;
  border: solid 1px #eee;
  box-shadow: 0 0 5px rgba(0, 0, 0, 0.3);
  border-radius: 5px;
}
.profile-image {
  margin-right: 20px;
}
.profile-content {
  display: flex;
}
```

14. To import this in our **`Profile`** component, add the **`import`** statement to **`Profile js`**:

`Profile.js`

```
1  import React, { Component } from "react";
2  // styles
3  import "./profile.css";
4
5  class Profile extends Component {
6  render() {
7    const { name, image, age, interests, color, movie }
     = this.props.user;
8    return (
9      <section className="profile">
```

`The complete code can be found here:` https://packt.live/2T3UqPH

With this implemented, our **`Profile`** component is now complete and shows the profile:

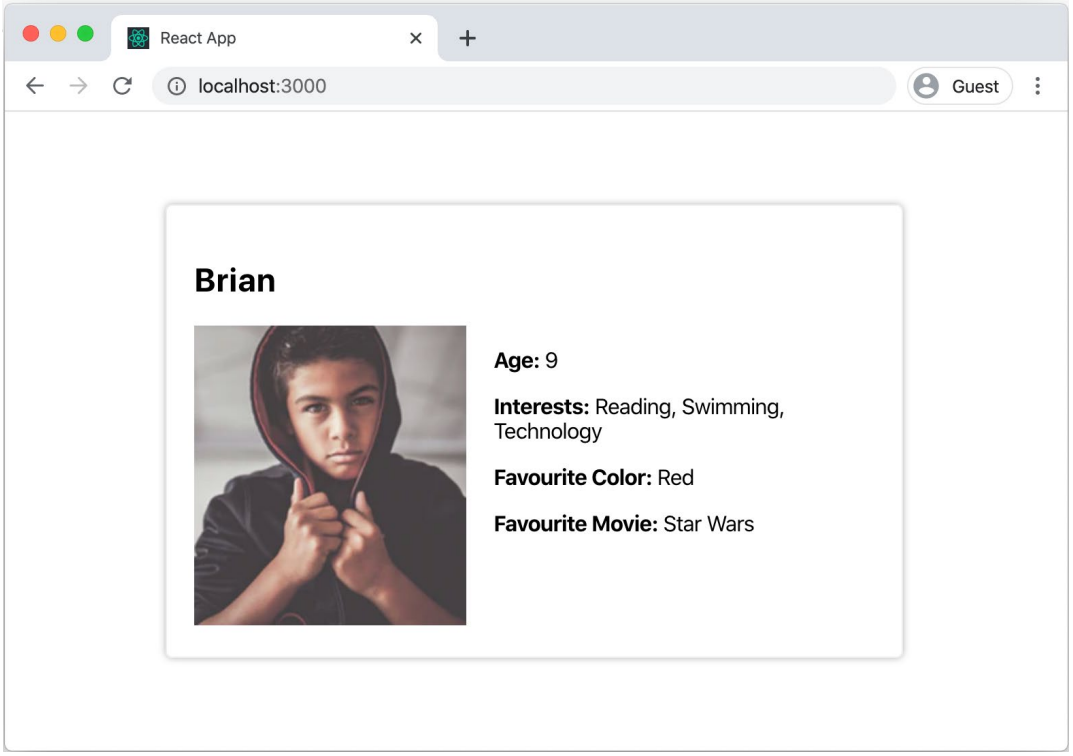

Figure 5.8: User profile class component

With this exercise, we have finished designing a full-blown class component.

FUNCTION COMPONENT

Simply put, a function component in React is a JavaScript function that returns JSX.

A simple **`HelloWorld`** function component is shown below:

```
function HelloWorld() {
  return <div>Hello, World!</div>;
}
```

The function can receive variable objects as props from the parent component that can be used for evaluation and rendering.

A **`Hello`** function with props is shown below:

```
function Hello(props) {
  return <div>Hello, {props.name}!</div>;
}
```

The preceding syntax can be simplified further with ES6.

Here's the function component written in ES6:

```
const Hello = (props) => (<div>Hello, {props.name}!</div>);
```

Let's examine the code we have written here to understand how this works.

```
const Hello
```

We are defining a new constant here and initializing it with a value. We have seen the const statement earlier as well. ES6 introduced block-scoped variables that can be defined with **`let`** and **`const`**. The value of the variable defined using **`const`** is constant and cannot be redefined; however, a **`let`** variable can be changed.

In the following example, the value of the **`greeting`** variable does not change, and so is defined as **`const`**, while the **`addressee`** changes value and so is defined using **`let`**. If this function is within another class or function, these values are not available outside this function since they are block-scoped:

```
function foo(name) {
  const greeting = "Hello";
  let addressee = "World";
  if (name) {
```

```
    addressee = name;
  }
  return `${greeting} ${addressee}`;
}
(props)=> (<div>Hello, {props.name}!</div>)
```

This is ES6 arrow notation for declaring a function that receives props as input:

You may have noticed that we have not returned the JSX using a **`return`** statement. In ES6, we may use concise body or block body. Block body is the traditional way we use an explicit **`return`** statement.

In the preceding example, however, we have chosen to use an implicit return in a concise body where only an expression needs to be specified. Here, the JSX is wrapped in braces, which signifies that this is what gets returned.

Now that we have seen how we can create a functional component, let's practice this with the following exercise.

EXERCISE 5.02: CHANGING THE LOADER COMPONENT AS A FUNCTION COMPONENT

We will now create a simple **`Loader`** component that will show a spinner. This is usually used when loading data from an API using AJAX. We will pass a prop to it, which will then be used to decide whether we need to show a spinner. To do so, let's go through the following steps:

1. Start again by opening the command-line tool and running the following command:

```
$ npx create-react-app loader-function-component
```

2. After which we navigate to the folder and run our app in development mode:

```
cd loader-function-component/
$ yarn start
```

This should start our app and render it in our browser.

3. Create a file named **`Loader.js`** and add a component with the syntax of a function component:

```
import React from "react";
const Loader = () => {
  return <div className="spinner"></div>;
};
export default Loader;
```

4. We will update **`App.js`** to include this new component and use it.

```
import React from "react";
import "./App.css";
import Loader from "./Loader";

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

export default App;
```

5. Remove unnecessary styling for our app to have some simple CSS:

```
.App {
  text-align: center;
}
```

Since we have no content displayed, we do not see any output on screen.

6. Return a circle in an SVG that can be styled to become our loader. Adding to our **`Loader.js`** file, our component now changes to:

`Loader.js`

```
12  <div className="spinner">
13      <svg
14        focusable="false"
15        width={spinnerSize}
16        height={spinnerSize}
17        viewBox={` ${spinnerSize} ${spinnerSize}`}
18      >
19        <circle
20          cx={spinnerSize / 2}
21          cy={spinnerSize / 2}
22          r={spinnerSize / 2 - 10}
23        />
24      </svg>
25    </div>
```

The complete code can be found here: https://packt.live/35Wq8DH

We are using a variable, **`spinnerSize`**, that is used to set the width and radius of a circle in the SVG. We have set it to an arbitrary value of **`100`**.

We are now able to see a filled circle:

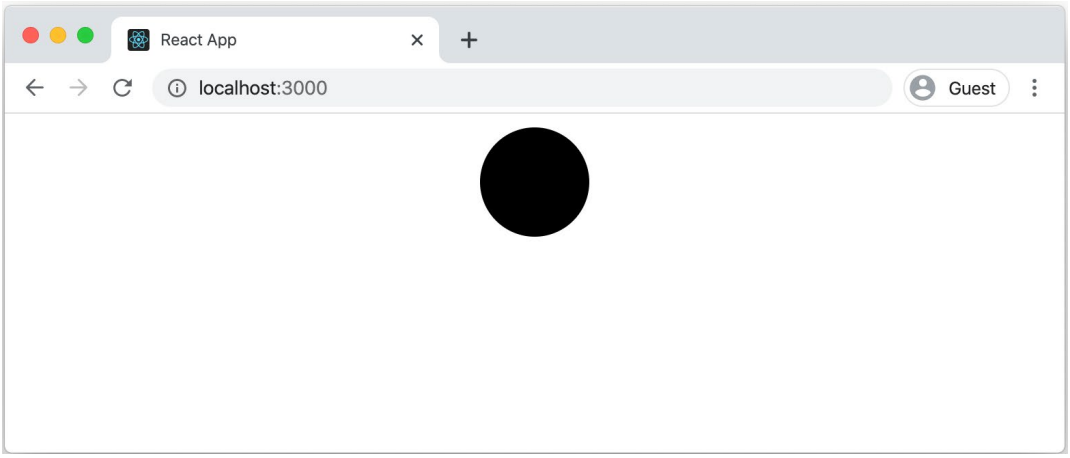

Figure 5. 9: Filled circle in App

7. Add some styling to the SVG to improve its appearance. We will first change the fill and stroke in our **`loader.css`** file and also include the file in our component:

```
.spinner svg {
  stroke: rgba(0, 0, 0, 0.4);
  stroke-width: 7px;
  fill: none;
}
```

8. Import the **`loader.css`** file in **`Loader.js`** file:

```
import React from "react";

// styles
import "./loader.css";
…
```

The SVG now looks better. The output is as follows:

Figure 5.10: Loader component

9. Add styling to our SVG. We will use SVG's stroke properties to make it look like a crescent shape with a rounded line end:

```
.spinner svg {
  stroke: rgba(0, 0, 0, 0.4);
  stroke-width: 7px;
  fill: none;
  stroke-dasharray: 100%;
  stroke-linecap: round;
}
```

The output is as follows:

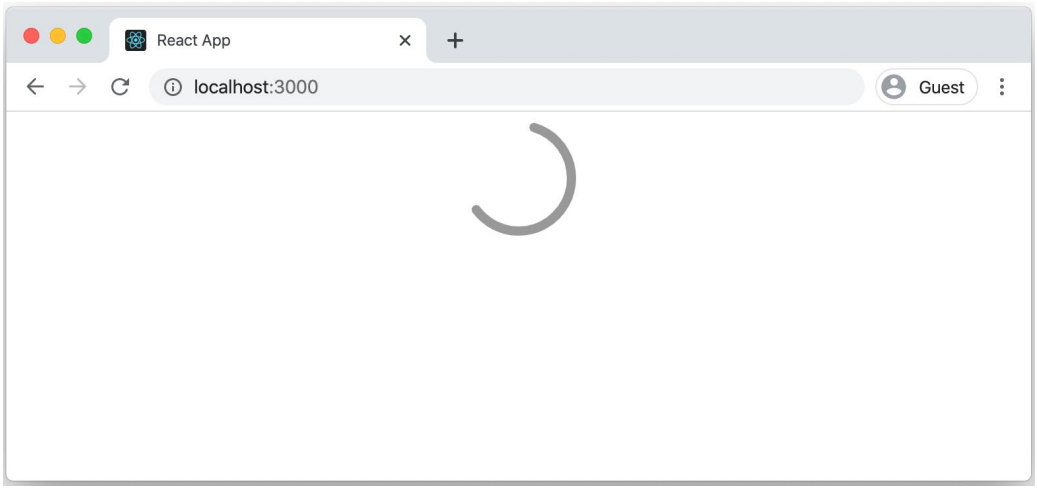

Figure 5.11: The display showing the component is loading

10. Add a CSS animation to the SVG so that it rotates in a loop in **`loader.css`** file:

```
@keyframes spinnerAnimation {
  to {
    -webkit-transform: rotate(360deg);
    -ms-transform: rotate(360deg);
    transform: rotate(360deg);
  }
}
.spinner svg {
  stroke: rgba(0, 0, 0, 0.4);
  stroke-width: 7px;
  fill: none;
  stroke-dasharray: 100%;
  stroke-linecap: round;
  animation: 0.86s cubic-bezier(0.4, 0.15, 0.6, 0.85) infinite
spinnerAnimation;
}
```

11. Add some styles to center the spinner and add some space around it:

`loader.css`

```
1 .spinner {
2  margin: 20px;
3  text-align: center;
4 }
5
6 @keyframes spinnerAnimation {
7  to {
8    -webkit-transform: rotate(360deg);
9    -ms-transform: rotate(360deg);
10    transform: rotate(360deg);
11  }
```

The complete code can be found here: https://packt.live/3fOfN12

With this, the loader starts animating, goes into an infinite loop, and is styled to appear in the center horizontally:

Figure 5.12: The animation of the loader component

Though it looks the part, there are a couple of improvements we can make so that it serves a purpose while building our applications. We will now pass a new prop, **`isVisible`**, and also set the size using a prop spinner, **`size`**.

12. Utilize the **`Loader`** component in **`App.js`** appears as follows:

```
function App() {
  return (
    <div className="App">
      <Loader spinnerSize={60} isVisible={true} />,
    </div>
  );
}
```

This should start running your app and render the initial react app screen.

13. Extract and use the values provided to the component de-structuring assignment from ES6:

Loader.js

```
13    <svg
14      focusable="false"
15      width={spinnerSize}
16        height={spinnerSize}
17        viewBox={`0 0 ${spinnerSize} ${spinnerSize}`}
18      >
19        <circle
20          cx={spinnerSize / 2}
21          cy={spinnerSize / 2}
22          r={spinnerSize / 2 - 10}
23        />
```

The complete code can be found here: https://packt.live/35Wq8DH

Now that we have the value for **isVisible**, we need to render nothing when it is set to **false**. This we can do using an early return.

In React, if you return **null** from a component, it will not render the component and we use this here.

14. Check whether the value for **isVisible** is set to **false**, and if so, we return **null**:

Loader.js

```
6  const Loader = props => {
7  const { spinnerSize, isVisible } = props;
8  if (!isVisible) {
9    return null;
10  }
11  return (
12
13      <svg
14        focusable="false"
15        width={spinnerSize}
16        height={spinnerSize}
17        viewBox={`0 0 ${spinnerSize} ${spinnerSize}`}
18      >
```

The complete code can be found here: https://packt.live/35Wq8DH

With this, our component is complete. When we change the size to, say **`60`**, you can see the loader changing size. And when **`isVisible`** is set to **`false`**, it doesn't render the component on screen:

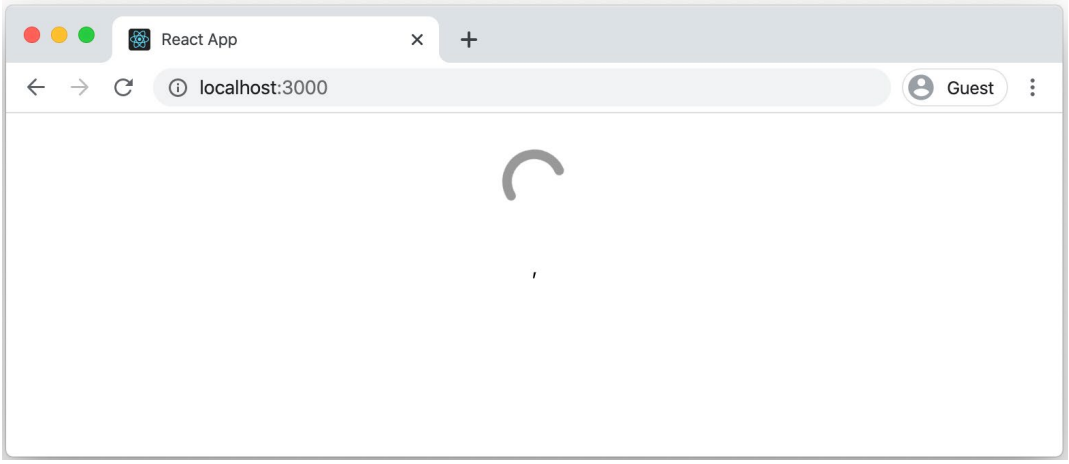

Figure 5.13: The changing size of the loader is displayed

We now have a good understanding of how to create class-based and functional components. In the next section, let's look at a few key pointers to understand the basic differences between the two.

DIFFERENCES BETWEEN CLASS AND FUNCTION COMPONENTS

So far, we have seen how we can define a class component by extending **`React.Component`**, and a function component that can use the power of ES6 to define a concise body with an implicit **`return`** statement to create a simple component.

Apart from sheer size, though, there are also other differences between the two that we need to consider while deciding to create a component using either of the following methods. Let's discuss the pointers one by one.

SYNTAX

Let's consider a simple component that receives a **`name`** as a value from a parent component and shows a **`hello`** message.

The following code is from the **`Hello`** component that is defined as a class component:

```
import React, { Component } from "react";
export class Hello extends Component {
  render() {
    return <h1>Hello {this.props.name}!</h1>;
  }
}
```

The same component can be written as a functional component in a simple way.

The following code is from the **`Hello`** component that is defined as a function component:

```
import React from "react";
const Hello = (props) => (<h1>Hello {props.name}!</h1>);
```

We can see how the syntax differs from the preceding code examples. A class component is defined as a class and uses the **`extends`** keyword to extend the component. A function component is a JavaScript function and is defined using the arrow syntax.

Now, when you transpile the code to ES5, we see the difference in how it is implemented.

> **NOTE**
>
> Transpiling means compiling the source code in one language to another and, in our case, since we write our code in ES6 and browsers may still not understand this code, we transpile to ES5, which browsers can understand and run.

Here is the transpiled class component – **`Hello`**:

```
var Hello =
/*#__PURE__*/
function (_Component) {
  _inherits(Hello, _Component);
```

```
  function Hello() {
    _classCallCheck(this, Hello);

    return _possibleConstructorReturn(this, _getPrototypeOf(Hello).
apply(this, arguments));
  }

  _createClass(Hello, [{
    key: "render",
    value: function render() {
      return React.createElement("h1", null, "Hello ", this.props.name,
"!");
    }
  }]);

  return Hello;
}(Component);
```

Whereas the function component transpiles into a function that is still concise:

The transpiled Function Component – **`Hello`** is as follows:

```
var Hello = function Hello(props) {
  return React.createElement("h1", null, "Hello ", props.name, "!");
};
```

With this knowledge, when we create our components, we should try to use the function component where possible because not only is it simple to code, but it is also far more efficient.

HANDLING STATE

In React, state is the local object that holds values that change within the component. In a class component, however, the state is available, and any changes made to it will cause a re-render. To do that, we have the life cycle methods, which are available only to class components, allowing us to "hook" events and change state as needed. As discussed in *Chapter 4*, *React Lifecycle Methods*, this is a powerful aspect of a class component and will perhaps be a deciding factor when we need to decide which way to go.

Following is an example where we see state initialized with the greeting **`Hello World!`** in a class component. **`3`** seconds after the component has loaded, we change the message to **`Welcome to React!`** using the **`setState()`** method. We can refer to the following **`App.js`** file to see how state is handled in React:

```
class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      greeting: "Hello world!"
    };
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({ greeting: "Welcome to React!" });
    }, 3000);
  }
  render() {
    const { greeting } = this.state;
    return <div className="App">{greeting}</div>;
  }
}
```

Function components are stateless, which traditionally means they cannot have state. In body block, however, variables can be defined, and these may be evaluated for a render. When they change, this does not cause a re-render.

We have inferred that although class components are powerful, the advantage of function components is that even though they are concise and simple, they are efficient. With React 16.8, we now have a concept called Hooks that allows us to access features such as state and life cycle events in function components too.

To use state in a function component, the **`useState`** hook can be used. This takes the initial value of the state as the parameter and returns a pair of values to the current state and a function that updates it. Here is an example of using **`useState`**, a hook that allows us to use state.

Here's a state in a function component using the **useState** hook:

```
import React, { useState } from 'react';
function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Along with **useState**, we also have other methods, such as **useEffect**, **useContext**, and **useReducer**, that provide different features. We will look at Hooks in detail in later chapters of this book. For now, it is enough to know that Hooks provide enhanced features to function components.

With the skills gained from the preceding two exercises in the chapter, let's utilize them with the following activity.

ACTIVITY 5.01: CREATING A COMMENTS SECTION

The objective of this activity is to build a comment section that renders comments showing a set of comments along with the user's name and profile image and that have up to one level nesting, which means one comment would be nested within the other. We will need to break the UI into both class components and function components to create this.

We are provided with data as JavaScript Object Notation, or JSON, which is the preferred format for use in JavaScript applications. This can be replaced with dynamic content from a server during integration.

Again, for the profile images in our example, we are using various images from Unsplash: https://unsplash.com/s/photos/portrait?orientation=squarish

Here is the **`.json`** file provided that could be used for this activity:

`comments.json`

```
{
"comments": [
  {
    "name": "John Smith",
    "text": "This is an awesome page. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean scelerisque, purus ac feugiat eleifend, ex.",
    "image": "https://images.unsplash.com/photo-1522075469751-3a6694fb2f61?auto=format&fit=crop&w=200&q=80",
    "time": "Oct 02, 2019",
    "comments": [
      {
```

The complete code can be found here: https://packt.live/2yV13wK

The following steps will help you to complete the activity:

1. Analyze the data provided to understand the schema and nested structure.
2. Think of the app structure with a component hierarchy.
3. Create the app and start running the app using a command-line tool.
4. Replace the default component with a custom container component.
5. Create a comment component that receives a comment and renders the properties.
6. Add data to the container component, loop through the data to add the comment component, and pass comment data to it.
7. Add styling to the comment component.
8. Since the comments are nested, add a condition to check whether child comments are available and loop.
9. Add styling as necessary to the components to match the intended design.

The final output should look like this:

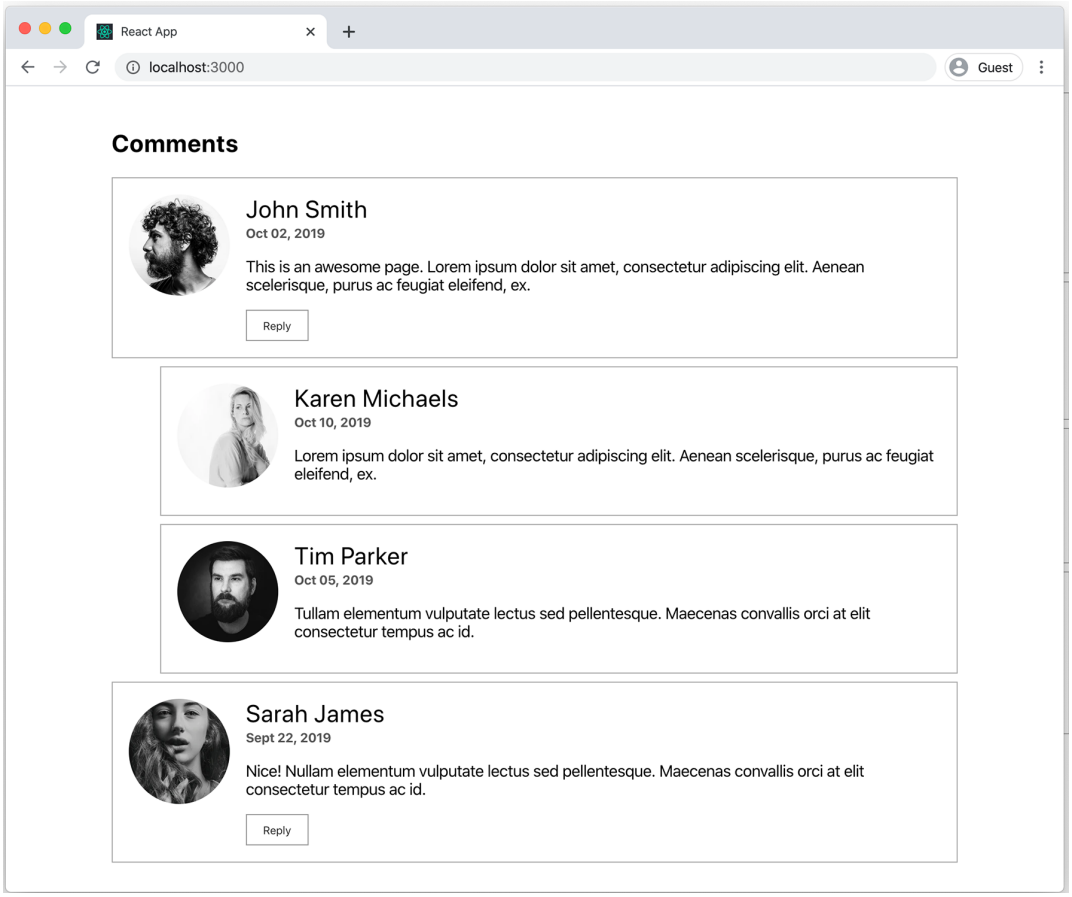

Figure 5.14: Final comments section app

> **NOTE**
>
> The solution steps for this activity can be found on page 637.

As next steps, we can add interaction to the components for which we will need to not just use props but mutate the state and pass it down to the components. We will be able to do this once we learn to work with state and props in the coming chapters.

SUMMARY

In this chapter, we have not just seen the way React components can be built, but also principles such as atomic design, which help us to think of structuring our components. We've learned the differences between class and function components and have also seen how we can build apps using them as reusable components. This has provided us with an essential toolkit to start building our applications of varying complexity and sizes.

In the next chapter, we will dive deeper to understand states and props for the components so that we may also add interactivity and better understand data flow.

6

STATE AND PROPS

OVERVIEW

This chapter will teach you how to add interactivity to React components using states and props. You will learn how to handle state in a React application and how to change state variables according to the requirements. You will also learn how to pass props down in components and the right usage of state and props.

INTRODUCTION

While building React applications, to improve their quality, we should use industry best practices. Data and its flow are important in our applications and for guidance, it is important to consider the Model-View-Controller (MVC) architecture. MVC is a common architectural pattern that is comprised of three main logical components: the Model, the View, and the Controller. The **Model** relates to data in the application and might even connect to a database. The **Controller** holds any business logic and causes changes to be made to the Model. Finally, the **View** is the presentation layer where changes in the Model are reciprocated.

In information technology, another aspect to consider is that software systems can store information about user interactions and events in something called the state. It is used to serve relevant content. A system that uses state is said to be stateful.

The Model part of the MVC architecture and the state in a stateful system are key to data integrity. In React applications, this data is held in the states and props of the components. A state is a local variable that is mutable, which is why it's used to initialize and hold information regarding the current status of the application or component while props are passed down to the component from a parent.

STATE IN REACT

When a user interacts with a software system, in order to provide relevant output, the system needs to be aware of the user inputs and also any previous interactions between the user and the system. This can then be used to evaluate the output. In React, such information can be stored in a data store called **state**, which is available to the components. State allows us to make the UI interactive, where interactions such as mouse and keyboard events trigger changes in the Model store, effectively changing the rendered component.

Let's look at an example of a component that toggles the component's state to show a greeting message when a button is clicked:

```
class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      isActive: false
    };
  }
  render() {
    const { isActive } = this.state;
```

```
    return (
      <div>
        <button
        onClick={() => {
          this.setState({ isActive: !isActive });
        }}
      >
      Say Hello
      </button>
      {isActive && <div>Hello World</div>}
    </div>
  );
  }
}
```

The preceding code creates a component where, in order to evaluate whether the greeting is shown, the state of the class component is initialized with a **`boolean`** value called **`isActive`**. This value can be altered using **`this.setState`**. The **`Hello World`** greeting is rendered when the **`boolean`** value is set to **`true`**. A button is rendered that, when clicked, toggles the state value to **`true`** or **`false`**, depending on the current value.

Something we should take into consideration while building React applications is that we need to take a pragmatic approach and think of the minimal set of changes or the mutable state that is required for the app; otherwise, we might end up causing side effects and memory issues in our application.

INITIALIZING AND USING STATE

Now, let's take a look at how we can define and use states.

SETTING STATE

State can only be defined and initialized for a class component in the following ways:

- As a property of the **`Component`** class:

```
class App extends Component {
state = {
  count: 1
};
```

  In the preceding code, we are initializing **`state`** as a property within the class and adding **`count`** with the value **`1`**.

- In the class constructor:

```
class App extends Component {
  constructor (props) {
    super(props);
    this.state = { count: 1 };
  }
```

Here, we have declared the constructor and within it, we are defining **`state`** as a property using **`this.state`**.

> **NOTE**
>
> When a constructor is used, **`super(props)`** should be the first statement in it so that **`props`** become available and can be used.

SETSTATE

State can be modified using the **`setState()`** method in React. The syntax is as follows:

```
setState(updater, [callback])
```

The first parameter, **`updater`**, is the value being set as an object, while the second parameter is an optional callback function that gets called right after the value is set and the component is re-rendered.

Changing state will always cause the component to re-render, and this can be controlled by using the React **`shouldComponentUpdate`** life cycle method.

Now that we have learned how to initialize state in class-based component, let's discuss how to create custom methods, which will allow us to structure our code and mutate our state.

CUSTOM METHODS AND CLOSURE

While building our components in React, we often write custom methods and utility functions that get called by different events, including user events. These functions may also need to have access to the class object in order to make changes to an object, and so we need to have it available in scope.

When attaching a function to an event listener, the object can be brought into scope using a closure. A closure in JavaScript combines a function with references to the surroundings and thus provides access to the scope of the outer function or object. We use a closure to change state and either attach it to an event or pass it down as a prop.

> **NOTE**
>
> For more information on JavaScript scopes and closures, refer to the following links:
>
> https://developer.mozilla.org/en-US/docs/Glossary/Scope
>
> https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures

In the next exercise, we will use a closure to access the state and mutate it.

EXERCISE 6.01: BUILDING AN APP TO CHANGE THE THEME

Let's build an app where we can use the state and toggle the theme between light and dark mode. The user should be able to click a button and alter the state. We will store the state, called theme, as a string with a **`light`** or **`dark`** value. To do so, perform the following steps:

1. Using the command line, create a new React project called **`app-theme`** and navigate to the folder. Then, run the app in development mode:

```
npx create-react-app app-theme
```

2. Navigate to the folder using the following command:

```
cd app-theme/
```

3. Run the app in development mode using the following command:

```
 yarn start
```

4. Delete the **`src/logo.svg`** file and the contents of **`src/App.css`**.
5. Replace the content of the **`App`** component with the new content to be rendered.

Inside **`App.js`**, add the following code:

`App.js`

```
17  render() {
18  return (
19  <div>
20    <div className="jumbotron">
21      <div className="container">
22      <h1>Hello, world!</h1>
23      <p>
24      This is a template for a simple marketing or informational
25      website. It includes a large callout called a jumbotron and three
```

`The complete code can be found here:` https://packt.live/3dBEyM2

6. Add basic styles for the light theme. We will use classes from bootstrap to start styling our app and for that, we will need to add the following **`import`** statement to CSS. In **`App.css`**, add the following line of code:

```
@import url(https://cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/3.3.7/css/bootstrap.min.css);
```

This will render the following light screen with the content:

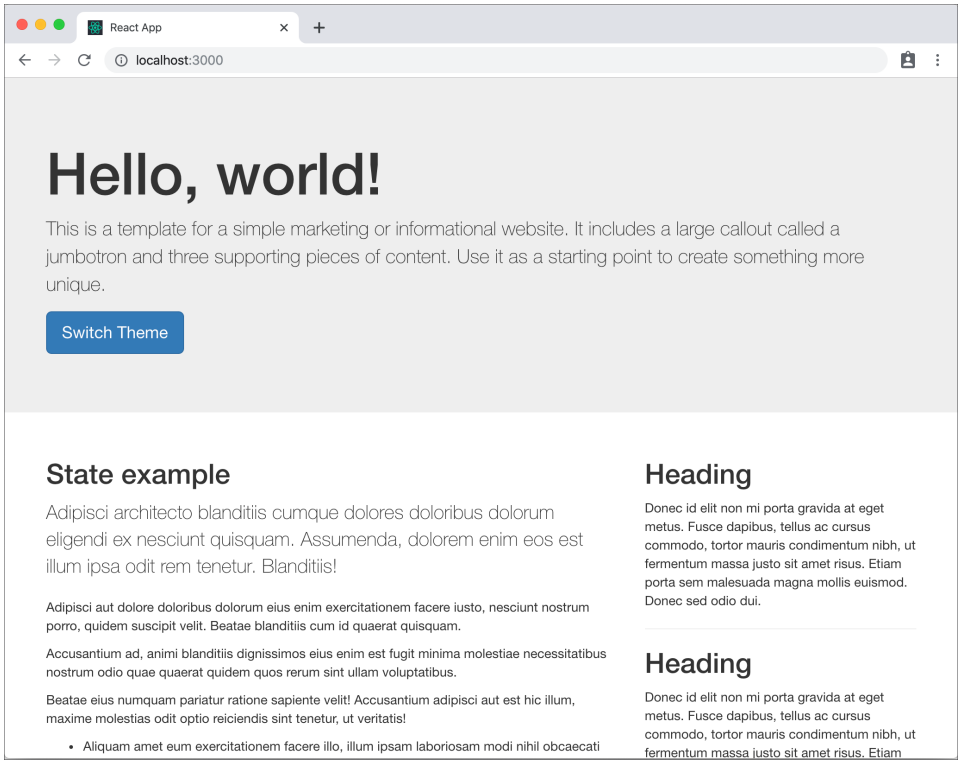

Figure 6.1: Hello World component

Now, we want to toggle the theme of this component. We want to change the theme of the component to a dark theme.

7. Write the logic to show the dark theme. Inside **App.js**, define the value for the theme in the state of the component, the value of which can be used to change the **css** class of the **div** wrapper element. Initialize state with the value for **theme** set to **light**:

App.js

```
5 class App extends Component {
6  constructor(props) {
7    super(props);
8    this.state = {
9      theme: "light"
10    };
16  }
17  render() {
18   return (
19    <div className={`${this.state.theme}-theme`}>
20      <div className="jumbotron">
21        <div className="container">
22          <h1>Hello, world!</h1>
```

The complete code can be found here: https://packt.live/3dBEyM2

With the value of **theme** set to **light**, the **div** gets the class name **light-theme**:

8. Change the theme based on user interaction. We will do this by adding a function with a closure to change the state based on the previous value. To set the state value, in **App.js**, we can access the state and the **setState** method of the component:

```
class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      theme: "light"
    };
  }
  toggleTheme() {
    const theme = this.state.theme === "light" ? "dark" : "light";
    this.setState({ theme });
  }
```

9. To access the class object using **`this`** keyword, we will need to bind it to the method, which can be done in the constructor. In **`App.js`**, use **`this`** keyword to bind the theme:

```
class App extends Component {
  constructor(props) {
    super(props);
    this.toggleTheme = this.toggleTheme.bind(this);
  }
  state = {
    theme: "light"
  };
  toggleTheme() {
    const theme = this.state.theme === "light" ? "dark" : "light";
    this.setState({ theme });
  }
```

With the preceding changes, when the light or dark theme is selected, the class name that's applied will be **`light-theme`** or **`dark-theme`**, respectively.

10. Bind the **`toggleTheme`** function to the **`onClick`** event. This is able to access the object and methods of the class since **`this`** keyword was bound to it in the constructor:

```
class App extends Component {
  render() {
  return (
  <div>
    <div className="jumbotron">
     <div className="container">
      <p>
        <button
           className="btn btn-primary btn-lg"
            onClick={this.toggleTheme}
             Switch Theme
              </button>
            </p>
          </div>
        </div>
```

11. Add CSS styles for the dark theme in **`App.css`**:

```
@import url("https://cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/3.3.7/css/bootstrap.min.css");

.dark-theme {
  background: #333;
  color: #fff;
}
.dark-theme .jumbotron {
  background: #444;
}
```

Now, we can test that the style will work when the dark theme is applied.

The output is as follows:

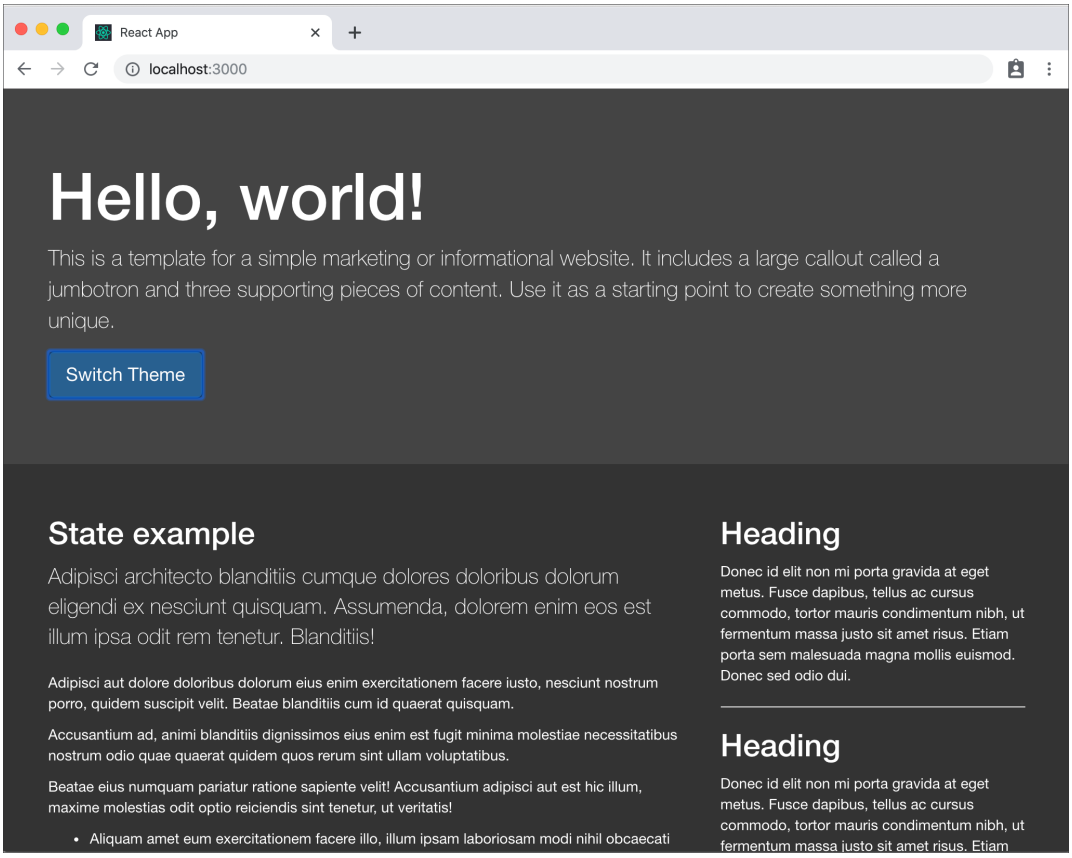

Figure 6.2: Dark theme component

With these changes, the screen now changes the theme when a user clicks on the button, the theme toggles between light and dark.

PROPS IN REACT

In our previous chapters and examples, we were introduced to React components and we also briefly saw how data can be passed from a parent component to a child component. This is a simple implementation of props.

Props in React is a way of passing data from parent to child. Let's look at a simple example where we pass a name to a component that renders a message. Setting props in our JSX is similar to setting an attribute in XML or HTML. Here's an example of sending props to the **`HelloMessage`** component:

```
function App() {
  return (
  <div>
  <HelloMessage name="John" />
  </div>
);
}
```

The props that have been sent to a class component can be accessed using **`this.props`**. Here's an example of the **`HelloMessage`** component receiving **`name`** as a prop:

```
import React, { Component } from "react";
export class HelloMessage extends Component {
  render() {
    return <h1>Hello {this.props.name}!</h1>;
  }
}
```

In a function component, however, props are received similar to how an object is sent as a parameter in a JavaScript function. An example of a function component receiving **`name`** as a prop is as follows:

```
import React from "react";
export const HelloMessage = props => <h1>Hello {props.name}!</h1>;
```

CHILDREN PROP

The content within the opening and closing tags (that is, between **>** and **<**) of a component in JSX is passed as a special prop called **`children`**. **`children`** is a special prop, the content of which is passed to the component, and only then is it aware of what it is.

Let's modify the preceding component to use **`children`** prop. An example of sending a **`Hello`** message as a **`children`** prop is as follows:

```
function App() {
  return (
    <div>
      <HelloMessage name="John">Hello</HelloMessage>
    </div>
  );
}
```

Here, since the **`HelloMessage`** component encompasses the word **`Hello`**, it is made available as the **`children`** prop. An example of using the **`children`** prop is as follows:

```
import React from "react";
export const HelloMessage = props => <h1>{props.children} {props.name}!</
h1>;
```

As you can see, the **`children`** prop is available just like other props and can be either a string or a part of JSX that gets rendered. This allows us to nest components by wrapping them around parent components and thus logically separate code blocks into components and structure our app more effectively.

PROPS ARE IMMUTABLE

React has a unidirectional data flow, which means that data flows from parent to child and never the other way around. Another caveat of React is that props cannot be changed or mutated within a component. Props are similar to parameters to a function and the value of the state may depend on changes that are made to it. If props were mutable, they could trigger an infinite loop and memory leaks, which would be undesirable. Since props are passed from parent to child, only the parent would be able to change the props before they are passed. Otherwise, this could cause changes outside the scope of the component or side effects, making the application inconsistent.

Developers who are new to these concepts might find this challenging to wrap their heads around. In the following exercise, we will learn how state and props can work in conjunction, where the parent component, acting as a container, has a state that gets passed to child components as props. When a change in values is required, the methods in the container component that are passed as callbacks can be used to trigger the change in state, effectively changing the props to the child components.

EXERCISE 6.02: CREATING A MODAL SCREEN

In this exercise, we will create a screen with a modal that is controlled using state and props. We are going to modify the **`.css`** file to show an overlay. We will add a popup and some text nested in the component, making it available as a child that will be shown as a layer on top of the component. Then, we will add a button so that we can close the popup:

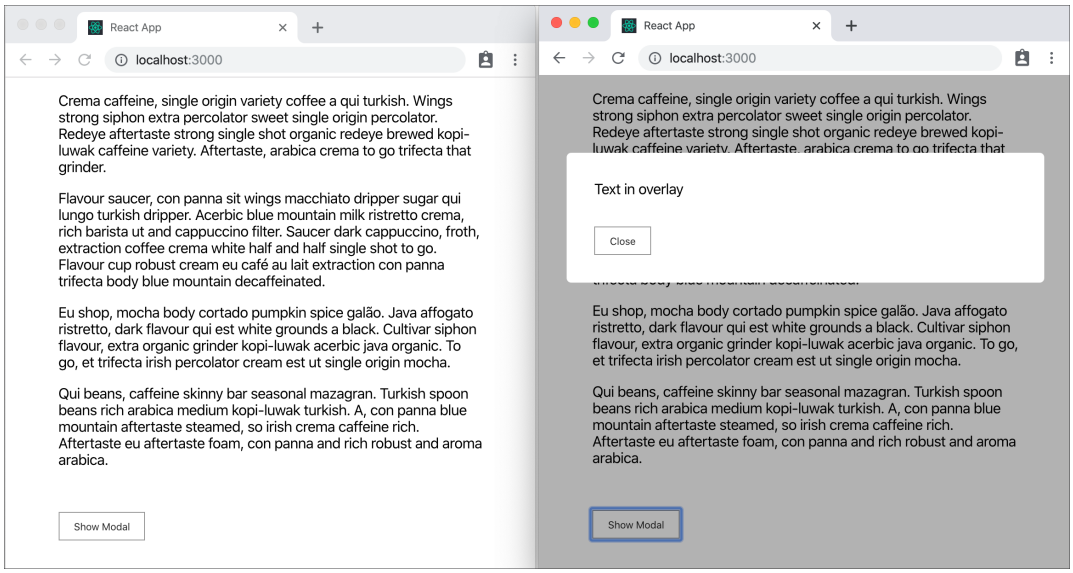

Figure 6.3: Modal component

Perform the following steps:

1. Use the command line to create an app called **`react-state-props`**. Then, navigate into the folder and run the app in development mode:

```
npx create-react-app react-state-props
cd react-state-props/
yarn start
```

2. Delete **`src/logo.svg`**.

3. Replace the content of the **`App`** component and add some content to be rendered with basic styles:

`App.js`

```
21  return (
22    <div className="container">
23      <div>
24        <p>
25          Crema caffeine, single origin variety coffee a qui turkish. Wings
26          strong siphon extra percolator sweet single origin percolator. Redeye
30        </p>
```

`The complete code can be found here:` https://packt.live/2YLJRmi

4. Replace the CSS in **`App.css`** file with the following code so that the content sits at the center of the screen:

```
.container {
  max-width: 960px;
  margin: 0 auto;
  padding: 0 10%;
}
```

This will render the following screen with the content:

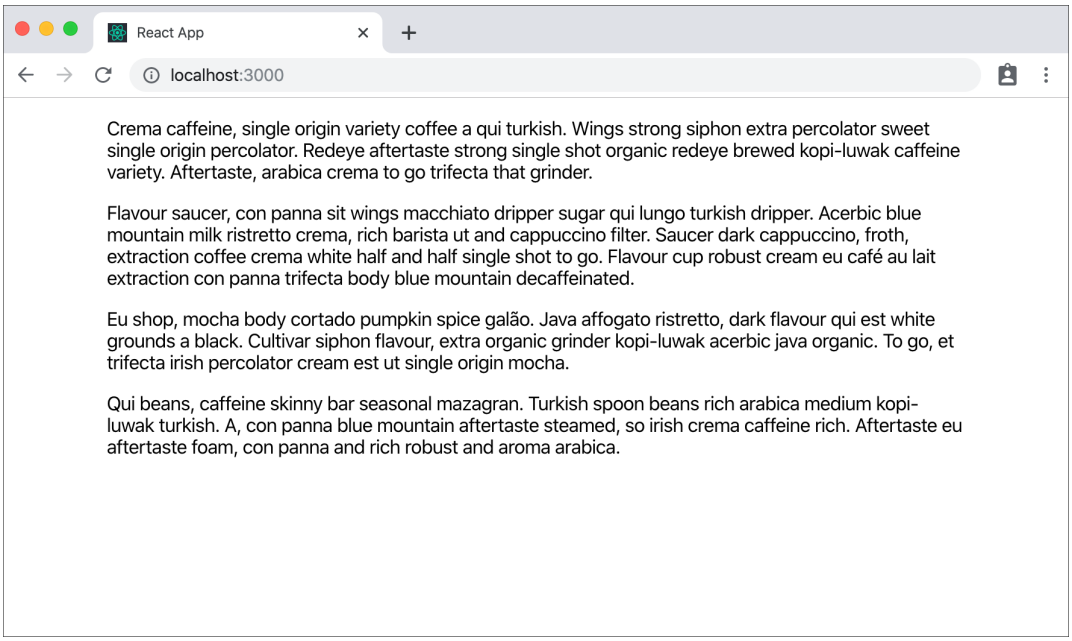

Figure 6.4: App component

5. Add some markup to show an overlay in **`App.js`**:

```
import React from "react";
// styles
import "./App.css";

function App() {
  return (
    <div className="container">
      <div> …
      </div>
      <div className="modal">
        <div className="modal_content">Text in overlay</div>
      </div>
    </div>
  );
}

export default App;
```

6. Similarly, to show an overlay, add the following CSS code in **`App.css`**:

```
.container {
  max-width: 960px;
  margin: 0 auto;
  padding: 0 10%;
}
.modal {
  background: rgba(0, 0, 0, 0.3);
  position: fixed;
  top: 0;
  right: 0;
  bottom: 0;
  left: 0;
  padding: 30px;
}
.modal_content {
  max-width: 500px;
  margin: 10vh auto;
  background: white;
```

```
  padding: 30px;
  border-radius: 5px;
}
```

We are using a CSS position that's fixed to add the overlay and adding a background with a dark semitransparent overlay. We've also added some styling to the content to give it a background color, width, and spacing so that it sits well on the screen:

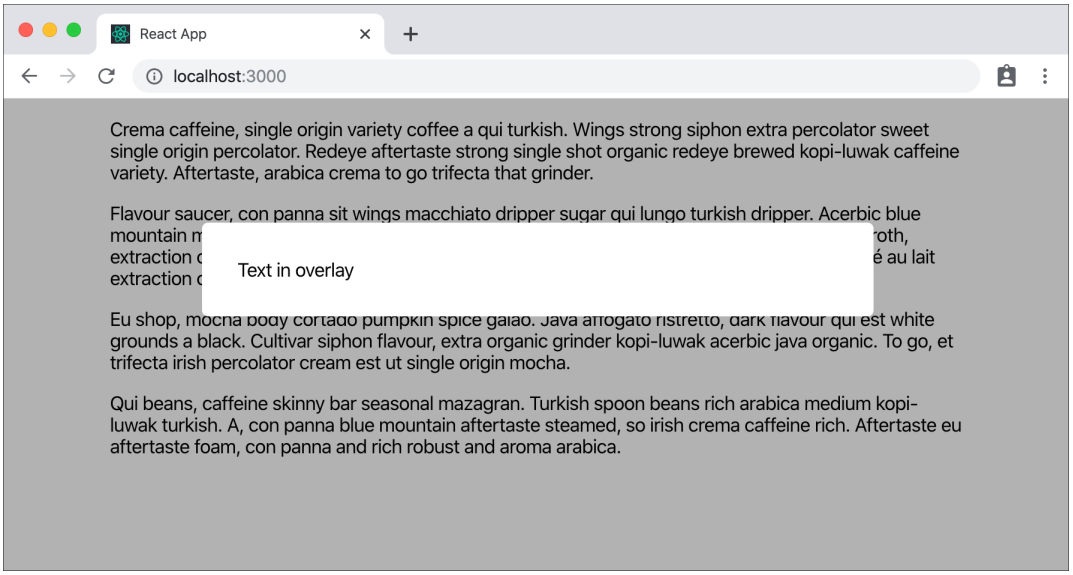

Figure 6.5: Text in the overlay of the modal

7. Let's improve the structure by moving the code and style for the modal into a file in **`App.js`**:

```
import React, {Component} from "react";
// styles
import "./App.css";
// component
import { Modal } from "./components/Modal";

  function App(){
  return (
    <div className="container">
      </div>
      <Modal>
        Text in overlay
```

```
          </Modal>
        </div>
      );
    }
    export default App;
```

8. Modify the contents of **`/src/components/Modal/index.js`**:

`index.js`

```
5  class Modal extends Component {
6  render() {
10    return (
11      <div className="modal">
12        <div className="modal_content">
13          {this.props.children}
14        </div>
```

`The complete code can be found here:` https://packt.live/2US6DHW

9. Move the css to **`/src/components/Modal/styles.css`**:

```
    .modal {
      background: rgba(0, 0, 0, 0.3);
      position: fixed;
      top: 0;
      right: 0;
      bottom: 0;
      left: 0;
      padding: 30px;
    }

    .modal_content {
      max-width: 500px;
      margin: 10vh auto;
      background: white;
      padding: 30px;
      border-radius: 5px;
    }
```

10. To make it interactive, add a Boolean value to a state that can take values of true or false. Call it **`showModal`**. The modal will only be visible if the value is true in **`/src/components/Modal/index.js`**

```
    import React, { Component } from "react";
    // styles
    import "./styles.css";
```

```
class Modal extends Component {
  constructor(props) {
    super(props);
    this.state = {
      showModal: true
    };
  }
  render() {
    if (!this.state.showModal) {
      return null;
    }
    return (
      <div className="modal">
        …
      </div>
    );
  }
}
```

The preceding state is initialized with the value for **`showModal`** set to **`true`**. With the **`if (!this.state.showModal)`** condition, we do not render the component when the value is set to **`false`**. We can change the value in the file to **`false`** to see that the modal does not render.

11. Add a function that can toggle the value of our state based on user interaction using the **`setState`** method in **`/src/components/Modal/index.js`**:

```
import React, { Component } from "react";
// styles
import "./styles.css";

class Modal extends Component {
  constructor(props) {
    super(props);
    this.state = {
      showModal: true
    };
  }
  toggleModal() {
    this.setState({
      showModal: !this.state.showModal
    });
```

```
    }
  …
  }

  export { Modal };
```

12. Bind **`this`** keyword to this function in the constructor to make this work in **`App.js`**:

```
  class Modal extends Component {
    constructor(props) {
      super(props);
      this.state = {
        showModal: true
      };
      this.toggleModal = this.toggleModal.bind(this);
    }
```

13. In **`/src/components/Modal/index.js`**, a button can be added with styling that changes state and effectively removes the modal from the screen:

```
  class Modal extends Component {
  …
  toggleModal() {
  …
    render() {
      if (!this.state.showModal) {
        return null;
      }
      return (
        <div className="modal">
          <div className="modal_content">
            {this.props.children}
            <div>
              <button onClick={this.toggleModal}>Close</button>
            </div>
          </div>
        </div>
      );
    }
  …
```

14. Modify the **`App.css`** file with the following code:

```
.container {
  max-width: 960px;
  margin: 0 auto;
  padding: 0 10%;
}
button {
  margin-top: 2rem;
  border: solid 1px #999;
  padding: 0.5rem 1rem;
  cursor: pointer;
}
```

The output will be as follows:

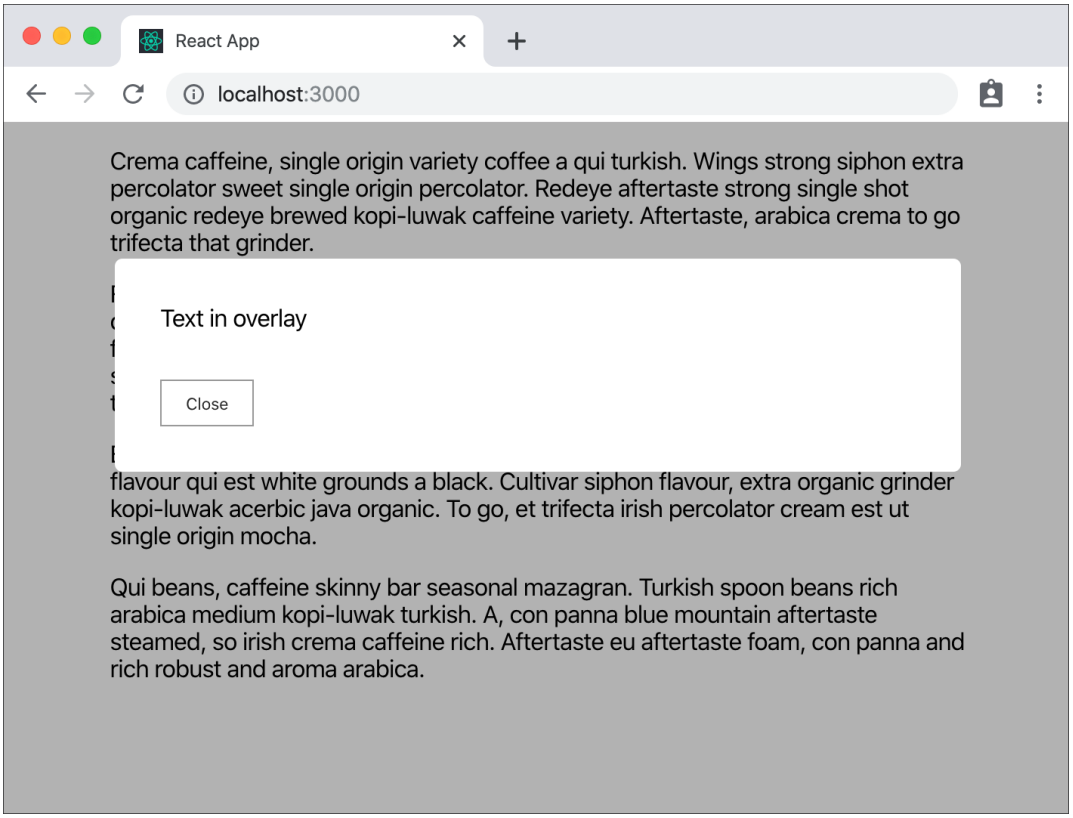

Figure 6.6: Overlay component in the modal

This gives us a functioning modal where, when you click on the button, it closes the modal. However, with the state being local, we have no way for the parent to trigger and change it.

The solution to this is to lift the state to the parent component. So, the changes affect the parent and it passes the values to the children.

15. Convert the **`App`** into a class component:

```
class App extends Component {
  render() {
    return (
      <div className="container">
        <div>
        </div>
      </div>
    );
  }
}
export default App;
```

16. In **`App.js`**, add state to the app using a constructor and add the method that will toggle the state:

```
class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      showModal: false
    };
    this.toggleModal = this.toggleModal.bind(this);
  }
  toggleModal() {
    this.setState({
      showModal: !this.state.showModal
    });
  }
```

17. Add the **`Show Modal`** button to show the modal and attach its **`click`** event to the **`toggle`** function we created:

`App.js`

```
5  import { Modal } from "./components/Modal";
6
7  class App extends Component {
8  constructor(props) {
…
15  toggleModal() {
…
19  }
20  render() {
21    return (
22      <div className="container">
```

`The complete code can be found here:` https://packt.live/2Ye4ZCM

18. In **`App.js`**, pass the value of the state and the function as props and add a **`Show Modal`** button to handle the click:

```
class App extends Component {
  render() {
    return (
      <div className="container">
        <div>
  …
        </div>
        <button onClick={this.toggleModal}>Show Modal</button>
        <Modal showModal={this.state.showModal}
        toggleModal={this.toggleModal}>
          Text in overlay
        </Modal>
      </div>
    );
  }
}
export default App;
```

19. In **`/src/components/Modal/index.js`**, refactor the **`Modal`** component so that it uses props instead of state:

`index.js`

```
5  class Modal extends Component {
6  render() {
7    if (!this.props.showModal) {
8      return null;
9    }
10    return (
11      <div className="modal">
12        <div className="modal_content">
13          {this.props.children}
14          <div>
15            <button onClick={this.props.toggleModal}>Close</button>
```

`The complete code can be found here:` https://packt.live/2US6DHW

This completes our exercise of getting the modal to appear when the user clicks on the button to show it, and to make it close when the user clicks the button to close it:

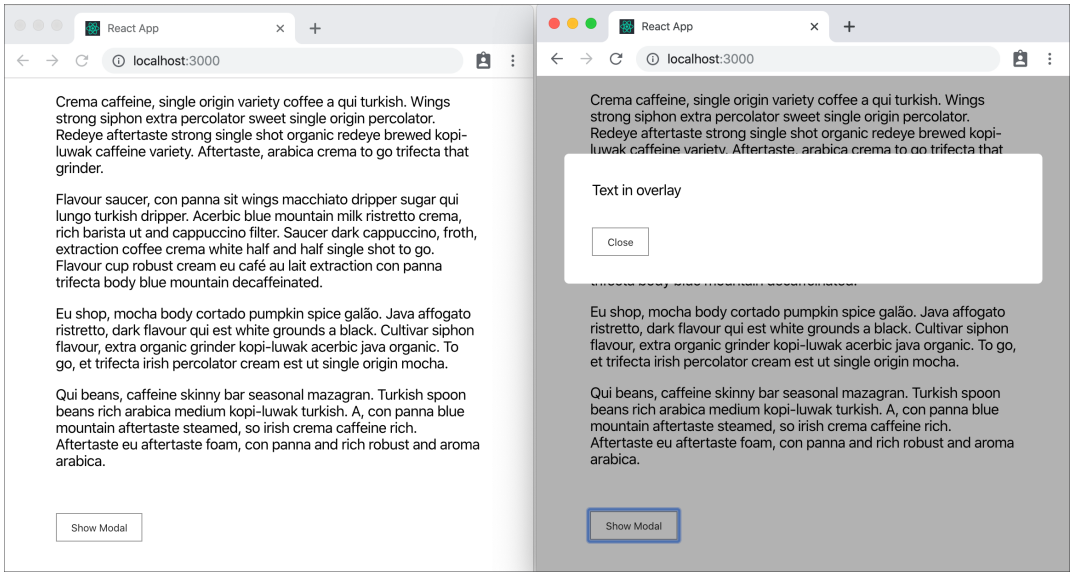

Figure 6.7: The modal screens with light and dark backgrounds

In this exercise, we've seen how state can be used to alter the rendered component while props can be passed down to child components. We also look at how this affects them. We used callback functions in the child components to make changes in the state of the parent component.

Let's take things further with state and props and check if we're able to build a more complex system that uses them.

ACTIVITY 6.01: CREATING A PRODUCTS APP PAGE

The objective of this activity is to build a product listing screen that shows a summary of the product that can be expanded so that we can view the description of the product. The products also have tags and a filter that will allow the tags to be filtered.

We have product information in the following JSON file, which we can use to render the components and build the interaction. This contains an array of products, which, in our case, is about different coffee drinks, along with some dummy text for the summary and tags that can be used to filter our products.

The **`.json`** file that you can use for this activity is called **`products.json`** and looks like this:

`products.json`

```
1  {
2  "products": [
3  {
4  "id": 1,
5  "name": "Espresso",
6  "price": 5.5,
7  "summary": "Galão, grinder mocha, filter plunger pot siphon mug variety est
   caramelization. Galão cortado medium latte organic aroma dripper lungo
    extraction crema. Bar variety spoon a fair trade filter iced. To go mocha
    wings irish kopi-luwak acerbic doppio kopi-luwak viennese saucer aroma
    arabica.",
8    "tags": ["black"]
9  },
```

The complete code can be found here: https://packt.live/2PZVK44

Perform the following steps:

1. Examine the requirements for the app and the structure of the provided product data to create an application design.
2. Create the app and use a command-line tool or **`cmd`** to start the application.
3. Create a component that will list and hold products.
4. Create a product component that will render the information about the product.
5. Use state to show and hide the product's description.
6. Create a component that will show tags with selection indication.

7. Use state at the app level to set the selected tag.
8. Use a closure to set the state and pass it as a prop.
9. Use the mutated state as a prop that will affect the change that's made in the child components.

The final output is as follows:

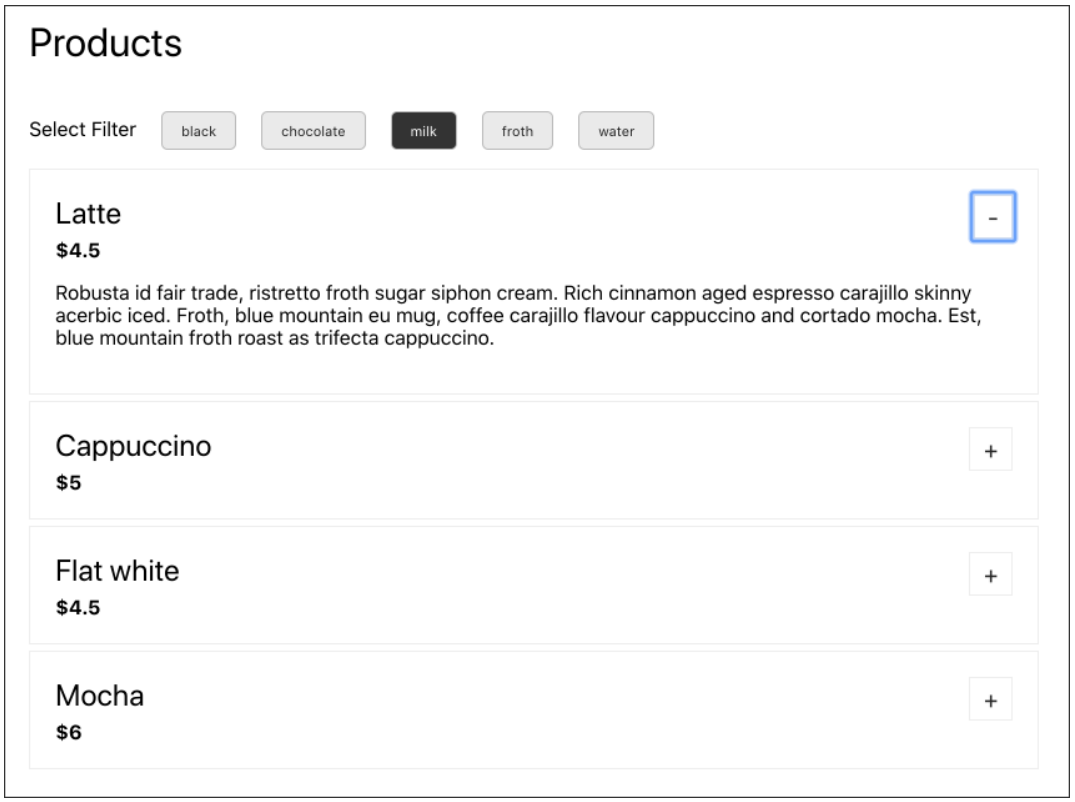

Figure 6.8: Products page

> **NOTE**
>
> The completed steps for this activity can be found on page 646

With this we have successfully completed this activity.

SUMMARY

In this chapter, we have seen how industry practices such as the Model View Controller architecture and application state allow us to think of data in a React application. These are represented in React using state and props.

First, we looked at how state can be initialized and used and how mutating it allows us to make our applications dynamic. We saw how we can write and use custom methods to manipulate state and how this allows us to create applications with complex logic.

We also went through props in React, which, coupled with callback functions, can be used to achieve a unidirectional data flow.

We covered examples that included a complex application with multiple components and used props and state for unidirectional data flow from one component to another.

With this understanding of state and props in React, we can build applications with complex data flows and start to delve further into complex concepts in React. In the next chapter, we will discuss how the components communicate with each other in React.

7

COMMUNICATION BETWEEN COMPONENTS

OVERVIEW

This chapter will equip you with essential React techniques required to pass data between React classes and functional components. First, you will be introduced to various ways of passing data from a parent to its child components. You will get a good hang of the techniques by practicing multiple hands-on exercises. This chapter will also introduce you to various design patterns, such as the Context API, higher-order components, and render props, which will enable you to solve common problems you will face when passing data between components. By the end of this chapter, you will have a firm grasp of the React techniques required to pass data.

INTRODUCTION

React is a UI library that helps us build single-page web and mobile applications. In single-page applications, the page is loaded only once. When the user interacts with the page, say, they click a button, the application will make a request to the server. When the page receives data from the server, it will only update the page partially without entirely reloading the page.

Since a single-page application has a lot of coding in one place, the code could get quite complex to manage. As we saw in the previous chapter, React allows us to split components into smaller pieces and helps in building reusable components with a consistent UI design.

Having reusable components means developers can use each component multiple times in their application as each component can be developed further separately and can contain its own business logic. It's much easier to test the individual smaller components as this will not bind any complex logic together.

While there are many benefits to having smaller reusable components, it is important to understand how these components communicate with each other and send data between each other properly.

For example, imagine a ToDo app with the ToDo container as the parent component and multiple **`ToDo`** items as the child components. When we look at the component structure, each **`ToDo`** container (parent component) will contain two child components – the **`ToDo`** item and a **`Complete`** button, as shown in the following diagram:

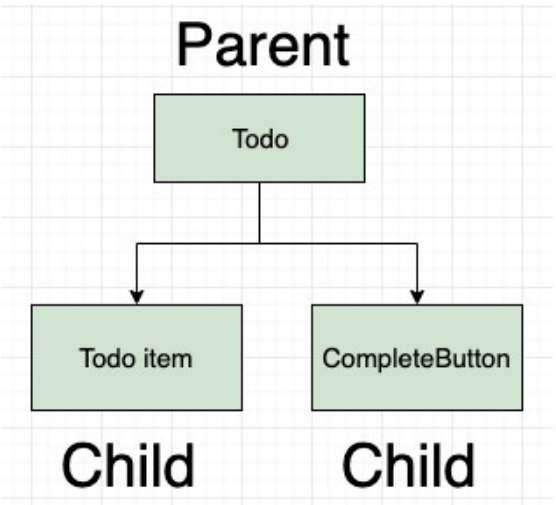

Figure 7.1: The parent and child components

In each **`ToDo`** list, we can update the status of a task by clicking the **`Complete`** button next to each item, as shown in the preceding diagram. When we change a task's status by clicking the button, we need to send data between components – that is, from the **`Complete`** button, which is a child component, to the **`ToDo`** container, which is the parent component. Similarly, the data is sent from the parent component (the **`ToDo`** container) back to the child component (the **`Complete`** button):

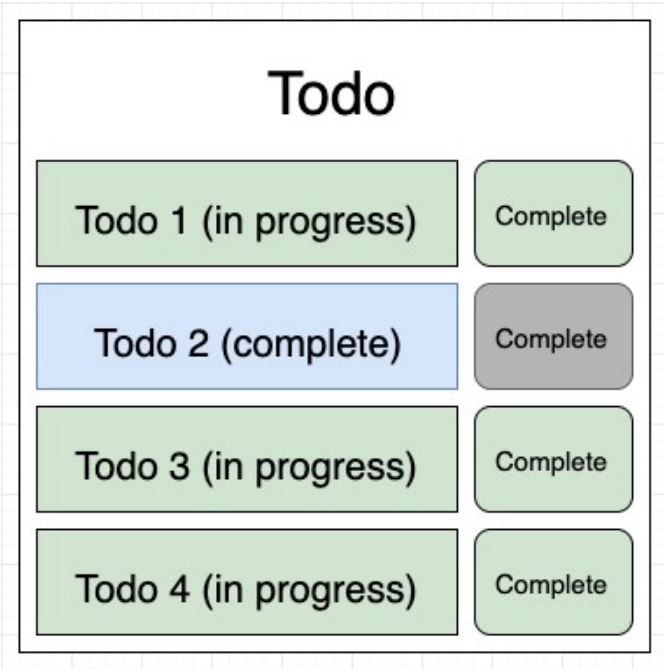

Figure 7.2: Interface of Todo

In the next section, we will learn how data is passed between parent and child components.

GETTING STARTED

There are many ways we can send data between components. In this chapter, we will cover four major ways of passing data:

- From a parent to a child component
- From a child to a parent component
- Between any components, such as sibling components

Through the React Context API, between components

> **NOTE**
>
> In React's hierarchical order, the parent component refers to the component located at the top, while the child components refer to the components located below the parent component in the hierarchy.

As you learned in the previous chapters, props are read-only, which means we cannot update props from a component. Instead, we are going to send data through props. In the React tree hierarchy, props can only be used to send data down from a parent to child components. This can cause a problem when we want to send data from child components to a parent component instead. We'll learn how to solve that problem in a bit.

Also, we will face another problem called prop-drilling. Prop-drilling refers to the process of passing down values to child components multiple levels down. Here, some of the child components in the middle might not actually make use of the data. For example, in the following diagram, the data is being sent from component 1 and being used in component 6:

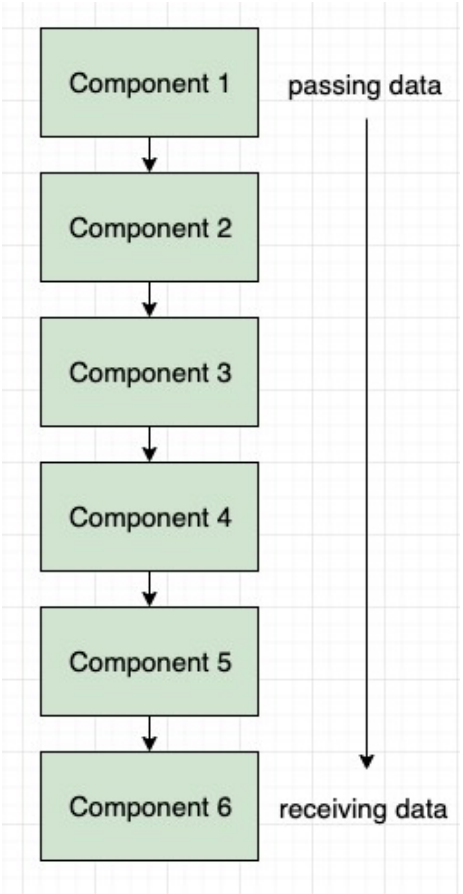

Figure 7.3: Prop-drilling problem

When we receive the data in component 6, we should pass it through all the components in the middle. This is the prop-drilling process.

For example, let's say we have an email app displaying our emails on the dashboard. The structure may look like this:

- Component 1: **`EmailApp`** component
- Component 2: **`EmailPanel`** component
- Component 3: **`EmailList`** component
- Component 4: **`EmailItem`** component
- Component 5: **`EmailContent`** component
- Component 6: **`EmailTitle`** component

Let's say the **`EmailApp`** component (component 1) receives the email data and we want to send the subject line of the email to the **`EmailTitle`** component (component 6) through props. In such a case, we need to pass the data through all the components between component 1 and component 6, which is exactly what happens when prop-drilling comes into the picture. Later in this chapter, we'll learn how to solve this problem by avoiding prop-drilling using the React Context API.

We are also going to create an app that will display a list of endangered animals. As we build the app, we will learn about all the important ways to pass data between components and allow them to communicate with each other.

PASSING DATA FROM PARENT TO CHILD COMPONENTS

React applications are typically built with many smaller components and, in most cases, each component sends and/or receives data from the others to make the app more functional. As we saw in the previous section, we can only send data down from parent components to child components.

This one-way data flow does the following:

- Helps components behave the same way since the data flow is predictable. So, we can make components reusable.
- Makes components easy to debug as we know where the data is coming from (a single source of truth).

To explicitly understand how data is transmitted from a parent to direct child components, let's move on to the following section.

PASSING DATA TO DIRECT CHILD COMPONENTS

Let's take a look at the following diagram, where data is being sent from a parent to its direct child component:

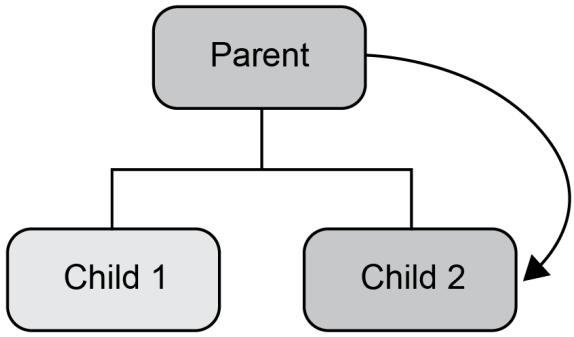

Figure 7.4: Sending data from a parent component to its direct child component

To send data to a direct child component, we provide a value to the props in the parent component, which the child component receives. As we learned in the previous chapter, props can be anything, including strings, numbers, arrays, and objects; however, props can also be a function or a component.

First, let's discuss how to send primitive values such as string, number, and Boolean to a direct child component through an example.

EXAMPLE 1: SENDING DATA FROM A PARENT COMPONENT TO A DIRECT CHILD COMPONENT

To explain how data is sent via a prop from a parent component, we are going to add a **`render()`** method in the **`App`** class. The **`render`** method will return an **`Animal`** component with a name as a property and **`Tiger`** as its value:

```
class App extends Component {
  render() {
    return <Animal name="Tiger" />;
  }
}
```

> **NOTE**
>
> The preceding code will not give an output since we have not created the **`Animal`** component yet. We will do that in the next section.

From the preceding code example, we can see that the prop called name has a value of **`Tiger`**. This value is sent as an object from the parent component, **`App`**, to the child component, **`<Animal>`**, as shown in the following diagram:

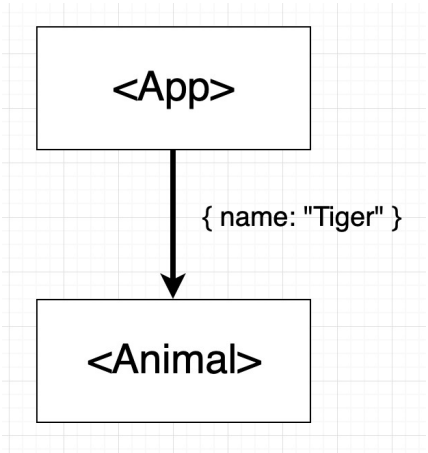

Figure 7.5: Prop as an object

Next, let's take a look at how to receive the prop values in the child class component.

EXAMPLE 2: RECEIVING DATA IN A CHILD CLASS COMPONENT

To understand how data is received, we will create an **`Animal`** class component. Then, we will add a render method that returns a **`<div>`** element. Inside the **`<div>`** element, we will add the text of **`Animal`**: To display the sent value, we will also add **`{this.props.name}`**, as shown in the following code:

```
class Animal extends Component {
  render() {
    return <div>Animal: {this.props.name}</div>;
  }
}
```

Now, let's see how data being sent and received via props happens in a child class component. When we combine the two preceding examples, we will get the following code:

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
  render() {
    return <Animal name="Tiger" />;
  }
}

class Animal extends Component {
  render() {
    return <div>Animal: {this.props.name}</div>;
  }
}
export default App;
```

If you run the preceding code, it will result in the following output:

```
// Animal: Tiger
```

In a class-based component, the props can be accessed using **`this.props`** from a Component instance. To receive the name prop value, we call **`{this.props.name}`** in the curly braces. The curly braces are JSX syntax for evaluating the JavaScript expression.

Based on this output, we can see that we have successfully sent the data from a parent component (in Example 1) and received it in a child component (in Example 2). Now, let's see how we can receive the data in a functional child component.

EXAMPLE 3: RECEIVING DATA IN A CHILD FUNCTION COMPONENT

In this example, we will create an **`Animal`** function component. We will pass a props value as an argument to the function component and return a **`<div>`** element. Inside the **`<div>`** element, we will add **`Animal`**: and **`{props.name}`**, like so:

```
const Animal = props => {
  return <div>Animal: {props.name}</div>;
};
```

As we can see from the preceding code example, we are receiving the prop, name, with its value, **`Tiger`**, from the parent component, **`<App>`**.

> **NOTE**
>
> Unlike the class-based component, we do not need **`this`** keyword to receive the name value from the prop as we have passed the props as an argument in the **`Animal`** functional component.

Run the following code in order to understand how data is received in a child function component:

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
  render() {
    return <Animal name="Tiger" />;
  }
}
const Animal = props => {
  return <div>Animal: {props.name}</div>;
};
export default App;
```

The output will be the same as before:

```
// Animal: Tiger
```

While the output is the same, here, we have received data in a child function component.

So far, we have only sent string data types across components. Now, let's send other data types such as number and Boolean as props and see what happens.

EXAMPLE 4: SENDING NUMBER AND BOOLEAN AS PROPS FROM THE PARENT COMPONENT

In the App parent component, we are going to add two props called number and endangered (along with the existing name prop) with two different datatypes: number and Boolean. Their respective values will be **`3890`** and **`true`**:

```
class App extends Component {
  render() {
    return <Animal name="Tiger" number= {3890} endangered={true} />;
  }
}
```

> **NOTE**
>
> The preceding code will not give an output since we have not defined the **`Animal`** component yet. We will do that in the next section.

In the preceding code example, we are sending props such as **`name`**, **`number`**, and **`endangered`** with their values as an object to the child component, **`<Animal>`**, from the parent component, **`<App>`**.

> **NOTE**
>
> Both the **`integer`** and **`Boolean`** values were enclosed in curly braces to keep the value type, such as **`integer`** and **`Boolean`**.

In the first two examples, we sent and received a single prop value. In this example, we sent two prop values of different data types. Now, let's see how we can receive them in class-based and functional child components.

EXAMPLE 5: RECEIVING NUMBER AND BOOLEAN VALUES IN CLASS-BASED AND FUNCTIONAL COMPONENTS

First, let's receive the number and Boolean values in a class-based component. To do this, in the **`Animal`** class component, we are going to add parentheses after return so that we have a multiline JSX expression. Inside **`<div>`**, we will add three **`<p>`** elements and inside each **`<p>`** element, we will add the following props:

```
Animal: {this.props.name}
Number: {this.props.number}
```

Let's design the **`Endangered`**: element like this: Add a JSX condition to display **`Yes`** if the value is **`true`** and **`No`** if the value is **`false`**:

```
class Animal extends Component {
  render() {
    return (
      <div>
        <p>Animal: {this.props.name}</p>
        <p>Number: {this.props.number}</p>
        <p>Endangered: {this.props.endangered ? 'Yes' : 'No'}</p>
      </div>
    );
  }
}
```

In the **`Animal`** child component, we are receiving props with the number and endangered statuses. For the endangered status, we check whether the value is true or false and if it's **`true`**, we display **`Yes`**; otherwise, we display **`No`**.

Now, let's see what happens when we run the code for the parent App (we created this in the previous example) and the child Animal class components together. We are doing this to see how data of multiple datatypes can be sent and received between them:

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
  render() {
```

```
    return <Animal name="Tiger" number={3890} endangered={true} />;
  }
}

class Animal extends Component {
  render() {
    return (
      <div>
        <p>Animal: {this.props.name}</p>
        <p>Number: {this.props.number}</p>
        <p>Endangered: {this.props.endangered ? 'Yes' : 'No'}</p>
      </div>
    );
  }
}

export default App;
```

The output of the preceding code will be as follows:

```
// Animal: Tiger
// Number: 3890
// Endangered: Yes
```

To receive the prop value in a child functional component, we just need to modify the previous code and drop this keyword, as shown here:

```
const Animal = props => {
  return (
    <div>
      <p>Animal: {props.name}</p>
      <p>Number: {props.number}</p>
      <p>Endangered: {props.endangered ? 'Yes': 'No'}</p>
    </div>
  );
};
```

> **NOTE**
>
> You can run the parent **`App`** component along with the child **`Animal`** function component to see the output.

The output will be similar to the previous output, as shown here:

```
// Animal: Tiger
// Number: 3890
// Endangered: Yes
```

DESTRUCTURING PROPS

To make the code simpler and more readable, let's destructure the props array object. Destructuring assignment was introduced in ES2015 (https://www.ecma-international.org/ecma-262/6.0/#sec-destructuring-assignment) and it allows us to extract values from arrays or properties from objects into their own variables.

For example, let's say we have an array called **`colors`**:

```
const colors = ['blue', 'red', 'green', 'pink'];
```

In the colors array, we are storing four **`color`** names as strings. We can destructure the array so that we can define the values that have been unpacked from the sourced variables:

```
const colors = ['blue', 'red', 'green', 'pink'];
const [waterColor, fireColor, lawnColor] = colors;
console.log(waterColor); // expected output: blue
console.log(fireColor); // expected output: red
console.log(lawnColor); // expected output: green
```

In this example, we have assigned the items in the **`colors`** array to three local variables – **`waterColor`**, **`fireColor`**, and **`lawnColor`** – using array destructuring. Notice that each variable is mapped to the corresponding item at the same index on the **`colors`** array. So, **`waterColor`** takes the value **`blue`**, **`fireColor`** takes the value **`red`**, and **`lawnColor`** takes the value **`green`**.

Let's utilize this concept to destructure our prop values.

EXAMPLE 6: DESTRUCTURING PROP VALUES IN A CHILD CLASS COMPONENT

In this example, we are going to destructure prop values in a child class component. Since we will be starting with the render method, we will add a destructuring assignment with the properties: **`name`**, **`number`**, and **`endangered`**. We are going to extract the values of these properties from the **`this.props`** object. We no longer need **`this.props`** inside the return method now, so we can remove it from there:

```
class Animal extends Component {
  render() {
    const { name, number, endangered } = this.props;
    return (
      <div>
        <p>Animal: {name}</p>
        <p>Number: {number}</p>
        <p>Endangered: {endangered ? 'Yes' : 'No'}</p>
      </div>
    );
  }
}
```

> **NOTE**
>
> For object destructuring, use a pair of enclosing parentheses **`{}`**.
> We used a pair of third brackets **`[]`** previously since we were performing array destructuring .

Now, we can use our return method to just display the values of the three properties.

We can also directly extract values if we have the assignment inside the function parameter, as shown in the following code. So, instead of having the destructuring assignment in the **`render`** method, we can move it inside the **`render`** method so that it acts as the function parameter:

```
class Animal extends Component {
  render({ name, number, endangered } = this.props) {
    return (
      <div>
        <p>Animal: {name}</p>
        <p>Number: {number}</p>
        <p>Endangered: {endangered ? 'Yes' : 'No'}</p>
```

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

If you want to see the output, combine the parent **`App`** component and the child **`Animal`** class component together, as follows:

```
import React, { Component } from 'react';
import './App.css';
class App extends Component {
  render() {
    return <Animal name="Tiger" number={3890} endangered={true} />;
  }
}
class Animal extends Component {
  render({ name, number, endangered } = this.props) {
    return (
      <div>
        <p>Animal: {name}</p>
        <p>Number: {number}</p>
        <p>Endangered: {endangered ? 'Yes' : 'No'}</p>
      </div>
    );
  }
}
export default App;
```

The output will be the same as what we got in the previous example:

```
// Animal: Tiger
// Number: 3890
// Endangered: Yes
```

For the function component, we can destructure the object directly in the function parameter.

EXAMPLE 7: DESTRUCTURING PROP VALUES IN A FUNCTION COMPONENT

In this example, we are going to destructure prop values in a child function component. To do this by using the preceding code, for the parameter, we will add parentheses and remove props. Then, we will add curly braces inside the parentheses and add the name, number, and endangered properties. Finally, inside **`return`**, we are going to remove the props in the JSX expression so that only the values of the **`{name}`**, **`{number}`**, and **`{endangered ? 'Yes' : 'No'}`** properties will be displayed:

```
const Animal = ({ name, number, endangered }) => {
  return (
    <div>
      <p>Animal: {name}</p>
      <p>Number: {number}</p>
      <p>Endangered: {endangered ? 'Yes' : 'No'}</p>
    </div>
  );
};
```

To see the output, we need to combine the parent **`App`** and child **`Animal`** components:

```
import React, { Component } from 'react';
import './App.css';
class App extends Component {
  render() {
    return <Animal name="Tiger" number={3890} endangered={true} />;
  }
}
const Animal = ({ name, number, endangered }) => {
  return (
    <div>
      <p>Animal: {name}</p>
      <p>Number: {number}</p>
      <p>Endangered: {endangered ? 'Yes' : 'No'}</p>
    </div>
  );
};

export default App;
```

The output should appear as follows:

```
// Animal: Tiger
// Number: 3890
// Endangered: Yes
```

So far, we have discussed how to send and receive primitive values in various ways. Now, let's take a look at how to send an object and receive it in the child component.

In regard to the preceding examples, instead of sending each prop separately, this time, we will gather all the props in an object and send it to the child component.

EXERCISE 7.01: SENDING AND RECEIVING OBJECTS AS PROPS FROM THE PARENT

In this exercise, we will work with objects and send and receive them from the parent and child components. We will continue to use the same code we wrote in the *Passing Data to a Direct Child Component* section. To do so, let's go through the following steps:

1. Open the **`App.js`** file inside the **`src`** folder.
2. Import the component from React:

```
import React, { Component } from 'react';
```

3. Construct an object called **`details`** and add some properties and values.
4. In return, remove all the props from the **`<Animal>`** component and add **`details`** as a prop:

```
class App extends Component {
  render() {
    const details = {
      name: 'Tiger',
      number: 3890,
      endangered: true
    };
    return <Animal details={details} />;
  }
}
```

5. When receiving the object in the child class component, destructure it in the same way as we did for the class-based component but this time for the function component:

```
class Animal extends Component {
  render() {
    const {name, number, endangered} = this.props.details;
    return (
      <div>
        <p>Animal: {name}</p>
        <p>Number: {number}</p>
        <p>Endangered: {endangered ? 'Yes' : 'No'}</p>
      </div>
    );
  }
}
export default App;
```

In the preceding code, we have specified the details for the destructuring assignment. For the function component, we will do something similar to what we did for the class-based component.

6. Instead of destructuring directly in the function parameter, take it out and create a destructuring assignment right above the **`return`** function, as shown in the previous examples:

```
const Animal = props => {
  const { name, number, endangered } = props.details;
  return (
    <div>
      <p>Animal: {name}</p>
      <p>Number: {number}</p>
      <p>Endangered: {endangered ? 'Yes' : 'No'}</p>
    </div>
  );
};
```

One more optional way of sending multiple props to a child component is to use the **`spread`** attribute as a **`spread`** operator and send whole props at once, as shown in the following code:

```
class App extends Component {
  render() {
    const details = {
      name: 'Tiger',
      number: 3890,
      endangered: true
    };
    return <Animal {...details} />;
  }
}
```

7. In the **`Animal`** component, we should receive the data by destructing the props instead of **`props.details`**:

```
const Animal = props => {
  const { name, number, endangered } = props;
  return (
    <div>
      <p>Animal: {name}</p>
      <p>Number: {number}</p>
      <p>Endangered: {endangered ? 'Yes' : 'No'}</p>
    </div>
  );
};
```

The output is as follows:

```
// Animal: Tiger
// Number: 3890
// Endangered: Yes
```

As we can see, we can receive the props sent by the spread attribute by doing what we did in *Step 5* for the class component and *Step 6* for the function component.

> **NOTE**
>
> Spread attributes are a JavaScript concept. According to JSX in Depth (https://reactjs.org/docs/jsx-in-depth.html#spread-attributes), "Spread attributes can be useful but they also make it easy to pass unnecessary props to components that don't care about them or to pass invalid HTML attributes to the DOM. We recommend using this syntax sparingly."

We recommend using the spread attribute with caution. Use it only if it is certainly necessary and makes the component more reusable. We will see how it is used later.

THE {CHILDREN} PROP

In React, there is a special prop called **`children`**. The **`children`** prop takes the DOM elements inside the component and passes them down to the child component.

For example, let's say we want to add a paragraph tag inside a child component from the parent component:

```
<ChildComponent><p>this is a paragraph</p></ChildComponent>
```

Here, we can receive the paragraph, **`<p> this is a paragraph</p>`**, from **`ChildComponent`** using a **`children`** prop, like this:

```
<p>this is a paragraph</p>
const ChildComponent = props => {
  return <div>{props.children}</div>;
}
```

We will practice how to receive the children prop in the following exercise.

EXERCISE 7.02: SENDING CHILD ELEMENTS USING THE CHILDREN PROP

In this exercise, we are going to practice how to send the child elements from the parent component and receive them through the children prop in the child component. We will use the same code from *Exercise 7.01*, *Sending and Receiving Objects as Props from the Parent*. Perform these steps to complete this exercise:

1. In the **`App`** parent component, add the **`<h1>`** element with the text **`Endangered Animals`**.
2. Inside **`src/App.js`**, write the following:

```
import React, { Component } from 'react';
class App extends Component {
  render() {
    return (
      <Animal name="Tiger" number={3890} endangered={true}>
        <h1>Endangered Animals</h1>
      </Animal>
    );
  }
}
```

3. To receive the child component, **`h1`**, from the function component, add an extra property, children, in the destructuring assignment. Add **`{children}`** right below the opening **`<div>`** element in the **`return`** method:

```
const Animal = props => {
  const {name, number, endangered, children} = props;

  return (
    <div>
      {children}
      <p>Animal: {name}</p>
      <p>Number: {number}</p>
      <p>Endangered: {endangered ? 'Yes' : 'No'}</p>
    </div>
  );
};
export default App;
```

The output of the preceding coding example should appear as follows:

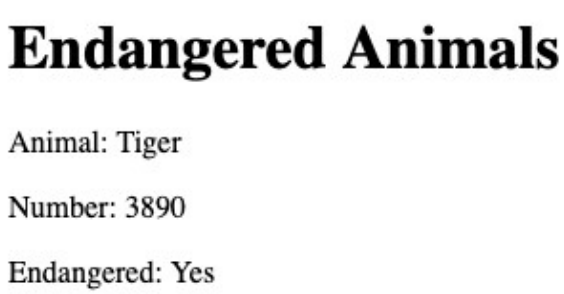

Figure 7.6: Output of the children prop

The children prop is a great thing to use when you need to reuse the child elements from the component. Furthermore, you can read each element by iterating with the **`React.Children`** (https://reactjs.org/docs/react-api.html#reactchildren) utility and modifying the element if necessary.

So far, we have looked at how to send and receive strings, numbers, Booleans, and objects. Now, let's discuss how to send an array and iterate within that array.

SENDING AND RECEIVING AN ARRAY THROUGH PROPS

In the previous sections, we learned how to receive data of various data types in various formats through props in a child component. In this section, we will discuss how to send and receive an array through props and display it in the child components.

To send an array, we would do exactly the same as what we did for the other types of data. Inside the render method, add the fruits array:

```
const fruits = [
{
  name: 'apple',
  color: 'red'
},
{
  name: 'banana',
  color: 'yellow'
},
{
```

```
  name: 'melon',
  color: 'green'
}
];
<FruitList list={fruits} />
```

In the preceding example, the **`fruits`** array contains three elements where each element has a **`name`** and **`color`**. Now, to receive the data from the child component, we can get it through a prop, just like we did for the other types of data we discussed in the previous sections:

```
const FruitList = props => {
  const fruits = props.list;
  return(<ul><li>Fruit list goes here</li></ul>);
};
```

Now, to render each array item from the array, we can use the **`map()`** method to loop the array.

> **NOTE**
>
> **`Array.Map`** is a part of the JavaScript methods.

Please make sure that you add a key to the outermost element inside **`map()`**. key is a special string attribute that helps React identify what element has been added, updated, or removed:

```
const FruitList = props => {
  const fruits = props.list;
  const fruitList = fruits.map((fruit, index) => (
    <li key={index}>
     <p>Name: {fruit.name}</p>
     <p>Color: {fruit.color}</p>
    </li>
  ));
  return(<ul>{fruitList}</ul>);
};
```

The complete code for the preceding example, which combines the parent and child components, looks as follows:

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
  render(){
    const fruits = [
    {
      name: "apple",
      color: "red"
      },
    {
      name: "banana",
      color: "yellow"
    },
    {
      name: "melon",
      color: "green"
    }
    ];

    return (
      <FruitList list={fruits} />
    );

  }
}
```

The functional component **`FruitList`** would like this:

```
const FruitList = props => {
  const fruits = props.list;
  const fruitList = fruits.map((fruit, index) => (
    <li key={index}>
     <p>Name: {fruit.name}</p>
     <p>Color: {fruit.color}</p>
    </li>
  ));
```

```
  return(<ul>{fruitList}</ul>);
};

export default App;
```

If you run the preceding code, the output will be as follows:

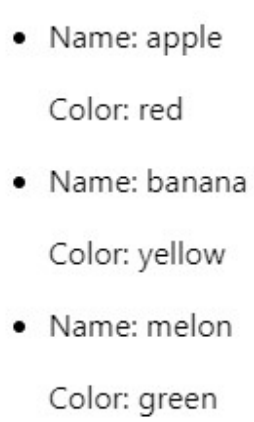

Figure 7.7: The FruitList child component

Alternatively, you can use **`map()`** directly inside **`return()`** like so:

```
const FruitList = props => {
  const fruits = props.list;
  return(
    <ul>
      {fruits.map((fruit, index) => (
        <li key={index}>
          <p>Name: {fruit.name}</p>
          <p>Color: {fruit.color}</p>
        </li>
      ))}
    </ul>
  );
};
```

We now have a good idea of how to send an array to the child components and display each array item in the child components using the **`map()`** method. Next, let's practice what we have learned so far through an exercise.

EXERCISE 7.03: SENDING, RECEIVING, AND DISPLAYING AN ARRAY FROM A PARENT TO A CHILD

In this exercise, we will work with arrays and learn how to send, receive, and display an array through props sent from a parent to a child component. We will add two more objects and put those objects in an array. To do so, let's go through the following steps:

1. Start with the parent **`App`** component. Inside the render method, define the **`details`** array:

```
class App extends Component {
  render() {
    const details = [
      {
        name: 'Tiger',
        number: 3890,
        endangered: true
      },
      {
        name: 'Brown Bear',
        number: 200000,
        endangered: false
      },
      {
        name: 'Red Panda',
        number: 10000,
        endangered: true
      }
    ];
```

2. In the **`return`** method, inside the **`Animal`** component, add **`<h1>`** with a heading, **`Endangered Animals`**:

```
    return (
      <Animal details={details}>
        <h1>Endangered Animals</h1>
      </Animal>
    );
  }
}
```

3. Now, let's receive the array from the child **`Animal`** class component. Assign the **`details`** prop in an array to the **`details`** variable:

```
class Animal extends Component {
  render() {
    const details = this.props.details;
  }
}
```

4. Loop the **`details`** array with a **`map`** method. Using the **`map`** function creates a new array and assigns it to the **`listDetails`** variable:

```
class Animal extends Component {
  render() {
    const details = this.props.details;
    const listDetails = details.map(detail => (

    ));
  }
}
```

5. Inside the **`map`** method, we are going to receive each object value with name, number, and endangered. Construct JSX with the **`<li>`**, **`<div>`**, and **`<p>`** elements and assign the constructed JSX to the **`listDetails`** variable. Finally, add **`{listDetails}`** inside **`<ul>`** in JSX. The complete code will look as follows:

```
class Animal extends Component {
  render() {
    const details = this.props.details;
    const listDetails = details.map(detail => (
      <li>
        <div>
          <p>Animal: {detail.name}</p>
          <p>Number: {detail.number}</p>
          <p>Endangered: {detail.endangered ? 'Yes' : 'No'}</p>
        </div>
      </li>
    ));

    return (
      <div>
```

```
            {this.props.children}
            <ul>{listDetails}</ul>
          </div>
        );
      }
    }
```

The output of the preceding code should appear as follows:

Figure 7.8: Output

This looks all good; however, if you open the Developer Tools (the *F12* key in Chrome) from your browser, you may see a similar warning message to the following:

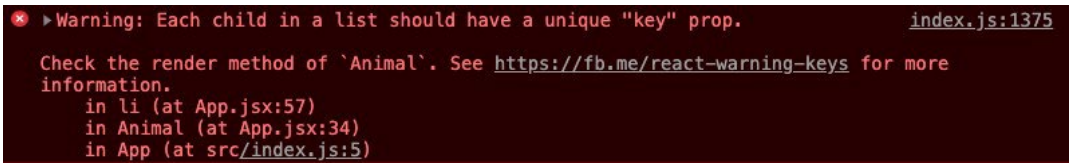

Figure 7.9: Array warning with no unique key prop

This has occurred because we have not assigned a unique key to each element inside the array for a stable identity. The key is required when you create a list of elements so that React can identify which element is being added, changed, reordered, and removed. The best way to choose the key is by getting an ID from data; however, as we have not sent an ID inside the array, we will use the item index as a key.

6. Let's add a unique key. Add index as the second argument inside the map method. Then, add a key attribute to **`<li>`** and add **`{index}`** to key as a value. Now, the code will look like this:

```
class Animal extends Component {
  render() {
    const details = this.props.details;
    const listDetails = details.map((detail, index) => (
      <li key={index}>
        <div>
          <p>Animal: {detail.name}</p>
          <p>Number: {detail.number}</p>
          <p>Endangered: {detail.endangered ? 'Yes' : 'No'}</p>
        </div>
      </li>
    ));
…
    return (
      <div>
        {this.props.children}
        <ul>{listDetails}</ul>
      </div>
    );
  }
}
```

By doing this, each **`<li>`** element obtains a key value and the warning sign should disappear. So far, we have constructed the JSX elements from the loop, **`<listDetails />`**, and added them to the return method. This time, we will discuss how we can directly loop the array inside JSX. For this example, we will update the code for the function component.

7. Alternatively, let's receive the array inside JSX. Assign the details prop in an array to the **`details`** variable. In JSX, use the **`map`** method to loop the details. Since this is a JSX expression, we should enclose the code in curly braces. Get the detail and index in the function parameter, like so:

```
const Animal = props => {
  const details = props.details;
  return (
    <div>
      {props.children}
      <ul>
        {details.map((detail, index) => (
…
        ))}
      </ul>
    </div>
  );
};
```

8. Construct JSX with the **`<li>`**, **`<div>`**, and **`<p>`** elements. Add values such as **`name`**, **`number`**, and **`endangered`**. Also, add a key with **`{index}`**. The complete code will look as follows:

```
const Animal = props => {
  const details = props.details;
  return (
    <div>
      {props.children}
      <ul>
        {details.map((detail, index) => (
          <li key={index}>
            <div>
              <p>Animal: {detail.name}</p>
              <p>Number: {detail.number}</p>
              <p>Endangered: {detail.endangered ? 'Yes' : 'No'}</p>
            </div>
```

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

The output is as follows:

Figure 7.10: Output of the app

In this exercise, we learned how an array can be passed from a parent component and received as a prop in the child components.

PASSING DATA TO A CHILD COMPONENT MULTIPLE LEVELS DOWN

So far, we have learned how to send data to a direct child component and display the data in the UI. In this section, we are going to learn how to send data to child components located multiple levels down. As we mentioned earlier, in React, it is recommended to split your components into smaller pieces to make the components reusable. When we have several hierarchical levels of components, quite often, we need to send data through nested components, as shown in the following diagram:

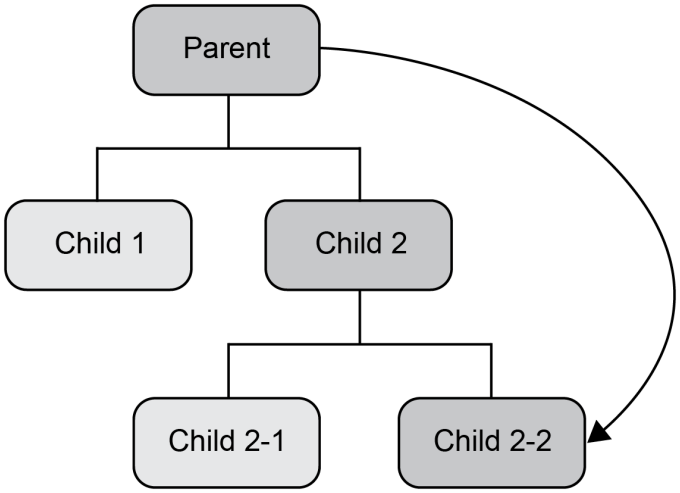

Figure 7.11: Data passed from the parent to its child components

There are several ways we can send data down to such child components:

- **Props**: As we practiced in the previous section, we can keep passing the props down to the child components until the last child component in the hierarchy. In this section, we are going to learn more about how to pass props through multiple levels of child components.
- **React Context API**: The main problem with passing props down to each child component is there could be a lot of layers of components in-between the data source and the user. This is called props-drilling. For example, if there are 10 child components inside a parent component and we want to send data to the tenth child component from the parent component, we need to pass the data 10 times through the other child components. With the React Context API, we can provide the data from the parent component and consume it directly from a component on any level without passing it through the other child components. We will learn how to use the Context API in the following section.
- **Redux**: Redux is one of the most powerful state management libraries. It is commonly used with React, but it can be used standalone with other JavaScript libraries. With Redux, we can centralize the state by storing it in an object tree within a store so each component can access any state from the store.

In this book, we are not going to discuss Redux but if you want to learn more about it, please check the getting started guide (https://redux.js.org/introduction/getting-started) on the Redux documentation page.

Before we jump into any of these ways to pass data from a parent to a child at any other level, it will be useful to understand how to split a component into further smaller, reusable components. We will see how to do that in the following section.

SPLITTING A COMPONENT INTO SMALLER COMPONENTS

To understand how to split a component into smaller, reusable ones, we'll be splitting the Animal component from our previous examples into smaller components and passing the data to the child components:

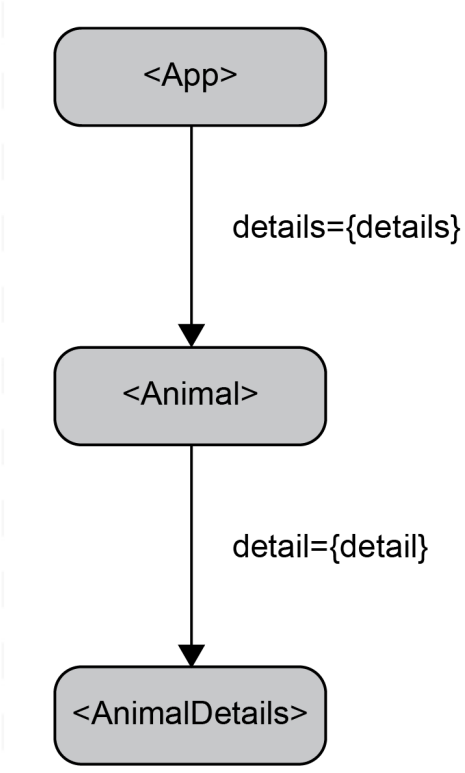

Figure 7.12: Passing data down to multiple child components

In the following exercises, we are going to separate a portion of the code from the Animal component and add the code to a new child component called **`AnimalDetails`**, as shown in the preceding diagram. Let's practice how to split the component by taking a look at the following exercise.

EXERCISE 7.04: SPLITTING INTO SMALLER COMPONENTS

In this exercise, we are going to separate the Animal component and move the animal details to the **`AnimalDetails`** component. To do so, let's go through the following steps:

1. Create a new function component called **`AnimalDetails`**. Then, copy the **`<li>`** elements from the **`Animal`** component and paste them into the return function:

```
const AnimalDetails = props => {

  return (
    <li key={index}>
      <div>
        <p>Animal: {detail.name}</p>
        <p>Number: {detail.number}</p>
        <p>Endangered: {detail.endangered ? 'Yes' : 'No'}</p>
      </div>
    </li>
  );
};
```

2. Above the **`return`** function, destructure the props:

```
const {name, number, endangered} = props.detail;
```

3. Update the **`props`** value in the **`return`** function accordingly:

```
const AnimalDetails = props => {
  const {name, number, endangered} = props.detail;
  return (
    <li key={props.key}>
      <div>
        <p>Animal: {name}</p>
        <p>Number: {number}</p>
        <p>Endangered: {endangered ? 'Yes' : 'No'}</p>
      </div>
    </li>
  );
};
```

4. Now, let's update the **`Animal`** component. Inside the map method in the **`Animal`** component, add the **`<AnimalDetails>`** component. Add the detail and index props to the **`<AnimalDetails>`** component:

```
const Animal = props => {
  const details = props.details;

  return (
    <div>
      {props.children}
      <ul>
        {details.map((detail, index) => (
          <AnimalDetails detail={detail} key={index} />
        ))}
      </ul>
    </div>
  );
};
```

   The output of this will be as follows and will be similar to what we mentioned earlier:

```
// Animal: Tiger
// Number: 3890
// Endangered: Yes
```

As you can see, first, we passed the details data through the details prop to the **`<Animal>`** component and the detail prop to **`<AnimalDetails>`**.

PASSING A COMPONENT THROUGH A PROP

So far, we have learned about passing props of various types, such as string, number, Boolean, array, and object. In the following sections, we are going to discuss sending components and functions through props. First, let's learn how to send a component.

Passing a component as a prop to a child component from a parent component allows us to reuse components and directly display the React elements with updated values.

> **NOTE**
>
> In this section, we are using Unsplash (https://unsplash.com/), which is where you can download free photos and use them for any project.

We are going to put the **`details`** data, including the animal's name, number, endangered status, and photo, in the App component and make use of this data in the following sections:

```
// the details data located in the App component
const details = [
      {
        name: 'Tiger',
        number: 3890,
        endangered: true,
        photo: 'https://source.unsplash.com/Si6Obte6Bu0/200x100'
      },
      {
        name: 'Brown Bear',
        number: 200000,
        endangered: false,
        photo: 'https://source.unsplash.com/c8XlAc1akIU/200x100'
      },
      {
        name: 'Red Panda',
        number: 10000,
        endangered: true,
        photo: 'https://source.unsplash.com/2zYHKx8jtvU/200x100'
      }
    ];
```

To display the photo of the animal, we are going to create a new component called <Photo> and reuse this component by sending it through a prop. The following exercise will teach us how we can create a new functional component.

EXERCISE 7.05: CREATING A PHOTO FUNCTION COMPONENT

In this exercise, we are going to create a child function component called **`Photo`** and pass that as a component through the props from the parent component, **`<Animal>`**, to the child component, **`<AnimalDetails>`**. To do so, let's go through the following steps:

1. Create a **`<Photo>`** function component and return an **`<img>`** element. We are also going to receive two props, path for the **`src`** attribute and name for the alt attribute:

```
const Photo = props => {
  return <img src={props.path} alt={props.name} />;
};
```

2. Let's pass the **`<Photo>`** component to the **`<AnimalDetails>`** component.

3. For **`<AnimalDetails>`** in the **`<Animal>`** component, add a new prop called **`image`** with the value **`<Photo path={detail.photo} title={detail.name} />.`** Make sure to enclose it in curly braces:

```
const Animal = props => {
  const details = props.details;
…
  return (
    <div>
    {props.children}
    <ul>
      {details.map((detail, index) =>
…
      <AnimalDetails image={<Photo path={detail.photo} title={detail.
name} />}
    detail={detail} key={index} />
      )
```

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

4. Now, let's receive the **<Photo>** component from the **<AnimalDetails>** component.
5. Right below the opening ** **, add **{props.image}**. This will display the **<Photo>** components with the props value we sent through, such as the photo path and title:

```
const AnimalDetails = props => {
  const { name, number, endangered } = props.detail;

  return (
    <li key={props.key}>
      <div>
        <p>{props.image}</p>
        <p>Animal: {name}</p>
        <p>Number: {number}</p>
        <p>Endangered: {endangered ? 'Yes' : 'No'}</p>
      </div>
    </li>
  );
};
```

The output of the preceding code should appear as follows:

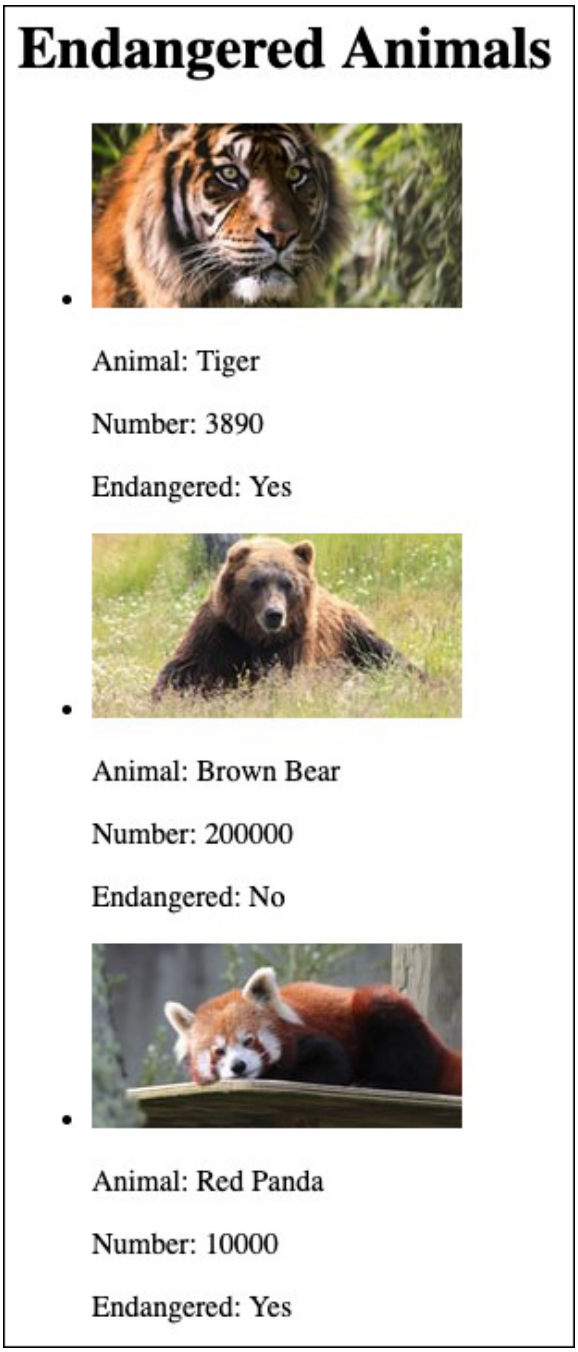

Figure 7.13: Photo function component

As we saw in this exercise, our Endangered Animals app now contains all the necessary information to function, including the Photo component. The Photo component was passed as a prop from the parent to the child component. In the next section, we are going to learn about a more advanced React concept known as Higher-Order Components (HOC).

HIGHER-ORDER COMPONENTS

According to the React documentation (https://reactjs.org/), "... a higher-order component is a function that takes a component and returns a new component". A HOC is not a React API, it is a pattern that shares common functionalities. From the name itself, you may think it is some sort of component; however, it is actually a function. A HOC function accepts a component as an argument and returns a new component. HOCs help us reuse code with the same functionalities between components so that we do not have to repeat the same code.

To see HOCs in action, we are going to add a new feature to our **`Endangered Animals`** app that we created previously. We are going to add another property in the details data called **`Donation`** and if the amount of a donation is bigger than **`50`**, we will update the color of the donation to green. Otherwise, we will keep the donation color red.

Before we start the exercise, let's update the details object we used in the *Passing a Component* through a Prop section and add the donation property with a number for each animal:

```
const details = [
      {
        name: 'Tiger',
        number: 3890,
        endangered: true,
        photo: 'https://source.unsplash.com/Si6Obte6Bu0/400x300',
        donation: 100
      },
      {
        name: 'Brown Bear',
        number: 200000,
        endangered: false,
        photo: 'https://source.unsplash.com/c8XlAc1akIU/400x300',
        donation: 10
      },
      {
```

```
        name: 'Red Panda',
        number: 10000,
        endangered: true,
        photo: 'https://source.unsplash.com/2zYHKx8jtvU/400x300',
        donation: 50
      }
    ];
```

To iterate through the **`details`** array, inside the **`map`** method, instead of calling the **`AnimalDetails`** component, we are going to call a new component called **`WrapperComponent`**:

```
const Animal = props => {
  const details = props.details;
  return (
    <div>
      {props.children}
      <ul>
        {details.map((detail, index) => (
          <WrapperComponent image={<Photo path={detail.photo}
title={detail.name}
    />} detail={detail} key={index} />
        ))}
      </ul>
    </div>
  );
};
```

Now, we are going to create another component by calling the HOC function. Here, we are going to create the **`AnimalDetails`** component as an argument, as follows:

```
const WrapperComponent = withDonationColor(AnimalDetails);
```

By doing so, in the HOC, we can access all the props that were sent to the **`AnimalDetails`** component in the HOC function.

EXERCISE 7.06: CREATING A HOC FUNCTION THAT CAN BE CALLED WITH DONATIONCOLOR

In this exercise, we are going to create a HOC function that will check if the amount of the donation is greater than **`50`**. If it's greater than **`50`**, we will change the text color of the donation amount to green; otherwise, it will stay red.

> **NOTE**
>
> Before you complete this exercise, ensure that you have modified the code so that it looks like what we had in the previous section.

1. Create a HOC function called **`withDonationColor`** that receives a component as an argument called **`WrappedComponent`**. Inside the function, we are going to create a class-based component and directly return it:

```
const withDonationColor = WrappedComponent => {
  return class extends Component {
  }
};
```

2. Inside the class component, add a constructor method and pass the props. Add a **`super()`** keyword while passing the props to it and initialize the state with the donation color set to black:

```
const withDonationColor = WrappedComponent => {
  return class extends Component {
    constructor(props) {
      super(props);
      this.state = { donationColor: 'black' };
    }
  }
};
```

3. Under the constructor method, create the **`componentDidMount`** method. Inside the **`componentDidMount`** life cycle, we are going to get the amount of the donation, followed by checking whether it's bigger than **`50`**. If it's bigger than **`50`**, update the **`donationColor`** state to green; otherwise, update it to red:

```
const withDonationColor = WrappedComponent => {
  return class extends Component {
    constructor(props) {
      super(props);
      this.state = { donationColor: 'black' };
    }
…
    componentDidMount() {
      const donationAmout = this.props.detail.donation;
      const donationColor = donationAmout > 50 ? 'green' : 'red';
      this.setState({ donationColor });
    }
  }
};
```

4. Lastly, add a **`render`** method. In the **`render`** method, we are going to return the **`WrappedComponent`** component we received as an argument, but with a new prop called **`donationColor`**. We will still apply the remaining props with the spread attributes we learned about in the previous section:

```
const withDonationColor = WrappedComponent => {
  return class extends Component {
    constructor(props) {
      super(props);
      this.state = { donationColor: 'black' };
    }

    componentDidMount() {
      const donationAmout = this.props.detail.donation;
      const donationColor = donationAmout > 50 ? 'green': 'red';
      this.setState({ donationColor });
    }

    render() {
```

```
        return <WrappedComponent {...this.props}
      donationColor={this.state.donationColor} />;
      }
    };
};
```

5. Now, in the **`AnimalDetails`** component, we can access the **`donationColor`** prop that we sent from the HOC function.
6. Now, let's add **`donationColor`**. Add an inline style to the donation amount, along with the amount. Please make sure to add two sets of curly braces. The first set of curly braces (outside) will be for adding a JavaScript expression, while the second set of curly braces (inside) will be for the inline style (https://reactjs.org/docs/dom-elements.html#style):

```
const AnimalDetails = props => {
  const { name, number, endangered, donation} = props.detail;

  return (
    <li key={props.index}>
      <div>
        <p>{props.image}</p>
        <p>Animal: {name}</p>
        <p>Number: {number}</p>
        <p>Endangered: {endangered ? 'Yes' : 'No'}</p>
        <p style={{ color: props.donationColor }}>Donation amount:
    ${donation}</p>
      </div>
    </li>
  );
};
```

The output of the preceding code should appear as follows:

Figure 7.14: Output of the donation amount (in color)

In this exercise, we practiced how to update the color of the animal depending on the amount of the donation. From here, you can make the HOC function more reusable by separating the color and amount.

RENDER PROPS

In the previous section, we learned what a HOC is and how to use it. In this section, we are going to learn about one more advanced React concept, render props.

Like HOCs, render props help us reuse the code between components and help us avoid repeating the same code. On the other hand, unlike HOCs, which take a component and return an updated component, render props take a function and return a React element. In the next exercise, we are going to do exactly the same as what we did for HOCs, that is, we'll update the color of the donation amount if the amount is bigger than 50.

The details object we mentioned previously will be the same and we are going to update it from the Animal component.

EXERCISE 7.07: ADDING DONATIONCOLOR

In this exercise, similar to what we did for the HOC function exercise, we are going to create a render prop function and check if the amount of the donation is greater than **`50`**. If it's greater than **`50`**, we will change the text color of the donation amount to green; otherwise, we'll change it to red. Follow these steps to complete this exercise:

As with the HOC, create a new component called **`WrapperComponent`**, but this time for render props. Here, we are going to add a render prop to **`<WrapperComponent>`** and send a function.

1. Inside the map method, add **`<WrapperComponent>`**. Also, add the **`donationAmount`** and render props to **`<WrapperComponent>`**:

```
const Animal = props => {
  const details = props.details;

  return (
    <div>
      {props.children}
      <ul>
        {details.map((detail, index) => (
          <WrapperComponent
            key={index}
            donationAmount={detail.donation}
            render={() => (

            )}
```

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

2. In the **`render`** prop, we are going to pass a function. Since we want to receive the **`donation color`**, add **`donationColor`** as a parameter. Make sure to destructure the object as we only want to receive the **`donationColor`** parameter from the object:

```
const Animal = props => {
  const details = props.details;

  return (
    <div>
      {props.children}
      <ul>
        {details.map((detail, index) => (
          <WrapperComponent
            key={index}
            donationAmount={detail.donation}
            render={({ donationColor }) => (

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

3. Add the **`<AnimalDetails>`** component in the function alongside the **`donationColor`** parameter we received previously, along with the image, detail, and index props:

```
const Animal = props => {
  const details = props.details;

  return (
    <div>
      {props.children}
      <ul>
        {details.map((detail, index) => (
          <WrapperComponent
            key={index}
            donationAmount={detail.donation}
            render={({ donationColor }) => (
              <AnimalDetails
                donationColor={donationColor}
                image={<Photo path={detail.photo} title={detail.name}
/>}
                detail={detail}
                key={index}
              />
            )}
          />
        ))}
      </ul>
    </div>
  );
};
```

4. Now, let's create the **<WrapperComponent>**. The **WrapperComponent** will have the logic that will decide the color of the **donationColor** and return an object containing the **donationColor**. First, add a constructor method and call **super(props)** . Also, initialize the state of the **donationColor** parameter with black as a default value:

```
class WrapperComponent extends Component {
  constructor(props) {
    super(props);
    this.state = { donationColor: 'black' };
  }
}
```

5. Add the **componentDidMount()** life cycle method. When the component is mounted, we are going to check if the donation amount is bigger than **50** and if so, update the **donationColor** state to green; otherwise, we will update it to red:

```
class WrapperComponent extends Component {
  constructor(props) {
    super(props);
    this.state = { donationColor: 'black' };
  }

  componentDidMount() {
    const donationAmout = this.props.donationAmount;
    const donationColor = donationAmout > 50 ? 'green' : 'red';
    this.setState({ donationColor });
  }

}
```

6. Add the **render()** method. In the **render** method, return an object with **donationColor** as a property:

```
class WrapperComponent extends Component {
  constructor(props) {
    super(props);
    this.state = { donationColor: 'black' };
  }

  componentDidMount() {
```

```
    const donationAmout = this.props.donationAmount;
    const donationColor = donationAmout > 50 ? 'green' : 'red';
    this.setState({ donationColor });
  }

  render() {
    return this.props.render({
      donationColor: this.state.donationColor
    });
  }
}
```

7. Lastly, in the **<AnimalDetails>** component, we are going to update the **donation** color. Add an inline style with the **donationColor** parameter for the **donation** color:

```
const AnimalDetails = props => {
  const { name, number, endangered, donation } = props.detail;

  return (
    <li key={props.index}>
      <div>
        <p>{props.image}</p>
        <p>Animal: {name}</p>
        <p>Number: {number}</p>
        <p>Endangered: {endangered ? 'Yes' : 'No'}</p>
        <p style={{ color: props.donationColor }}>Donation amount:
  ${donation}</p>
      </div>
    </li>
  );
};
```

The output from the render props we created here should display the donation amount text in either green or red, as shown in the following screenshot:

Figure 7.15: App showing the donation amount

As we can see, the **`donationColor`** details, which were created as a **`WrapperComponent`**, were added to each Animal component.

So far, we have learned how to use HOCs and render props. Both HOCs and render props allow us to share the same functionality between components. There is no hard and fast rule of which one to use over the other. However, as a general rule of thumb, go with render props first. This is because render props require less boilerplate code and are easier to set up. Apart from this, they are more predictable when debugging with state or props. However, when you find too many nested render props (similar to callback hell), go with small HOCs and compose them together.

PASSING DATA FROM CHILDREN TO A PARENT

As we have already mentioned, data can only be passed down to a child component from the parent component. However, there could be instances where you might need to send data from the child components to the parent components:

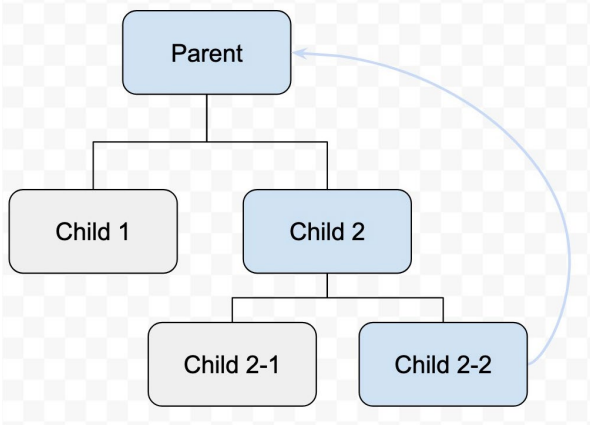

Figure 7.16: Communicating from children to a parent

In this section, we are going to continue with the Endangered Animals app from *Exercise 7.07*, *Adding donationColor* ,but this time, we are going to add a **`remove`** button to each animal section and when we click the **`remove`** button, we will remove the animal from the list. The outcome of this should look something like the following:

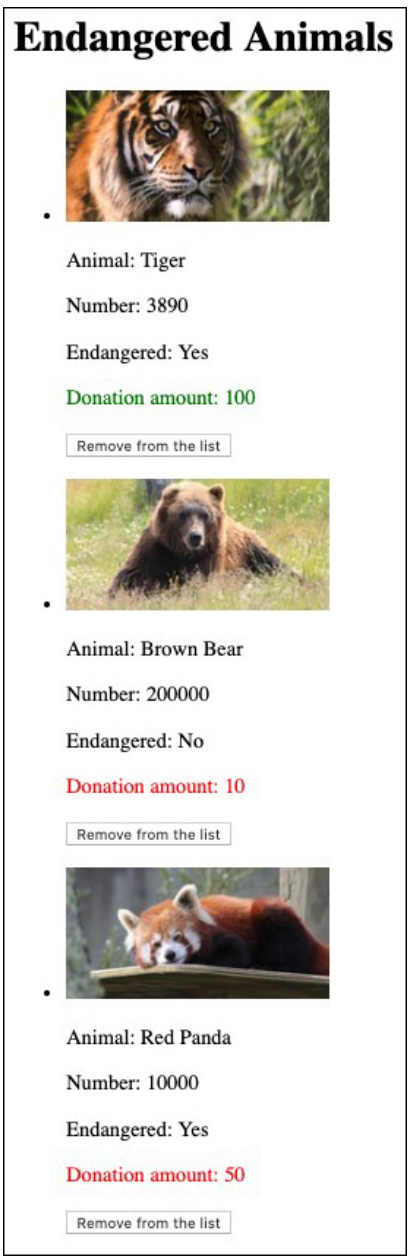

Figure 7.17: Outcome of the remove button

The overall plan for this functionality is as follows:

- In the **<App>** component, we are going to update the details object to a state so that when we remove the details in the state, it will re-render the components and display the updated UI.
- In the **<App>** component, we are going to create a new function called **removeList**. This function will take an id of the details object as an argument and have the logic to update the state by removing the object matching the ID.
- In the **<App>** component, we are going to send the **removeList** callback function through a prop.
- In the **<Animal>** component, we are going to send the **removeList** prop to **<WrapperComponent>**.
- In the **<AnimalDetails>** component, we are going to reference the **removeList** callback function in the **onClick** event on the button element.

We are going to execute this in the form of steps of an exercise.

EXERCISE 7.08: PASSING DATA FROM A CHILD TO A PARENT COMPONENT

In this exercise, we will pass data from a child to a parent component using a callback function that will remove the **animal** section if we click the remove button in the child component. Follow these steps to complete this exercise:

1. At the top of the **<App>** component, create a constructor method and pass props as an argument. Inside the constructor method, add a **super()** keyword while passing props to it. Move the details object from the render method to **this.state** under the **super** keyword:

App.js

```
3  class App extends Component {
4  constructor(props) {
5    super(props);
6
7    this.state = {
8      details: [
9        {
10          id: '1',
11          name: 'Tiger',
12          number: 3890,
13          endangered: true,
14          photo: 'https://source.unsplash.com/Si6Obte6Bu0/200x100',
15          donation: 100
```

The complete code can be found here: https://packt.live/3dKxyMV.

2. In the **<App>** component, create the **removeList** function under the constructor method. Accept **id** as an argument. The **id** is the **id** of the animal in the **details** state. Call **this.setState** with the functional approach. We will filter out by matching the ID from the **details** state and return a new state:

App.js

```
3 class App extends Component {
4  constructor(props) {
5    super(props);
…
37  removeList(id) {
38    this.setState(prevState => {
39      const list = prevState.details.filter(item => item.id !== id);
40      return { prevState, details: list };
41    });
42  }
44  render() {
45    return (
46      <Animal details={this.state.details}
    removeList={this.removeList.bind(this)}>
```

The complete code can be found here: https://packt.live/3dVNgVJ.

3. In the **render()** method, pass the **removeList** function through the **removeList** prop in the **<Animal>** component. Make sure to **bind(this)** to make **this** keyword work in the callback:

```
    render() {
      return (
        <Animal details={this.state.details}
      removeList={this.removeList.bind(this)}>
            <h1>Endangered Animals</h1>
        </Animal>
      );
    }
  }
```

4. In the **<Animal>** component, we are going to receive the **removeList** prop as **props.removeList**. Pass it through the **removeList** prop in **<WrapperComponent>**:

```
  const Animal = props => {
    const details = props.details;

    return (
      <div>
        {props.children}
        <ul>
```

```
          {details.map((detail, index) => (
            <WrapperComponent
              key={index}
              image={<Photo path={detail.photo} title={detail.name} />}
              detail={detail}
              index={index}
              removeList={props.removeList}
            />
          ))}
        </ul>
      </div>
    );
};
```

5. In the **<AnimalDetails>** component, add a **<button>** element with the text, **Remove from list**. In the **<button>** element, add an **onClick** event and a reference to the **removeList** callback function. Make sure to send the id of the animal as an argument:

```
const AnimalDetails = props => {
  const { id, name, number, endangered, donation } = props.detail;

  return (
    <li key={id}>
      <div>
        <p>{props.image}</p>
        <p>Animal: {name}</p>
        <p>Number: {number}</p>
        <p>Endangered: {endangered ? 'Yes' : 'No'}</p>
        <p style={{ color: props.donationColor }}>Donation amount:
  {donation}</p>
        <button onClick={() => props.removeList(id)}>Remove from the
  list</button>
      </div>
    </li>
  );
};
```

The output is as follows:

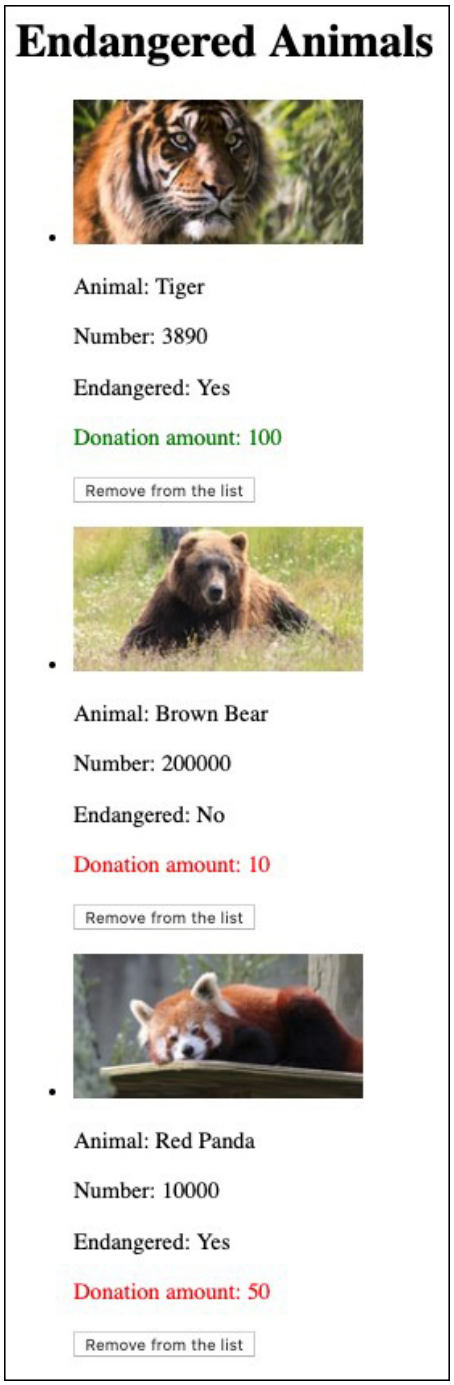

Figure 7.18: Output

As you can see, when you click on the remove from the list button, the **`animal`** section should be removed from the list. This is how you can communicate with the parent component from the child component.

PASSING DATA BETWEEN COMPONENTS AT ANY LEVEL

In the previous sections, we learned how to communicate between parent components and child components. In this section, we are going to learn how we can pass data between components at any level.

A good example of this is an e-commerce site. When you add a product to a shopping cart, by clicking the **`Add to cart`** button, it will update the number of products in the cart. In this case, we may have a product details component under the product component, whereas the cart component would be located in a totally different hierarchical level to the product details. In the following exercises, we will use this concept and we will learn how to send data from a component to a component in a different hierarchical tree.

Let's look at the diagram for our case:

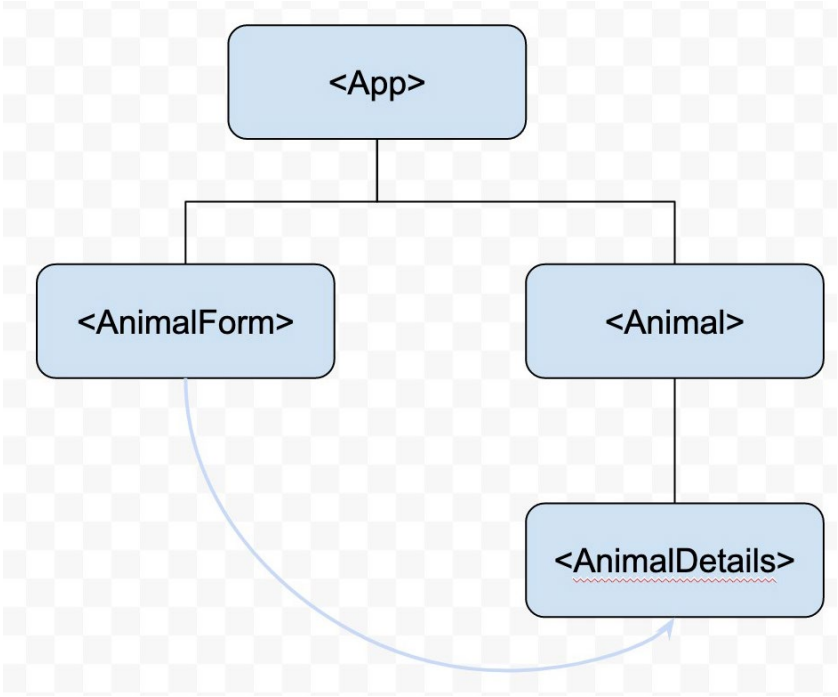

Figure 7.19: Passing data between any components

In this section, we are going to create new component called **`<AnimalForm>`**. The **`<AnimalForm>`** component will have a form where people can submit new animal details. When people submit the form with the new details, the newly added animal details will appear through the **`<AnimalDetails>`** component:

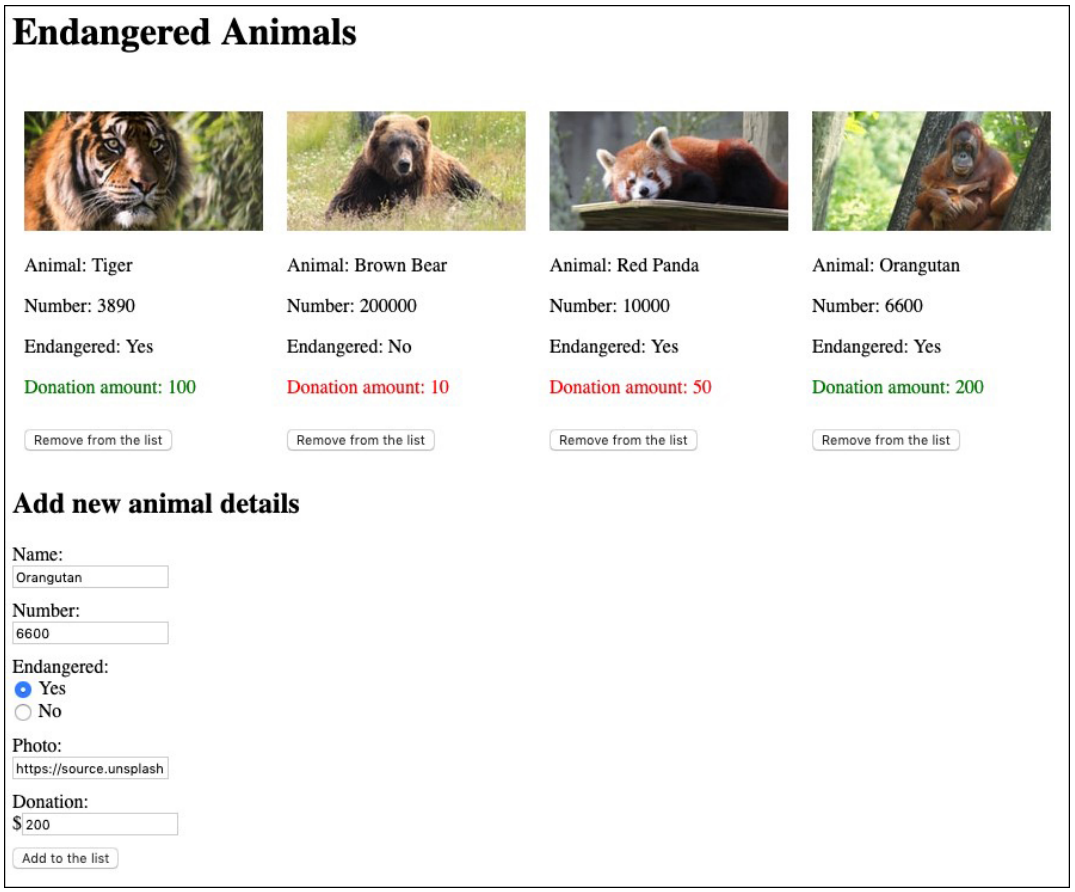

Figure 7.20: Output of the <AnimalForm> component

First, let's add some simple styles so that we can lay out the list of the animals better. We will create a new file called **`styles.css`** in the **`src`** folder. Right below this, we will add an import statement for React at the top of the page, import a **`styles.css`**. Then, add the styles like so:

```
label {
  display: block;
  min-width: 50px;
}

.title {
  margin-top: 10px;
}

button {
  margin-top: 10px;
}

.list {
  display: flex;
  list-style: none;
  margin: 0 0 20px;
  padding: 0;
}

.list li {
  margin: 10px;
}
```

Let's start off with some exercises to learn how we can pass data between any component. In the first exercise, we will create a callback function and then in the next exercise, we will pass that callback function through a prop so that we can pass data between any component, irrespective of the hierarchy.

EXERCISE 7.09: ADDING THE ADDLIST CALLBACK FUNCTION

In this exercise, we will create a callback function called **addList**, within which we are going to update the state. Follow these steps to complete this exercise:

1. In the **<App>** component, add a new function called **addList** and receive the details parameter. Inside the function, we are going to update the state. Add **this.state** and receive **prevState**. In the **setState** function, first, get a new ID by incrementing one from the total number of the details array (**prevState.details.length + 1**) and create a **newDetails** object by adding the new ID. Finally, return the new state with the **newDetails** added:

```
addList(details) {
    this.setState(prevState => {
      const newId = prevState.details.length + 1;
      const newDetails = { ...details, id: newId };
      return { ...prevState, details: [...prevState.details,
newDetails] };
    });
  }
```

2. To use **addList** as a callback function, we need to send it through a prop. In the render method, add the **<AnimalForm>** component, add a new prop called **addList**, and send the callback function we just created:

```
render() {
    return (
      <React.Fragment>
        <Animal details={this.state.details}
    removeList={this.removeList.bind(this)}>
          <h1>Endangered Animals</h1>
        </Animal>
        <AnimalForm addList={this.addList.bind(this)} />
      </React.Fragment>
    );

}
```

So far, we have prepared the callback function and sent it through the **addList** prop to the **AnimalForm** component. This callback function will allow us to add a new animal list.

3. Now, let's create the **<AnimalForm>** component. The **<AnimalForm>** component will contain several functions and JSX elements so that we can output the form with a submit button. We will pass the **addList** callback function as a prop and pass a value from one component to the other.
4. Create a class-based component called **AnimalForm**.
5. In the constructor method, initialize a state. This technique is called *Controlled* component (more on that in *Chapter 8*, *Introduction to Formik*):

```
        this.state = {
          name: '',
          number: 0,
          endangered: false,
          photo: '',
          donation: 0
        };
```

6. Add a **render** method and return the following JSX element to display a form:

App.js

```
115 render() {
116    return (
117      <form onSubmit={this.handleSubmit}>
118        <h2>Add new animal details</h2>
119        <label>
120          <div className="title">Name:</div>
121          <input type="text" name="name" />
122        </label>
123        <label>
149          <div className="title">Photo:</div>
150          <input type="text" name="photo" />
151        </label>
```

The complete code can be found here: https://packt.live/3cx9Vas.

7. For each **input** element, add a value from the state. Each property name from the state we initialized in the constructor method will be the same as the name of the input element:

App.js

```
render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <h2>Add new animal details</h2>
        <label>
          <div className="title">Name:</div>
          <input type="text" value={this.state.name} name="name" />
        </label>
        <label>
          <div className="title">Number:</div>
          <input type="number" value={this.state.number} name="number" />
```

The complete code can be found here: https://packt.live/3cx9Vas.

8. Let's update the value of each input element when the value gets changed. To do this, we are going to create a new function called **handleChange** and reference to it in the **onChange** event. First, add the **onChange** event and add a reference to the **handleChange** function:

App.js

```
115  render() {
116    return (
117      <form onSubmit={this.handleSubmit}>
118        <h2>Add new animal details</h2>
119        <label>
120          <div className="title">Name:</div>
121          <input type="text" value={this.state.name} name="name"
    onChange={this.handleChange} />
122        </label>
123        <label>
129          <div className="title">Number:</div>
130          <input type="number" value={this.state.number} 131name="number"
    onChange={this.handleChange} />
```

The complete code can be found here: https://packt.live/2Asns4Z.

9. For the **onChange** event, we need to bind **this** keyword. Add it to the constructor method:

```
    constructor(props) {
        super(props);

        this.state = {
          name: '',
          number: 0,
          endangered: false,
```

```
        photo: '',
        donation: 0
      };

      this.handleChange = this.handleChange.bind(this) ;
  }
```

10. Create the **`handleChange`** function and receive the event.
11. First, assign **`event.target`** to **`inputTarget`**. By doing this, we are going to get the value from the target. Since we have two different input types – text (including number) and radio – we are going to create another function called **`getInputValue`** and return the value in the next step. On the next line, assign the target name.
12. Finally, update the state with the input name as a property with the value:

```
  handleChange(event) {
      const inputTarget = event.target;
      const inputValue = this.getInputValue(inputTarget);
      const inputName = inputTarget.name;

      this.setState({ [inputName]: inputValue });
    }
```

13. Create the **`getInputValue`** function. We are going to receive a target and check the type and value and return the proper input value to update in the state:

```
  getInputValue(target) {
      if (target.type === 'radio' && target.value === 'yes') {
        return true;
      } else if (target.type === 'radio' && target.value === 'no') {
        return false;
      }

      return target.value;
    }
```

14. Now, we need to handle the **`submit`** event. So, when people submit the form, we are going to send the details from the state through the **`addList`** callback function.

15. First, add a **`handleSubmit`** reference in the form's **`onSubmit`** event:

```
render() {
  return (
  <form onSubmit={this.handleSubmit}>
     <h2>Add new animal details</h2>
     <label>
     ...
```

16. Then, bind **`this`** keyword in the constructor method:

```
constructor(props) {
    super(props);

    this.state = {
      name: '',
      number: 0,
      endangered: false,
      photo: '',
      donation: 0
    };

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }};
```

17. Create the **`handleSubmit`** function and receive event.

    Inside the **`handleSubmit`** function, we will add **`event.preventDefault()`** to prevent a browser reload/refresh when submitting the form. If the browser gets reloaded, our app will lose all the data in the state. On the next line, invoke the **`addList`** callback function by sending the state of the details that were updated by the form elements:

```
handleSubmit(event) {
    event.preventDefault();

    this.props.addList(this.state);
```

The output is as follows:

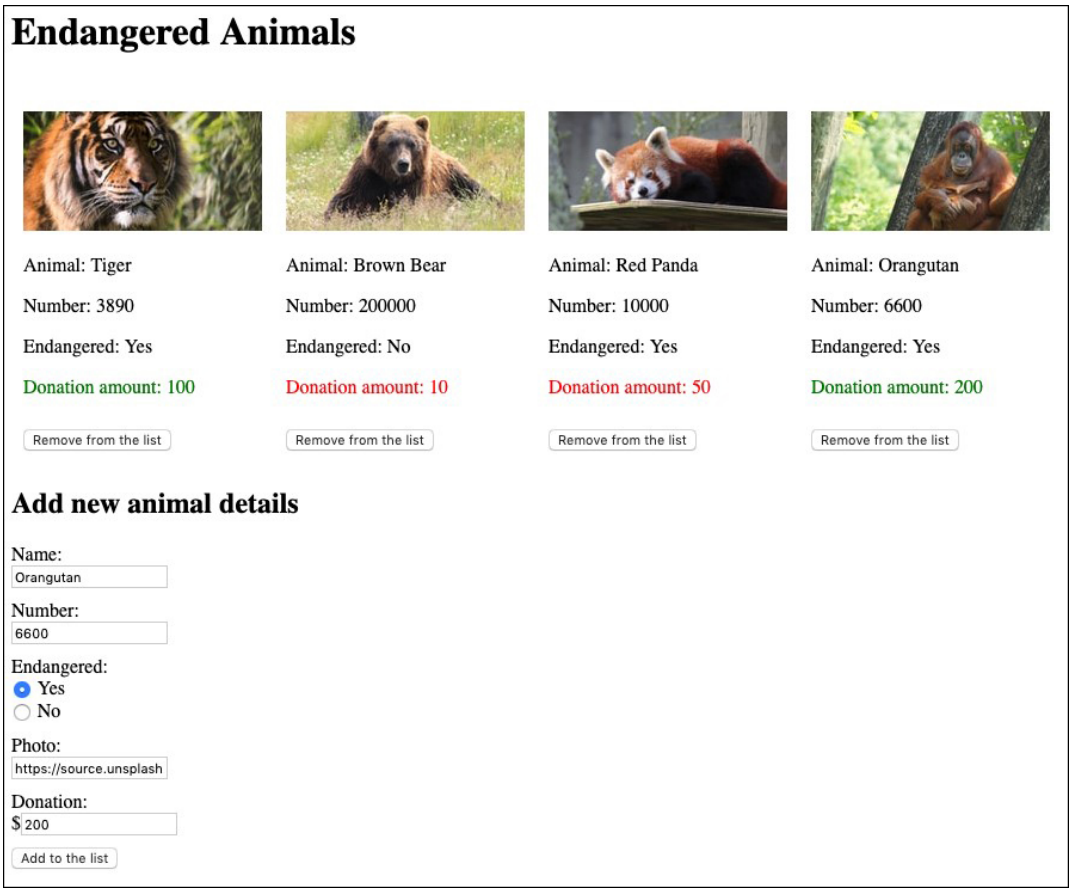

Figure 7.21: Animal details

When we submit new details, the **`handleSubmit`** function will invoke the **`addList`** callback function and it will update the state of the animal details. Once the state is updated, it will re-render the child components and display our newly added animal list. From the preceding output, we can see that we have added new animal details for **`Orangutan`**.

THE CONTEXT API

Throughout this chapter, we have learned how to pass data through props to communicate between components. However, as we had mentioned briefly earlier, we will face the prop-drilling issue while passing data through props.

From the Endangered Animals app we built earlier to pass data from the **`<App>`** component to the **`<AnimalDetails>`** component, we had to pass the data through the intermediate component, which is the **`<Animal>`** component, even if the **`<Animal>`** component does not make use of the prop.

In our app, there is only one intermediate component for now. However, if our app gets more complex, we will have to pass the data through several intermediate components.

So how can we pass the data to the child component directly without going through multiple intermediate components in-between? We could use Redux, which manages the state in one place, called a store, and each component can access its store when needed. However, since this is a React book, we will focus more on the React way and look at the Context API.

The Context API provides us with a way to pass state across multiple levels of components without passing through props. Therefore, we do not have to pass a prop through every layer of the component tree.

To see the Context API in action, we are going to add the total number of endangered animals between the list of endangered animals and the form for adding new animal details. The output of the app should appear as follows:

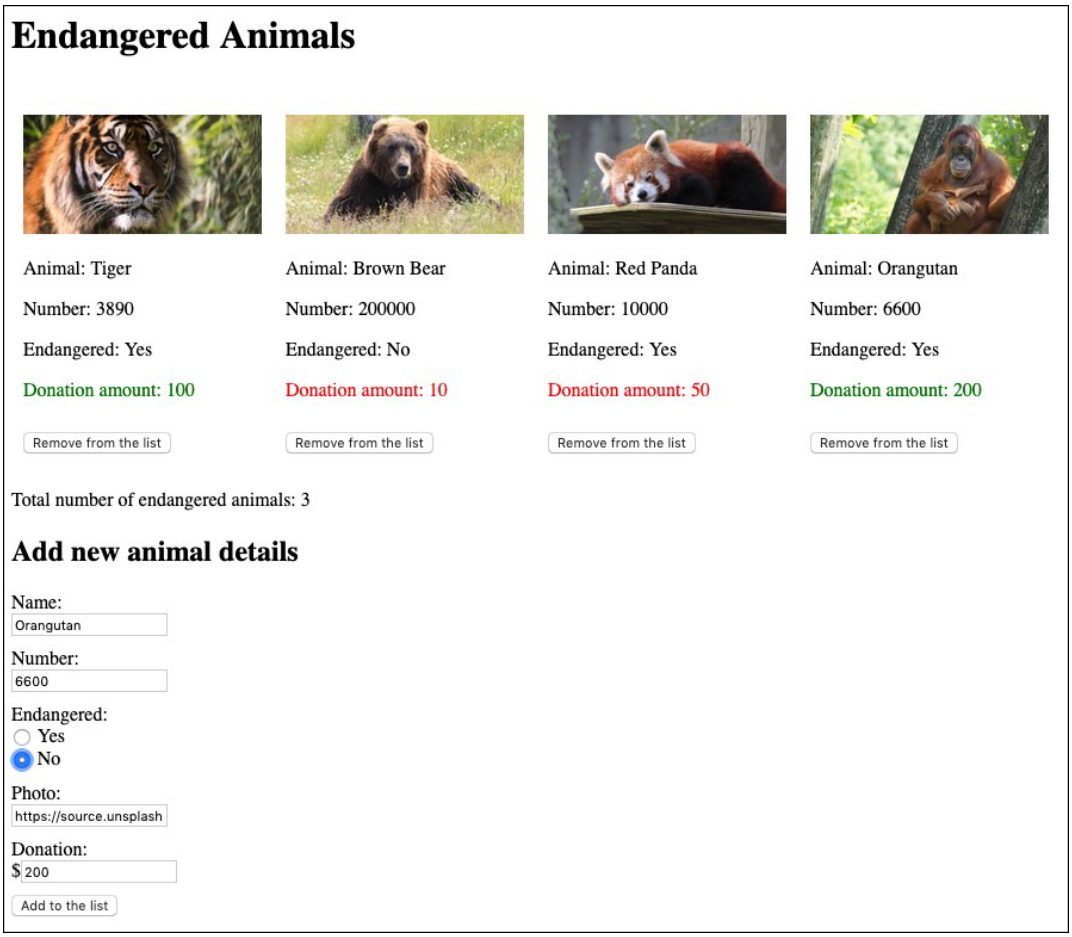

Figure 7.22: The output of the total number of endangered animals

To add the total number of endangered animals, we are going to create one more component called **`AnimalCount`**:

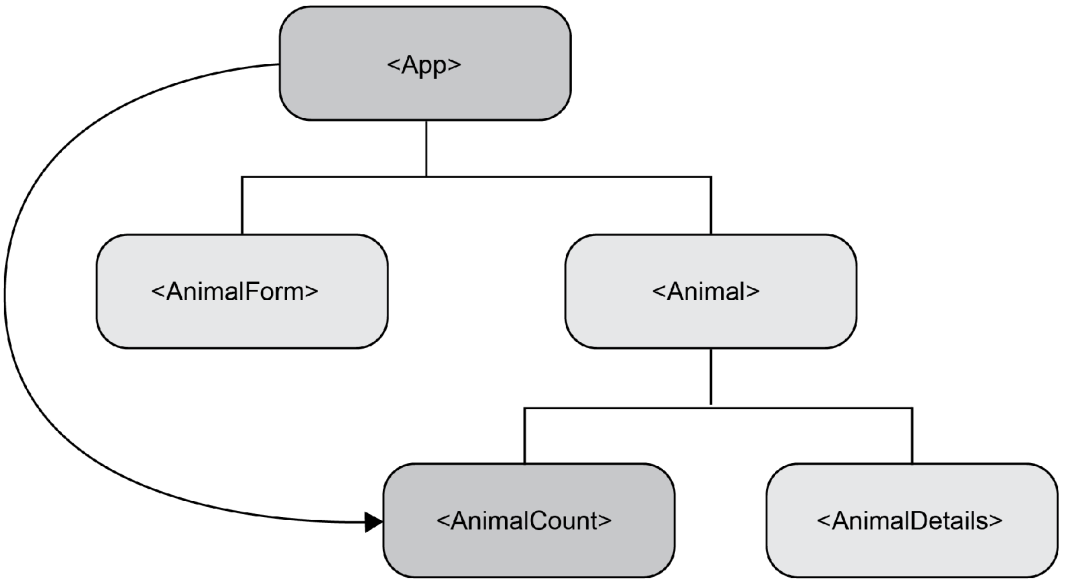

Figure 7.23: New AnimalCount component

The **`<AnimalCount>`** component will sit under the **`<Animal>`** component. The state of the total number will be managed from the **`<App>`** component level, but we will directly receive the total number in the **`<AnimalCount>`** component from the **`<App>`** component without passing through the **`<Animal>`** component.

Moreover, when we add new animal details from the **`<AnimalForm>`** component, the total number of endangered animals will be automatically updated in the **`<AnimalCount>`** component.

EXERCISE 7.10: CREATING THE <ANIMALCOUNT> COMPONENT USING THE REACT CONTEXT API

In this exercise, we will see the Context API in action. We will add another component called **<AnimalCount>** to the **<Animal>** component using the Context API. We will continue to add additional code where we left off in the previous section. Perform these steps to complete this exercise:

1. Create a **Context** object with a default value of the count, zero, above the **<App>** class-based component:

```
const CountContext = React.createContext(0);

class App extends Component {
  constructor(props) {
    super(props) ;
    ...
```

2. Initialize the count to zero in the state in the constructor method of the **<App>** component:

```
const CountContext = React.createContext(0);

class App extends Component {
  constructor(props) {
    super(props);

    this.state = {
      details: [...],
      count: 0
    };
  }
```

3. Create a **Provider** component with the count value from the state, **<CountContext.Provider value={this.state.count}>**, and wrap both of the **<Animal>** and **<AnimalForm>** components in the render method. The **Provider** component should be created and should wrap the parent component so that the consuming components will consume the value:

```
render() {
  return (
    <React.Fragment>
      <CountContext.Provider value={this.state.count}>
```

```
          <Animal details={this.state.details}
      removeList={this.removeList.bind(this)}>
            <h1>Endangered Animals</h1>
          </Animal>
          <AnimalForm addList={this.addList.bind(this)} />
        </CountContext.Provider>
      </React.Fragment>
    );
  }
```

4. Create a new function component to display the total number of endangered animals.

5. Create a **`Consumer`** in the **`return()`**, **`<CountContext.Consumer>`**, and receive the **`props`**. The **`props`** value will contain the count that was sent from the Provider:

```
const AnimalCount = () => {
  return (
    <CountContext.Consumer>
      {props => (
        <div>
          Total number of endangered animals:
          <span> {props}</span>
        </div>
      )}
    </CountContext.Consumer>
  );
};
```

6. Now, if we update the count from the **`App`** component, the total number in **`<AnimalCount>`** will be automatically updated too.

7. Let's update the count when a new list of animal details is added. The count will only go up by one if the new animal is endangered. In the **`<App>`** component, add a function called **`updateCount()`** and update the count state:

```
updateCount() {
  this.setState(prevState => {
    return {
      ...prevState,
      count: this.state.details.filter(item => item.endangered ===
    true).length
    };
```

```
    });
  }
```

8. When a new list of animal details is added, we will call the **`updateCount`** function. In the **`addList`** function, add a callback of **`setState`** and add the **`updateCount()`**. This way, we can invoke the **`updateCount`** function after the state gets updated and we can update the count number properly:

```
addList(details) {
  this.setState(
    prevState => {
      const newId = prevState.details.length + 1;
      const newDetails = { ...details, id: newId };
      return { ...prevState, details: [...prevState.details,
newDetails] };
    },
    () => {
      this.updateCount();
    }
  );
}
```

9. Finally, we also need to initially update the count number when the app is loaded. The best way to invoke the **`updateCount`** function is in the **`componentDidMount`** life cycle:

```
componentDidMount() {
  this.updateCount();
}
```

10. Finally, let's add the **`AnimalCount`** component to the Animal component:

```
const Animal = props => {
  const details = props.details;

  return (
  <div>
...

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

The output is as follows:

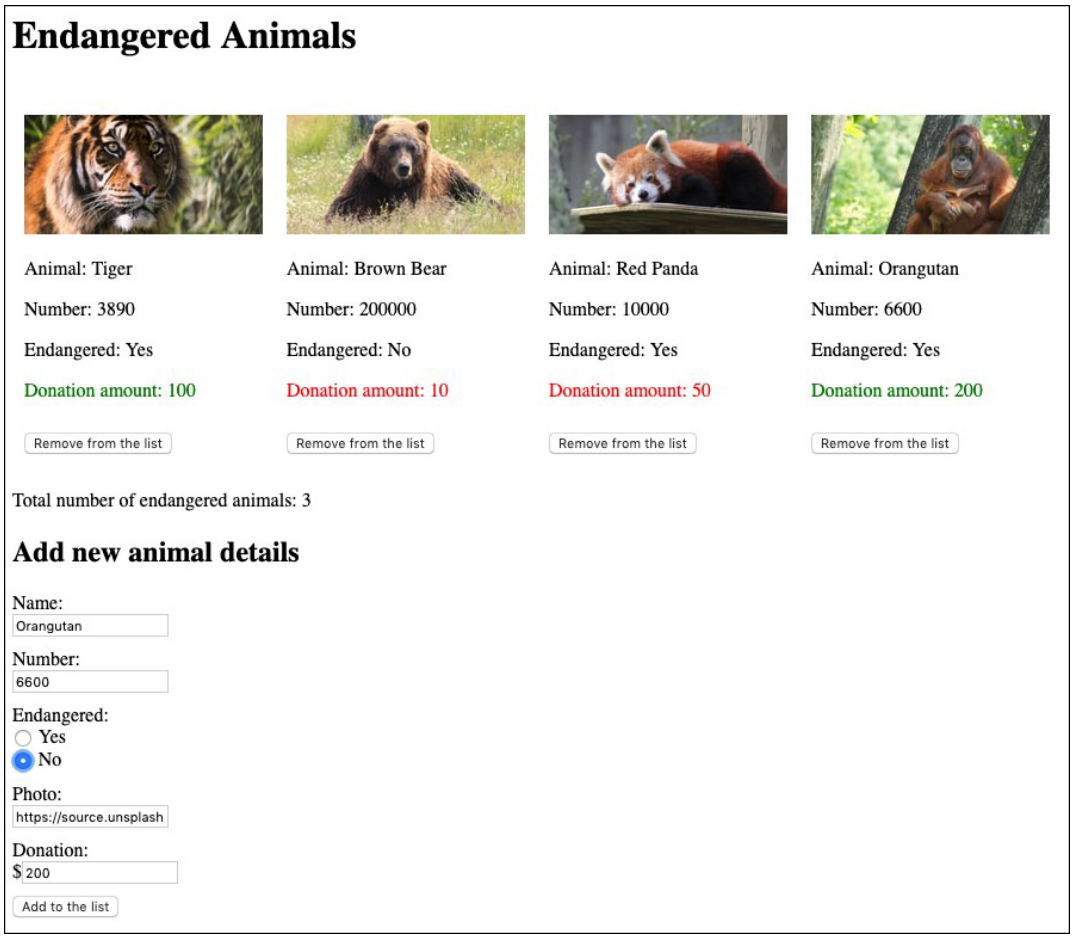

Figure 7.24: Endangered Animals app

In the output, the total number of endangered animals is displayed. When we add a new animal list, the number will go up, whereas when we remove the animal list, the number will go down. By using the Context API, we can receive the total number from the **`App`** component and send it to the **`AnimalCount`** component without passing through the intermediate component, **`Animal`**. There you have it. We have learned a lot of important techniques on how to communicate between components. Now, you can use these techniques in your real-world projects and make your apps more functional.

ACTIVITY 7.01: CREATING A TEMPERATURE CONVERTER

The aim of this activity is to build an app that converts the temperature between Celsius and Fahrenheit and displays the temperature status, such as hot, warm, cool, and cold. The output of the app should appear as follows:

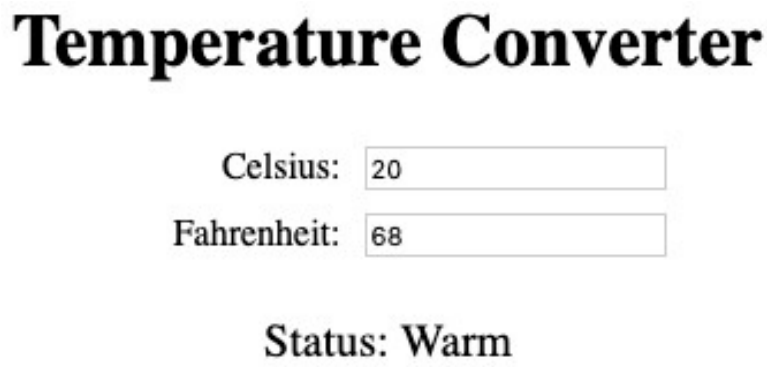

Figure 7.25: Temperature converter

When a user enters a number and updates either the Celsius or Fahrenheit input field, the other input field will display the number calculated and converted. Also, the status underneath the input fields will get updated too. Before we begin, ensure that you have completed all the exercises in this chapter as this activity will require some of the techniques we have discussed throughout the exercises.

Here is the diagram of the component tree you can follow:

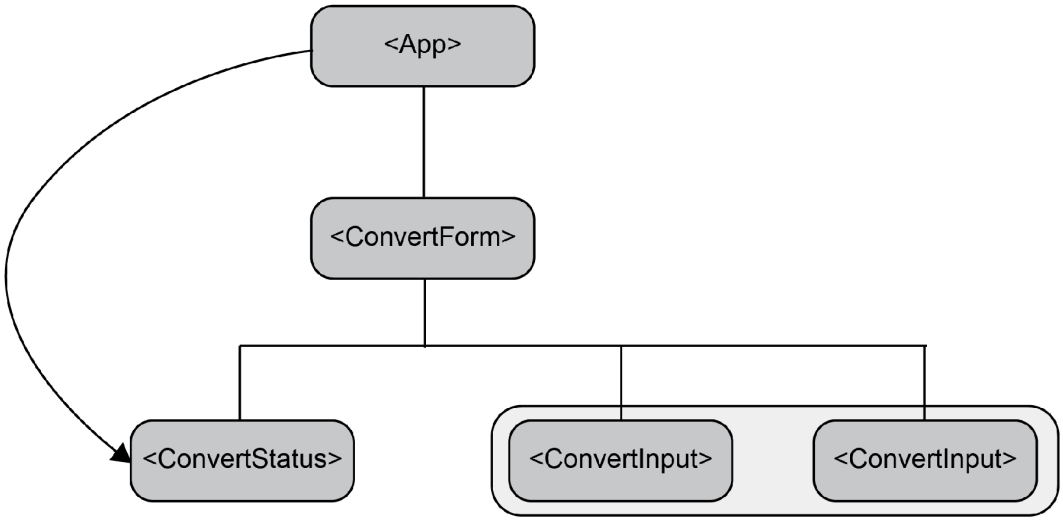

Figure 7.26: Pictorial representation of the temperature converter

The recommended file structure is as follows:

Figure 7.27: File structure of the activity

Here are some instructions to help you complete building the app:

1. The **<App>** component will contain the following functions:

- **getFahrenheit**: This function updates the **celsius** value to **fahrenheit**. The formula for this is **F = (C * 9) / 5 + 32**.
- **getCelsius**: This function updates the **fahrenheit** value to **celsius**. The formula for this is **C = ((F - 32) * 5) / 9**.
- **updateTemperature**: This is a callback function that's sent to the **<ConverInput>** component. It will be invoked when an **onChange** event is triggered and converts the temperature.
- The **<ConvertForm>** component will contain two **<ConverInput>** components, one for **celsius** and the other for **fahrenheit**. It also includes **<ConvertStatus>**.
- The **<ConvertInput>** component will contain the input elements with the callback function of **updateTemperature**.

2. To update the status of the temperature, we are going to use the Context API. Create the Context object and Provider components from the **<App>** component.
3. To consume the Context values, create a **<ConverStatus>** component and add the conditions shown in the following code block. Then, return the status wrapped with the **Consumer**.
4. Optionally, make use of HOCs for the status. In the **<ConvertForm>** component, create a HOC function and pass the **<ConvertStatus>** component through the HOC.
5. The sample code for the status is as follows. Please use it to show the status depending on the temperature. For example, when the **celsius** value goes higher than **50**, the status should show Very hot, whereas if it's lower than **0** but higher than **-10**, the Cool status should be displayed:

```
if (value > 50) {
  return 'Very hot';
} else if (value > 30) {
  return 'Hot';
} else if (value > 15) {
  return 'Warm';
} else if (value > 0) {
  return 'Cool';
} else if (value > -10) {
  return 'Cold';
} else if (value <= -10) {
  return 'Very cold';
}
```

> **NOTE**
>
> The solution to this activity can be found on page 656.

SUMMARY

In this chapter, we have learned how to pass data so that components can communicate with each other. There are various ways to pass data between components, and this chapter has covered all the essential methods of sending data to other components at different levels. Throughout this chapter, we looked at building an **`Endangered Animals`** app that showed us a list of animals along with a status of whether the animal is endangered or not. It also allowed us to submit new animal details.

First, we discussed how to send data from a parent to a child component. As React only allows us to pass data to a child component from a parent component, it is essential to understand how we can send data through props down to the child components first.

Then, we learned how to pass different types of props, including strings, numbers, Booleans, arrays, and components. We also learned about two important advanced React concepts, higher-order components and render props. Both techniques help make our app more reusable.

We also learned how to pass data from child to parent components by using callback functions. In the exercise on this, we discussed how to remove the list of animal details by referencing the callback function from **`<AnimalDetails>`** to the **`<App>`** component.

Furthermore, we discussed how components at different levels can communicate with each other. Between the components, we referenced a callback function that updates the state from the parent component. By updating the state, the app will be re-rendered and update the data from the component.

Finally, we learned what the React Context API is and how it can help avoid props-drilling. To use the Context API, we created the Context object and then created a Provider component so that we could pass the values to the consuming components. To receive the data from another component, we created a Consumer component and displayed the values from the props that were sent from the Provider.

With this essential understanding of how each component at different levels can communicate, we can now pass data around components as well as making components reusable. Having a good understanding of this will certainly help you make your React apps more functional. In the next chapter, we are going to discuss other important React topics, React forms and routing, which will allow us to build more complex React applications.

8

INTRODUCTION TO FORMIK

OVERVIEW

This chapter will introduce you to a new technique of writing form components in React, known as **`Formik`**. **`Formik`** will enable you to handle changes in forms, perform validation, handle error cases, and submit forms efficiently. By the end of this chapter, you will be equipped with the fundamental tools and techniques required to get started building forms with **`Formik`**, which is a scalable, performant form library in React.

INTRODUCTION

Frequently in web applications, we interact with the user to capture valuable information, such as their usernames, passwords, and preferences. Whether we want to register a new user, complete a bank statement, fill in a survey, or perform an advanced search using certain criteria and filters, or for any such similar cases, we have to use forms.

Since forms are an important aspect in any frontend development, the React team provided us with the minimum toolkit to customize forms based on our needs. There is a straightforward API called **`Formik`**, which works by leveraging existing functionality in JavaScript that allows us to write more portable and reusable code while dealing with forms.

The goal of this chapter is to convey to you a deep understanding of how forms work in React, composing resilient form elements as quickly as possible through a number of practical examples. We will cover all the various types of form handling techniques in React, from uncontrolled to controlled components, and take note of their simplicities.

We will then move on and learn more about **`Formik`**, the library that helps us to write full-fledged forms that include validation, controlling form submission, and keeping track of state. It has been built on the same principles that govern existing React components, so the learning curve is not pronounced, and we can quickly pick up the basics.

Let's get started with our first form handling technique, which is using uncontrolled components.

UNCONTROLLED COMPONENTS

We will start with a brief understanding of how forms are handled in React. In the most basic case, React treats forms as normal HTML components. Sometimes, we might need to take a close look at that input in a form so that we can see what its current value is. The most convenient way to achieve that is by using uncontrolled components, where, in essence, we maintain separate references to the DOM elements that we can utilize to manipulate and read the elements. By *uncontrolled*, we mean that we do not use React to change the value of the input field, but we let the browser handle the changes when we type something. This means that we can still write forms as usual and they would work as expected, as they do with plain HTML.

Uncontrolled components are useful when developers want to deal with the final state rather than the intermediate state of the component. For example, we can have the following form that asks for a username and a password:

```
import React from 'react';
class PlainForm extends React.Component {
  render() {
    return (
      <form noValidate={true} action="/login.php">
        <label>
          Email:
          <input type="text" />
        </label>
        <label>
          Password:
          <input type="password" />
        </label>
        <input type="submit" value="Login"/>
      </form>
    );
  }
}
export { PlainForm }
```

When we interact with this form, we can verify whether we can change our **`username`** and **`password`** fields using the **`noValidate`** attribute. When we click on the submit button, upon successful submission, we will see that the page will refresh and redirect to **`localhost/login.php`**.

However, as you may agree, this form is not very helpful as we do not have any good means to read the current input values or perform validation. In addition, there is no way to control the form-submitting process or to perform asynchronous updates.

In our case, all we need to do is to use React refs(references), covered in later chapters, which provides a way for the React library to have direct handles with the real DOM elements.

Let's see this in practice by running our first exercise.

EXERCISE 8.01: CREATING OUR FIRST UNCONTROLLED FORM COMPONENT

In this exercise, we are going to create our first uncontrolled form component in React using refs. Let's see how to do that:

1. Create a new React app using **`npx create-react-app uncontrolled`**. Start the project and take a look at the browser window that should have opened for you automatically:

```
$ npx create-react-app uncontrolled
$ cd uncontrolled
$ yarn start
```

2. Go to the **`src/App.js`** file and delete **`logo.svg`** and **`App.css`** and clear out the contents of **`App.js`**.
3. Inside **`App.js`**, create a new React component, **`App`**.
4. Create a new file named **`UncontrolledForm.js`** inside the **`src`** folder.

   We need to keep two references to the input elements, one for the name and one for the password.

5. Create a **`constructor`** property and define the two properties and their setters for each of the input fields:

```
  constructor(props) {
    super(props);
    this.name = null;
    this.password = null;

  this.setNameRef = element => {
    this.name = element;
  };
  this.setPasswordRef = element => {
    this.password = element;
  }
}
```

6. Next, we need to attach the refs to the actual components so that React will keep their DOM handles.
7. Add a **`ref`** property to each of the two fields:

```
<input type="text" ref={this.setNameRef}/>
<input type="password" ref={this.setPasswordRef}/>
```

8. To actually read the current values of the inputs, we need a handler that will be called when we submit the form. Add the following handler method:

```
  handleSubmit(e) {
  e.preventDefault();
  console.info('A name was submitted: ' + this.name.value);
  console.info('A password was submitted: ' + this.password.value);
  }
```

Here is the code for the complete component:

UncontrolledForm.js

```
3  class UncontrolledForm extends React.Component {
4  constructor(props) {
5    super(props);
6
7    this.name = null;
8    this.password = null;
9
10    this.setNameRef = element => {
11    this.name = element;
12    };
13
14    this.setPasswordRef = element => {
15    this.password = element;
16    };
17  }
```

The complete code can be found here: https://packt.live/3cyrff7

9. Load the component **UncontrolledForm** in **App.js**, using the styles provided in the **App.css** file located in the project folder (https://packt.live/2y0pvwa):

```
import React from 'react';
import {UncontrolledForm} from './UncontrolledForm';
import './App.css';

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

Run the app using the following command:

```
$ yarn start
```

When we load this component and we interact with it, we can inspect the values in the console of the browser:

Figure 8.1: Login form

When you click on the **LOGIN** button, the output should be as follows in the console of your browser:

```
A name was submitted: john.doe@gmail.com     UncontrolledForm.js:21
A password was submitted: johnd              UncontrolledForm.js:22
```

Figure 8.2: Uncontrolled component

The name and password properties of the Ref hold the current DOM input. Note that when the component loads, as the reference isn't attached to the DOM initially, it will show undefined for the input fields. Refs are always used to find the current value of a DOM element when needed.

We will be seeing more examples of Refs in the subsequent chapters. Refs have other beneficial uses; however, while handling forms, they are not recommended unless we have specific business requirements. In the majority of cases, we recommend using controlled components, which we will discuss next.

CONTROLLED COMPONENTS

A controlled input, or controlled component, is what we will be using most of the time when we want to implement forms in React.

The controlled part comes from the fact that the parent component possesses a reference to the current value that we assign to the input element. That value can be controlled using **`setState`**, while managing the state or the value can be passed as a prop from the parent component to its children components.

Let's take a look at the following code snippet:

```
handleOnNameChange = (e) => {
  this.setState({ name: e.target.value });
};
<input type="text" value={this.state.name} onChange={this.
handleOnNameChange} />
```

To obtain the current value of the input element and for any other updates of that element thereafter, we use the **`onChange`** event handler (or any similar handler that triggers when the actual DOM input gets updated). Then, on the event object itself, we get the current value using the **`e.target.value`** property.

> **NOTE**
>
> **`setState`** is necessary because if we don't update the state, React will not know when to re-render the new value. If we keep the value the same after each update, then we will see that the input will be locked down on the same value. Put differently, when we type, nothing happens. This is a source of widespread confusion among newcomers learning React.

Note that for certain inputs, the current value will be different; for example, checkboxes have an **`e.target.checked`** property instead.

Let's go through the following exercise to understand how a controlled form works.

EXERCISE 8.02: CONVERTING OUR FORM COMPONENT FROM UNCONTROLLED TO CONTROLLED

In this exercise, we are going to convert an uncontrolled form component to a controlled one using React's component state management. The purpose of this exercise is to understand the minimum steps required to perform that change. Perform the following steps:

1. Create a new React app using **`npx create-react-app controlled`**. Start the project and take a look at the browser window that should have opened for you automatically:

```
$ npx create-react-app controlled
$ cd controlled
```

2. Go to the **`src/App.js`** file and delete **`logo.svg`** and **`App.css`** and clear out the contents of **`App.js`**.
3. Inside **`App.js`**, create a new React component, **`App`**.
4. Create a new file named **`ControlledForm.js`** in the **`src`** folder.
5. Copy the contents of the **`UncontrolledForm.js`** file we saw in *Exercise 8.01*, *Creating Our First Uncontrolled Form Component* Now, rename the form as **`ControlledForm`**.
6. In **`ControlledForm.js`**, remove the references we created in the constructor in *Exercise 8.01*, *Creating Our First Uncontrolled Form Component* as they are not needed and write the following code:

```
this.setNameRef = element => {
  this.name = element;
};
this.setPasswordRef = element => {
  this.password = element;
}
```

7. Replace them with the state object, which will contain two properties. name and password should look like this:

```
constructor(props) {
  super(props);
  this.state = {
  name: '',
  password: '',
```

```
    };
  }
```

8. Replace the ref properties in each one of the inputs with the value properties and pass the current state value:

```
<input type="text" value={this.state.name}
    onChange={this.handleOnNameChange} />
```

9. Add an **onChange** event handler that will use a function callback to set the current input value in the state:

```
handleOnNameChange = (e) => {
  this.setState({ name: e.target.value });
}
```

Here is the code for the complete component:

ControlledForm.js

```
7  this.state = {
8   name: '',
9   password: '',
10  };
11  }
12
13  handleOnNameChange = (e) => {
14   this.setState({ name: e.target.value });
15   };
…
21  handleSubmit(event) {
22  event.preventDefault();
23   alert('A name was submitted: ' +
     this.state.name);
```

The complete code can be found here: https://packt.live/2LsHazH

10. To use this component **ControlledForm**, you need to import it into **App.js**:

```
import React from 'react';
import {ControlledForm} from './ControlledForm';
import './App.css';

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

Run the app using the following command:

```
$ yarn start
```

The following is the output of the form using the styles provided in the **`App.css`** file located in the project folder (https://packt.live/2AmA51i):

Figure 8.3: Controlled form component

When you click on the **`LOGIN`** button, an alert box will pop up displaying the **`username`**:

Figure 8.4: Uncontrolled component

Although we can clearly see that controlled components are the way we can handle forms in React, it turns out that they offer only limited functionality because there is nothing else other than manipulating simple form controls. We just assign the value to the state and we can change it using **`setState`**. How we update that state in different scenarios, for example, if we have to deal with complex forms, is up to us to configure. If we want to figure out how to perform proper validation or anything more advanced, we are essentially on our own. For simple cases, we may not need to do anything more than just using controlled components, but for real-world scenarios, we would like something more efficient. It turns out that there is a library in React called Formik that builds on top of the ideas of controlled components. Plus, it allows us to consolidate validation, form submission rules, and keep close track of the state of fields. Let's get started.

INTRODUCTION TO FORMIK

Formik is one of those libraries that came at just the right time to resolve a long-standing issue: how we handle really complex forms in React. Complex forms can include multiple embedded forms, dynamic fields and validation, or handling asynchronous checks with the backend. Prior to Formik, there were numerous options available, for example, using Redux-Form or React-Redux-Form, which basically stored the form state in a Redux store. That worked for a while, but complexity and peculiar bugs started to come in. Having to fix a gigantic form that was working once upon a time but now is not is something that we should all be wary of as it can happen all the time. For example, when we have a form where the validation happens in different stages and the outcome of each step depends on how the previous steps validate, it becomes difficult to find even the smallest bugs. And testing this logic can become even trickier because you would need to cover all the scenarios to figure out which one fails. Therefore, it is preferable to have the code as self-explanatory as possible and easy to understand.

Formik, on the other hand, goes back to basics by storing the form state locally and not globally, which is what is currently recommended by most React practitioners. What we mean by locally is that we are not using a global store object such as Redux to store the form state and values; instead, each component uses a state object to store its current status.

As mentioned on Formik's website, it offers significant advantages over plain React forms or the aforementioned libraries.

Let's discuss the benefits of Formik before we look at practical exercises on how we can use it. When we interact with this component again, nothing changes and it performs just like the previous two components, controlled and uncontrolled, as we saw previously.

In the following section, we are going to examine Formik.

ADVANTAGES OF FORMIK

Before we dive into the structure of a Formik component, let's have a look at a few key benefits of this library:

- Easy to integrate: Since Formik keeps the state local, it's more effortless to convert existing forms (either controlled or uncontrolled) into Formik ones. It does not compete against a state management library such as Redux or MobX, so the amount of dependencies stays small. In other words, you don't have to implement elaborate structures to develop forms.
- Easy to understand: Learning new libraries and extensions to existing ones always takes some time to read and understand the documentation. You can always come up with an example case that will not adapt to your needs, so you will have to improvise. With Formik, you don't deviate from the concepts that you already know, such as setting state and receiving props, so you will not have any issues learning how to utilize it quickly.
- Easy to configure: Formik offers several options when it comes to handling form validation, retrieving the values from and out of the state, and submitting the form. You can use Formik as an HOC when you want to pass one component to another, or as a render props pattern where the props of the parent component are passed to its children components. In addition, you can include a third-party validation library such as Yup or **`Spected`**, making it more suitable for real production cases.

Now, let's look at the anatomy of a Formik component.

ANATOMY OF A FORMIK COMPONENT

The steps required to create our first Formik component are not complex. First, we need to get access to the library. We can do this via the following command:

```
$ yarn add formik
```

Or, if you are using **npm**, run the following command:

```
$ npm install formik --save
```

There is also an option for a CDN script tag:

```
<script src="https://unpkg.com/formik/dist/formik.umd.production.js"></
script>
```

When the library loads, it will attach a Formik property (used for building forms) to the global scope of the browser: **window.object**.

After this step, let's replace the code in the controlled component, **ControlledForm.js**, which we implemented previously, in *Exercise 8.02*, *Converting Our Form Component from Uncontrolled to Controlled* and create a new file called **formikForm.js**:

```
<form onSubmit={handleSubmit}>
<label>
  Name:
  <input type="text" name="name" value={values.name}
    onChange={handleChange} />
</label>
<label>
  Password:
  <input type="password" name="password" value={values.password}
    onChange={handleChange} />
</label>
<input type="submit" value="Login" disabled={isSubmitting}/>
</form>
```

Noticed any differences? We replaced the **onChange** handlers of the controlled component with only one **handleChange**, we used **values.password** and **values.name** instead of using state, we added a name property to each of the input elements, and added an extra disabled value to the submit button. So eventually, we removed some handlers and delegated them to **Formik**, and so we got rid of boilerplate code that appears most of the time when we create forms using controlled components.

If you are wondering where we get those values from, it is from the Formik component itself. Let's complete the code to see it in action. To do that, we only need to import the Formik component and wrap our form with the following code:

FormikForm.js

```
6  <Formik
7  initialValues={{ name: '', password: '' }}
8  onSubmit={(values, { setSubmitting }) => {
9  setTimeout(() => {
10  console.info(JSON.stringify(values, null, 2));
11  setSubmitting(false);
12  }, 400);
13  }}
14  >
15  {({
16    values,
17    handleChange,
18    handleSubmit,
19    isSubmitting,
20  }) => (
```

The complete code can be found here: https://packt.live/2WwAhDG

As you can see, the Formik component is like a constructor of properties and configuration. We use **`initialValues`** to define our input values that we can change and customize later and also define an **`onSubmit`** handler that will trigger when we call the **`handleSubmit`** handler that gets passed as a render property. Speaking of render variables, we have to access a couple of them, such as values, the object that we used when we defined **`initialValues`** and **`isSubmitting`**, which will change whenever we call the **`setSubmitting`** function.

If we run the preceding component, we can see that when we submit the login form, the button is disabled for a moment and the following is logged in the console:

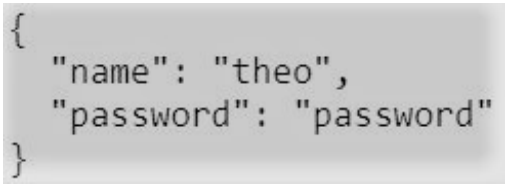

Figure 8.5: Console output of the form element

Therefore, if we inspect the code example, the following set of events happen:

- The user types in username and password.
- The **`onChange`** event handlers update the values property.
- The user clicks on the Login button.
- The **`handleSubmit`** handler is invoked.
- The form immediately re-renders with the **`isSubmitting`** property as true to prevent double submissions; we disable the login button.
- When the submit handler is triggered, the timeout handler runs for half a second. Then, it prints out the contents of the values property in the console and uses the **`setSubmitting`** function to set **`isSubmitting`** to false.
- The form, after approximately half a second, re-renders with the **`isSubmitting`** property as false and the login button becomes clickable again.

> **NOTE**
>
> Inside the **`submitHandler`** closure, it is possible, if we are sufficiently bold, to modify the values property and print something else in the console.

For instance, if we add the following line after the **`setTimeout`** call in the **`FormikForm.js`** file:

```
<Formik
initialValues={{ name: '', password: '' }}
onSubmit={(values, { setSubmitting }) => {
  setTimeout(() => {
  console.info(JSON.stringify(values, null, 2));
  setSubmitting(false);
  }, 400);
  values.name = "Hello";
  }}
>
```

Then the following things happen:

- **`values.name`** will change internally to **`Hello`**.

  Because there is no rendering during that time, nothing will happen over the next half-second.

- After the timeout triggers, **`console.info`** will print the new **`values`** object and the next re-render will update the **`input`** value of the element to **`Hello`**.

Therefore, it is recommended not to change or update the values object in this way as it's not predictable and may lead to weird behavior.

Let's take a closer look at how we can use initial values and handlers in **`Formik`**.

INITIAL VALUES AND HANDLERS

The Formik component is the main entry point for our forms and contains several properties that let us configure their behavior.

In any case, we first need to define the **`initialValues`** property, which must be a plain object with string property names. The values of each property can be anything; for example, we can use an array to collect user preferences:

```
initialValues={{
  preferences: ["Agree to Terms", "Subscribe to Newsletter"],
  socialAccountPage: {
    twitter: "https://twitter.com/packt",
    facebook: "https://facebook.com/packt",
    }
  }}
```

In the first case, for the nested object, **`socialAccountPage`**, we need to reference the absolute path in the input fields that the following name parameter needs to match for the form to work correctly. Therefore, the **`socialAccountPage`** nested object can be modified and will look like this:

```
<input type="text" name="socialAccountPage.facebook" value={values.
socialAccountPage.facebook}
    onChange={handleChange} />
<input type="text" name="socialAccountPage.twitter" value={values.
socialAccountPage.twitter}
    onChange={handleChange} />
```

When we have an array of values, we need to use brackets to access the index of the field, for example:

```
<input type="text" name="preferences[0]" value={values.preferences[0]}
onChange={handleChange} />
<input type="text" name="preferences[1]" value={values.preferences[1]}
onChange={handleChange} />
```

Formik also offers a **`<FieldArray>`** component that provides advanced manipulation functions when working with lists of values.

As for the handlers, the most important one is the **`onSubmit`** handler that we saw previously, and it triggers when we submit the form. This handler passes on the values and actions objects. Here is the signature of the actions object:

```
export interface FormikActions<Values> {
  setStatus(status?: any): void;
  setError(e: any): void;
  setErrors(errors: FormikErrors<Values>): void;
  setSubmitting(isSubmitting: boolean): void;
  setTouched(touched: FormikTouched<Values>): void;
  setValues(values: Values): void;
  setFieldValue(field: keyof Values & string, value: any, shouldValidate?:
  boolean): void;
  setFieldError(field: keyof Values & string, message: string): void;
  setFieldTouched(field: keyof Values & string, isTouched?: boolean,
  shouldValidate?: boolean): void;
  validateForm(values?: any): Promise<FormikErrors<Values>>;
  validateField(field: string): void;
  resetForm(nextValues?: Values): void;
  submitForm(): void;
  setFormikState<K extends keyof FormikState<Values>>(f: (prevState:
  Readonly<FormikState<Values>>, props: any) => Pick<FormikState<Values>,
K>,
  callback?: () => any): void;
}}
```

We have seen **setSubmitting** before, and there are other ones that we can use based on our specific requirements. For example:

- **resetForm**: This is used to reset the form back to the **initialValues** state.
- **validateField**: This is used to trigger a specific field validation.
- **setTouched**: This is used when we want to mark the field as touched or visited. It is useful in scenarios where we want to show an error message if that field has been edited once while the user is accessing the form and not when the form is loaded for the first time.
- **setStatus**: This is used when we want to attach a custom status property to our form, for example, **isApproving** or **isValidating**. This needs to be paired with an associated **initialStatus** property:

```
initialStatus={{isValidating: false}}
```

  Other than the **onSubmit** handler, we also have access to the following handlers:

- **onReset**: Triggered just before the form is reset back to its **initialValues** parameter.
- **validate**, **validateOnBlur**, **validateOnChange**, **validationSchema**: Triggered in cases when we want to validate the form. We will see more examples pertaining to validation later in the chapter.

In addition to the **Formik** component, there is another component that offers a different API and is suitable for cases when you want to pass one component to another. These components are known as Formik higher-order components.

FORMIK HIGHER-ORDER COMPONENTS

We saw the basics of Higher-Order Components (HOCs) in *Chapter 7*, *Communication between Components*. Now, let's see the behavior of HOCs with Formik.

withFormik is an HOC that allows us to separate the definition of the Formik component configuration from the UI layer. This is useful when we have a component where we just want to have access to the Formik properties and callback methods without actually defining form elements, or when we have an existing form and we want to delegate its callbacks to Formik without changing the UI of the original form.

Let's take a look at the template code of **`LoginForm`**.

> **NOTE**
>
> The template code is located in https://packt.live/3btLSIc.

```
const MyEnhancedLoginForm = withFormik({
mapPropsToStatus: () => ({ isValidating: 'false' }),
handleSubmit: (values, { setSubmitting, setStatus }) => {
setStatus({ isValidating: 'true' });
setTimeout(() => {
console.info(JSON.stringify(values, null, 2));
setSubmitting(false);
setStatus({ isValidating: 'false' });
}, 1000);
},
displayName: 'LoginForm',
})(LoginForm);
```

Then, add the **`LoginForm`** code in the same file as above

`WithFormikExample.js`

```
5  const LoginForm = (props) => {
6  const {
7    values,
8    handleChange,
9    handleSubmit,
10    isSubmitting,
11    status,
12  } = props;
13  return (
14  <form onSubmit={handleSubmit}>
16    Name:
17    <input type="text" name="name" value={values.name}
18      onChange={handleChange} />
19    </label>
…
26  <input type="submit" value="Login"
    disabled={isSubmitting}/>
27  Status: {status.isValidating}
```

The complete code can be found here: https://packt.live/2Wv0v9T

As you can see, the UI layer is separate from the component containing configuration and business logic.

There are several advantages associated with using **`withFormik`** as it allows some degree of reusability for our components. We can update the UI layer without touching the business layer, giving us more flexibility while organizing our code base.

CONNECT

This is just a utility component designed to inject the Formik context into any React component via their props. For example, let's look at the template code:

```
import { connect, getIn } from 'formik';
const StatusMessage = props => {
  const isValidating = getIn(props.formik.status, 'isValidating');
  return isValidating ? 'Is Validating' : 'Is not Validating'
};
export default connect(StatusMessage);
```

Now, we need to place this component anywhere inside the Formik wrapper. In our example, we have it just before the closing form tag:

```
Status:<StatusMessage />
</form>
```

The connect component allows us to turn any component into a Formik lookalike as we have access to the entire API, callbacks, and state variables.

Now that we have seen how we can work with straightforward cases of Formik, let's explore how to validate forms in React and the common challenges faced during the process.

VALIDATING A FORM

Formik makes it easy to add validation rules and checks throughout your forms and control when you can trigger them. At a basic level, it provides form-level validation checks. We only need to provide a validate property that needs to entail all the validation logic for the form.

Inside the validation function, we need to create an error object and assign properties using the same name as the values we provided in the **`initialValues`** object. That way, we can access the errors object inside the render props function and update the UI. Let's see how to add field validators to our login form through the following exercise.

EXERCISE 8.03: ADDING FIELD VALIDATORS

In this exercise, we are going to add field validators to the form component, **`LoginForm`**, created in the previous section. We will use Formik's validate property to do that. We will also use Formik's **`ErrorMessage`** component to design the error messages that will be shown once a validation error is thrown. Let's see how:

1. Start a new CRA app. You can choose any name for the new app:

```
$ npx create-react-app <name>
```

2. Go to the **`src/App.js`** file and delete **`logo.svg`**, **`App.css`**, and clear out the contents of **`App.js`**.
3. Copy **`App.css`** from the previous exercise.
4. Inside **`App.js`**, create a new React functional component, **`App`**.
5. Create a new file named **`FormLevelValidation.js`** in the **`src`** folder.
6. Copy the existing code from the login form in the previous section:

`FormLevelValidation.js`

```
21  const FormLevelValidation = () => {
22  return (
23  <Formik
24
25  initialValues={{ name: '', password:''}}
26  onSubmit={(values, { setSubmitting }) => {
28    setTimeout(() => {
29    console.info(JSON.stringify(values,
      null, 2));
```

`The complete code can be found here:` https://packt.live/2AkCOZ4

7. Add the individual field-level validation functions:

```
function validateName(value) {
  let error;
  if (!value) {
  error = 'Name is Required';
  }
  return error;
}
function validatePassword(value) {
```

```
    let error;
    if (!value) {
    error = 'Password is Required';
    }
    return error;
  }
```

8. Remove the existing validate property in the **`Formik`** constructor as we are going to perform validation on each field individually.
9. Replace the input elements with the **`<Field />`** element, adding the respective **`validate`** property function:

`FormLevelValidation.js`

```
41  form onSubmit={handleSubmit}>
42    <label>
43      Name*:
44      <Field type="text" name="name"
45        validate={validateName}
46        onBlur={handleBlur}
47        onChange={handleChange} />
48    </label>
49    <ErrorMessage name="name" />
50    <label>
51      Password*:
52      <Field type="password" name="password"
53        validate={validatePassword}
54        onBlur={handleBlur}
55        onChange={handleChange} />
56    </label>
```

`The complete code can be found here:` https://packt.live/2T39RaM

10. To use this component, you need to import it into **`App.js`**:

```
  import React from 'react';
  import FormLevelValidation from './FormLevelValidation';

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

11. Run the app using the following command:

```
$ yarn start
```

The output is as follows:

Figure 8.6: Login form component

If you perform the preceding steps, then the form will behave as before. Now, we have gained considerable advantages as we can extract and reuse validation logic across many components.

CONTROLLING WHEN FORMIK RUNS VALIDATION RULES

By default, **`Formik`** triggers a validation phase on change, on blur, and just before the **`onSubmit`** handler is invoked. However, we can configure this behavior using the following **`boolean`** properties:

- **`validateOnChange`**: True if validation happens when we call **`onChange`**
- **`validateOnBlur`**: True if validation happens when we call **`onBlur`**

  We also have the option to use the following two callback handlers provided by the render props:

- **`validateField`**: Validates the field specified by their name; for example:

```
validateField('name')
```

- **`validateForm`**: Validates the entire form whenever it is called.

To understand how the preceding fields, work, let's look at a small exercise on how to configure the validation phases.

SCHEMA VALIDATION

Schema validation is basically defined as an object of keys named as the **`initialValues`** property and, for each key, there is a function that performs the list of validations.

When we trigger a validation phase, **`Formik`** will use the **`validationSchema`** object to map the values object to it. Then, it will convert any errors that Yup returns into the familiar errors object and passes on the render props callback. Hence, those two libraries will work seamlessly together.

If we defined extra validation rules using the validation property, for example, then they will be called in conjunction. If any of them resolve with an error, they will combine to form the **`errors`** object. This is not recommended though, as you will see in the following example: the values keys will clash so only one of the validation messages will show.

First, install **`yup`**, which is an object schema validation library. We define some complex validation rules that we can check against the values that we passed during runtime:

```
$ npm install yup --save
```

Or, using **`Yarn`**:

```
$ yarn add yup
```

Create a new file named **`FormYupValidation.js`** and copy over all the code from the previous exercise.

Then, create a new function called **`LoginSchema`** with the following code:

```
  import * as Yup from 'yup';
…
  const LoginSchema = Yup.object().shape({
  name: Yup.string().required('Required'),
  password: Yup.string().min(8, 'Too Short!')
});
```

Here, we define a **Yup** object with the following rules:

- **name** should be a string and is required.
- **password** should be a string with a minimum of eight characters.

Now, assign **LoginSchema** to the **validationSchema** property:

```
  <Formik
  validationSchema={LoginSchema}
…
```

Finally, replace the input fields, **onChange** and **onBlur**, with the following:

FormYupValidation.js

```
53  <Field type="text" name="name"
54    onBlur={handleBlur}
55    validate={validateName}
56    onChange={handleChange}/>
61    <Field type="password" name="password"
62    onBlur={handleBlur}
63    validate={validatePassword}
64    onChange={handleChange}/>}
```

The complete code can be found here: https://packt.live/2y1hZRP

Now, if you use this form, you will see some error messages. If you enter a password with fewer than eight characters, then only the error message from Yup will show:

Figure 8.7: Error message displayed when the password is too short

As you can clearly see, mixing different validation rules can lead to confusing behavior and it should be avoided. Let's conclude our understanding of Formik by looking into the Form submission process.

EXERCISE 8.04: CONTROLLING SCHEMA VALIDATION PHASES

In this exercise, we will see how to control schema validation phases:

1. Start a new CRA app. You can choose any name for the new app:

```
$ npx create-react-app <name>
```

2. Go to the **`src/App.js`** file and delete **`logo.svg`**, **`App.css`**, and clear out the contents of App.js.
3. Copy **`App.css`** from the previous exercise.
4. Inside **`App.js`**, create a new React functional component, **`App`**.
5. Create a new file named **`FormValidationControl.js`**.
6. Copy the existing code from the **`FormLevelValidation.js`** file created in the previous exercise.
7. Add the following properties to the **`Formik`** component:

```
validateOnChange={false}
validateOnBlur={false}
```

   Doing so will trigger validation only when we submit the form.

8. To manually restore the validation phases, extract the following properties from the render props and add them to the input fields:

FormValidationControl.js

```
41  {({
42  handleChange,
43  handleBlur,
44  validateField,
45  handleSubmit,
50  Name*:
51  <Field type="text" name="name"
52    validate={validateName}
53    onBlur={(e)=> {
54    handleBlur(e);
55    validateField('name');
56    }}
57    onChange={(e)=> {
58    handleChange(e);
59    validateField('name');
60    }}/>
```

The complete code can be found here: https://packt.live/2Ww7P4V

9. Lastly, to make it more obvious how **onChange** triggers validation, add another check for the **password** field:

```
function validatePassword(value) {
  let error;
  if (!value) {
  error = 'Password is Required';
  }
  if (value && value.length < 8) {
  error = 'Min length of Password is 8 chars';
  }return error;
}
```

10. To use this component, you need to import it into **`App.js`**:

```
import React from 'react';
import './App.css';
import FormValidationControl from "./FormValidationControl";

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

export default App;
```

11. Run the app using the following command:

```
$ yarn start
```

Now, when we type our password and it has fewer than eight characters, and only when we lose focus (the **`onBlur`** event happens), the validation will trigger:

Figure 8.8: Form component with the onBlur event

In that case, we will not annoy the user who is typing the password by displaying the error message, as their interaction is in progress. Prior to that change, when the user started typing the first character, the **`onChange`** validation would trigger and display an error message. This helps in improving the UX (user experience).

Now, when we have more than a few simple checks to perform in the validation phase, or we are already using a third-party library to check parameters, there is another option available in **`Formik`** called schema validation.

SUBMITTING A FORM

The most crucial step while working with forms is the submission process. In this phase, typically, we gather all the form fields and process them to be sent to the server.

When we send the form data to the server, we want to show the interactions that are happening during that request. For example, we may want to disable the submit button so the user will not resubmit the form while the server is processing the form. We also want to include a loading indicator to indicate to the user that the request is being processed.

Finally, we want to act based on the server response. If the server accepts the request, then we may want to redirect to another page. If the server rejects the form for whatever reason, we may want to retrieve the list of errors and display them to the user. There are lots of use cases, and we would like to have customization in this process.

Formik is unopinionated when it comes to the submission phase. By default, when we trigger the **`onSubmit`** or **`handleSubmit`** actions, we need to perform the following:

- Iterate over the field names specified in the **`initialValues`** object and mark them as touched. This is to ensure that if we have a validation error, we will prevent the form from submitting and show an error message.
- Set **`isSubmitting`** to true. Doing that will pass the new state to the render props function, so we can update the UI. For example, we disable the login button by using either a **`css`** class or a property.
- Set **`isValidating`** to true. This is a secondary field that gets assigned before any validation check happens. We can leverage this field, for example, when we have asynchronous validation in place.
- Perform all validation checks.
- Either abort in the case of errors or proceed to submit the form if validation happens. In either scenario, **`isValidating`** is set to false.

- Trigger the **`onSubmit`** or **`handleSubmit`** handler. This is the user-controlled code we define. In addition to the logic we want to implement, we need to use the **`setSubmitting`** handler to update the **`isSubmitting`** value. If we don't do that, the UI will not update as the props may not change.

The following diagram depicts the entire process for each step:

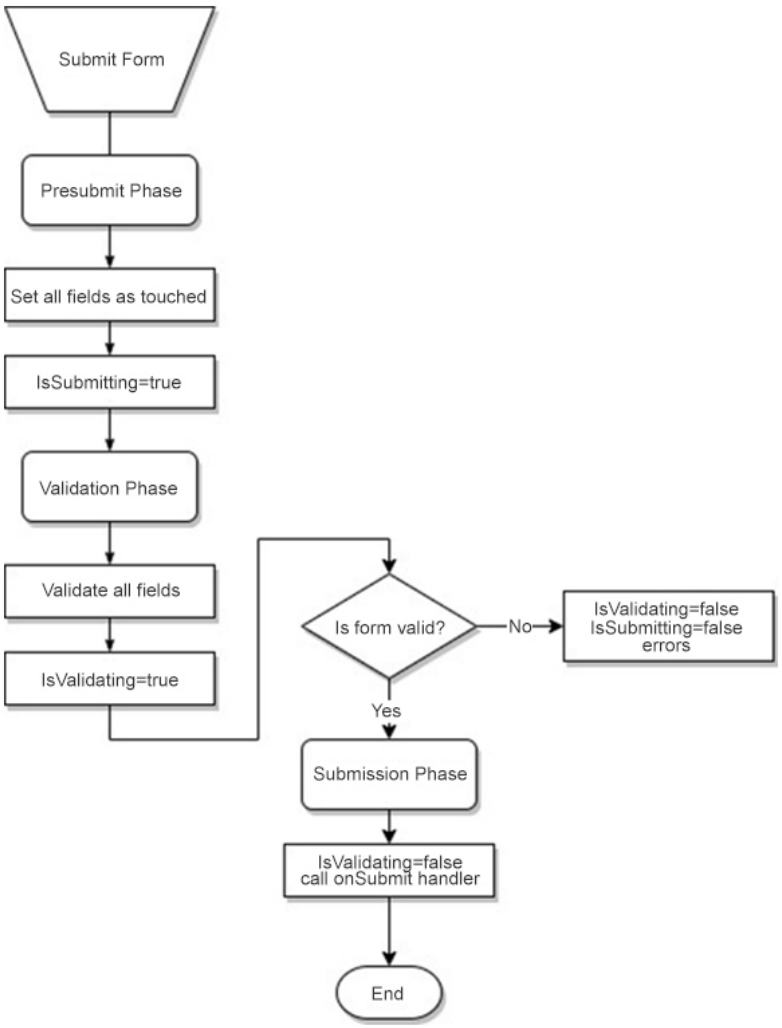

Figure 8.9: Flow chart of the Formik form component

Armed with all this knowledge about how Formik works and which components it offers, let's look at writing our own small form from scratch using real-world requirements this time. We will put into practice what we have learned in this chapter.

ACTIVITY 8.01: WRITING YOUR OWN FORM USING FORMIK

The aim of this activity is to design and write our own form component for a new user registration form. In this form, there will be a name, a unique email, a password, and a terms and conditions approval that the user has to fill. We will have a set of validation requirements for each field and we also need to perform asynchronous validation with the server before we submit the form. Our desired outcome will be a complete, user-friendly form that will be able to sign up new users, as shown in the following screenshot:

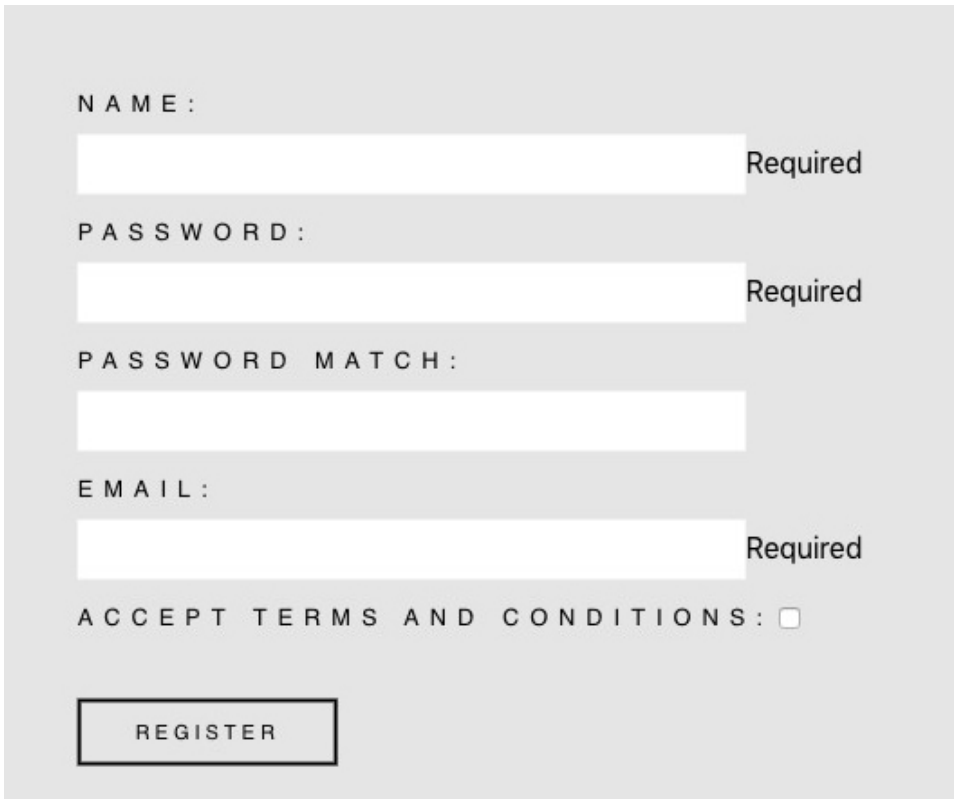

Figure 8.10: New user sign-up form

Before you begin, ensure that you have performed all the previous exercises and understood the various use cases. Here are the steps to complete the activity:

1. Create a new CRA.
2. Go to the **`src/App.js`** file and delete **`logo.svg`**, **`App.css`**, and clear out the contents of **`App.js`**.
3. Inside **`App.js`**, create a new React functional component, **`App`**.
4. Create new **`username`**, **`password`**, and email input fields.
5. Create a password match field. This should match the first password field for the form to validate. If it does not match, it should display an error message. This validation rule must run **`onBlur`**.
6. Create a **`checkbox`** field for accepting the terms and conditions.
7. Add a **`Register`** button.
8. Add required, six characters minimum, **`24`** characters maximum, and a string as validation rules for the Username field.

   The validation rules must run **`onBlur`** only.

   Hint: review the code for the **`validationSchema`** option.
9. Add a required, eight characters minimum, string, and at least one digit (one lowercase character and one uppercase character) as validation rules for the password field.

   The validation rules must run **`onBlur`** only.
10. Add the validation rules for the **`email`** field. Use a custom test example to validate the uniqueness of the email.
11. For server validation, we need to simulate a delay when checking with the server. You don't have to write code for the server; just have a list of **`10`** emails that you want to check against for uniqueness. If the user enters an email ID that is included in the list, it will be rejected with an error message. Add a **`200`**ms delay when performing validation; the validation rules must run **`onBlur`** only.

    Review the code to return a Promise object when defining a validation property or a schema. How can you simulate a delay using a Promise object? For the loading indicator, review which property is passed when the form is validating

12. Add the required validation rule for the **`checkbox`**.
13. When we submit the form, add a **`200ms`** delay to simulate the backend processing. During that time, ensure that the register button is disabled.

This form touches on a little bit of everything we have covered in this chapter. We have used form fields, validation states, controlling when validation happens, and how to perform asynchronous validations. Feel free to spend some time on this activity before moving on.

> **NOTE**
>
> The solution to this activity can be found on page 666.

SUMMARY

In this chapter, we have learned how forms work in React. We have examined the differences between controlled and uncontrolled components and practiced a few exercises.

We continued our journey by introducing Formik, which is a good helper library for building forms and is aligned with the fundamental concepts of React form components.

The majority of our time was taken up with the concepts of validation and state management. We discovered how we can add validation rules either via custom functions or via a schema validation engine. We looked at handlers and initial values and how we can utilize them to control the form state. We also explored the other HOC components that the library offers that allow us to hook anything into Formik's context.

At the end of the chapter, we seized the opportunity to create our own form component that represents a real-world scenario where we need to register a new hospital site management system. By utilizing our prior practical knowledge of form validation, state management, and error handling, we put into practice the skills we have learned.

Armed with this fundamental understanding of how forms work in React, we can now tackle the next fundamental concept, which is routing and handling various cases of navigation with React. In the next chapter, we will utilize React Router v4, which is the most stable routing library at the moment and will allow us to build bigger and more complex applications.

9

INTRODUCTION TO REACT ROUTER

OVERVIEW

This chapter looks at how to implement React Router on React applications in order to build single-page applications. It starts by building simple nesting routing structures and then goes on to create parameters via URL parameters in the routes. This chapter will enable you to design missing pages by displaying an appropriate page not found message to the user. You will be equipped with techniques that will enable you to introduce navigation into your application. By the end of this chapter, you will have a solid foundation of using basic React Router.

INTRODUCTION

When you are building a single-page application in any language or framework, it is typically not just one giant static page. It usually has links to help you navigate to the other components or sections of the application. A major part of the success of any such application goes to the design of the page-level manipulation, where there are different section links that enable you to navigate around the application with a sense of purpose. React remains no different in this regard. In fact, React has the ability to create a similar number of different URLs or paths that can be used to navigate around your app and to each of the individual components in a way that feels organic and intuitive to the user.

The goal of any single-page app is that the user should feel as though they are navigating through just a regular site but with far more interactivity. React Router goes a long way toward establishing that feeling by building a system for integrating different URLs seamlessly into your app but keeping the user within the confines of your React application.

This chapter includes a few hands-on approaches to explore the functionalities of React Router. Let's get started.

UNDERSTANDING BROWSER ROUTING

Modern web browsers have Hist**ory** and **`Location`** APIs that provide access, through the browser, to the different parts of the browser's history, location, and state. By using this functionality, you will be able to effectively handle routing. For example, the **`History`** API in modern browsers exposes the web browser's current session state and allows you to manipulate that state, either by moving forward using **`forward()`**, moving backward using **`back()`**, or moving to a specific point in the history using **`go()`**.

The Location APIs allow JavaScript code to inspect the current location and path through the **`window.location.href`** and **`window.location.path`** properties and pull different types of information from it, allowing developers to figure out the navigation paths on a website and handle them accordingly.

React Router is a library for React that handles navigation and URLs and translates them into displaying the appropriate component for the URL.

React Router watches for state changes in the browser (via **History**), gets information about where the user currently is and where they're going (via **Location**), and uses the combination of the two, along with the JavaScript events, to build a set of routes and components to handle those routes. Let's go through the following exercise to build a basic React router using the **Location** API.

EXERCISE 9.01: BUILDING A BASIC ROUTER USING THE LOCATION API

In this exercise, we will create a simplified router using the **Location** API. This will help us to understand a little more about how the React Router works, since being able to understand the underlying code concepts will help us to learn how to use libraries better:

1. Start by creating a new React app:

```
$ npx create-react-app router
```

2. Go to the **src/App.js** file, delete **logo.svg** and **App.css**, and then clear out the contents of **App.js**.

   We will need to access a few properties via the **Location** API, specifically through **window.location**. Inside **App.js**, create a new React functional component called **App**, and display two key properties: **window.location.href** and **window.location.pathname**. Add the **window.location.href** and **window.location.pathname** properties as follows:

```
import React from 'react';
  const App = () => (
    <div className="Router">
      <h1>Example Router</h1>
      <hr />
      <p>href: {window.location.href}</p>
      <p>path: {window.location.pathname}</p>
    </div>
  );
export default App;
```

This component is just a standard functional component that will only display the two properties that we listed. The **`window.location.href`** property will display the full URL including the protocol and the domain name, while the **`window.location.pathname`** property will contain only the **`pathname`** variable. Between **`window.location.href`** and **`window.location.pathname`**, the former property will not be of much help; instead, we only care about the pathname variable.

3. Try typing multiple URLs into your browser, such as **`http://localhost:3000/about`** or **`http://localhost:3000/foo`**. You should see **`/about`** or **`/foo`** show up in your component:

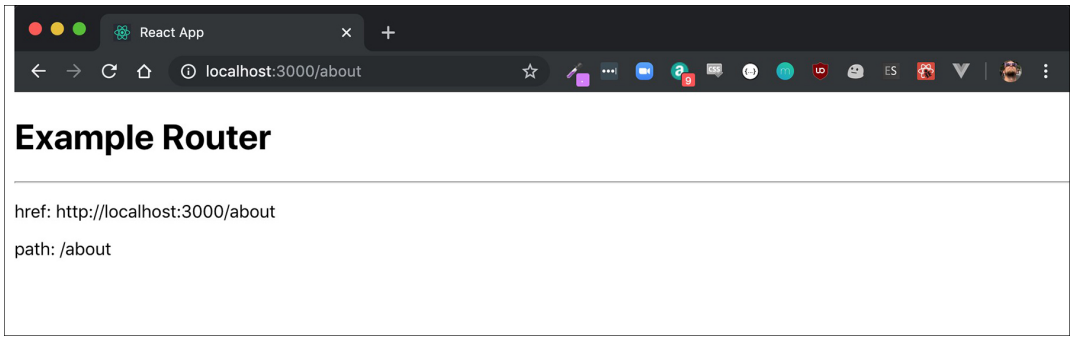

Figure 9.1: Basic router app

As you can see from *Figure 9.1*, the path variable contains what we really need: that is, information about the domain and port that will tell us what route or component the user is trying to navigate to.

From here, we can start implementing our basic router, which should just be a function that returns a valid component. To do this, we are going to split the path property when a forward slash is encountered.

> **NOTE**
>
> The catch here is that the split operation will return both sides of the string, whether there is a value present or not.

`/about` will split at the forward slash, **`/`**, returning two values: a blank string and about. We also want our paths to be *case-insensitive*. We can do this by changing the pathnames to lowercase.

4. Using the preceding guidance, create a **RouteTo** utility function that will split the path at the forward slash:

```
const RouteTo = path => {
  const paths = path
    .split('/')
    .map(p => p.toLowerCase())
    .slice(1);
  switch (paths[0]) {
  default:
  return <div className="Default">Default Route</div>;
  }
};
```

> **NOTE**
>
> The **p** argument in the map method is an ES6 arrow syntax for defining a function, while **p** is a temporary variable that gets declared within the scope of this function.

Inside the **RouteTo** utility function, we have created an array, called paths, that essentially contains two values: an empty string and about (in lowercase). We obtained these values after splitting path at the forward slash, **/**. We also have a default case where we have defined a route in case there are no paths to display.

5. Now we need to hook up the browser's **Location** API to our component, which We will do by placing a call to **RouteTo()** (that is, the function that we just wrote) in our **render()** function.

6. Call the **RouteTo()** method and hook the **window.location.pathname** property to it:

```
const App = () => (
  <div className="Router">
  <h1>Example Router</h1>
  <hr />
  <p>href: {window.location.href}</p>
  <p>path: {window.location.pathname}</p>
  <hr />
```

```
    {RouteTo(window.location.pathname)}
    </div>
  );
```

7. Refresh the page. We should expect to see **`Default`** Route show up at the bottom of the page because we haven't created an about page yet:

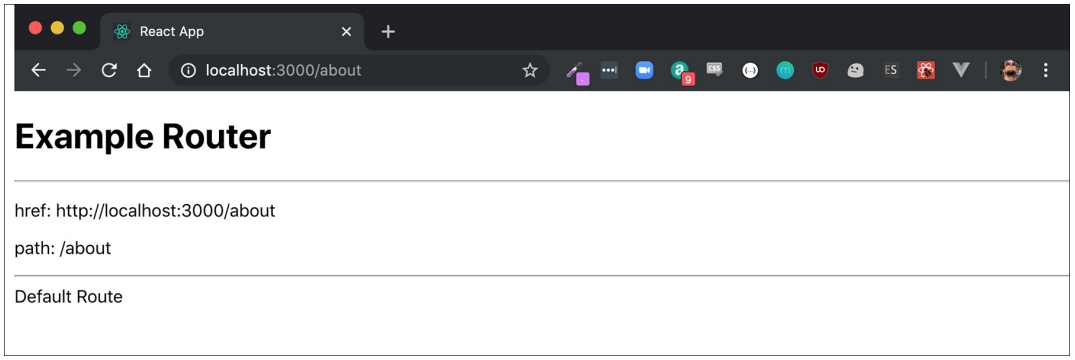

Figure 9.2: Showing Default Route

Now, let's implement a simple route. We will start off by adding one for **`/about`**, which will take us to an about page. The easiest way to do it is by adding a new functional About component.

8. Add an **`About`** functional component to our existing app:

```
const About = () => (
  <div className="About">
  <h1>About Page</h1>
  <hr />
  <p>Information about your app or who you are would go here.</p>
  </div>
);
```

9. Next, add an entry to the **`RouteTo`** function by adding another entry to our switch statement:

```
const RouteTo = path => {
  const paths = path
    .split('/')
    .map(p => p.toLowerCase())
    .slice(1);
  switch (paths[0]) {
    case 'about':
      return <About />;
```

```
        default:
          return <div className="Default">Default Route</div>;
      }
    };
```

> **NOTE**
>
> The **p** argument inside the **map** method is an ES6 arrow syntax for defining a function, where **p** is a temporary variable that gets declared within the scope of this function.

As you can see from the preceding code, a switch statement has been introduced, where the paths array has been passed as an argument. While looping through paths, when about is encountered, the **<About/>** component gets displayed.

10. Now, visit **/about** in your browser, where you will see the **About** route. If you visit another URL, you will instead see the **Default** route as shown below:

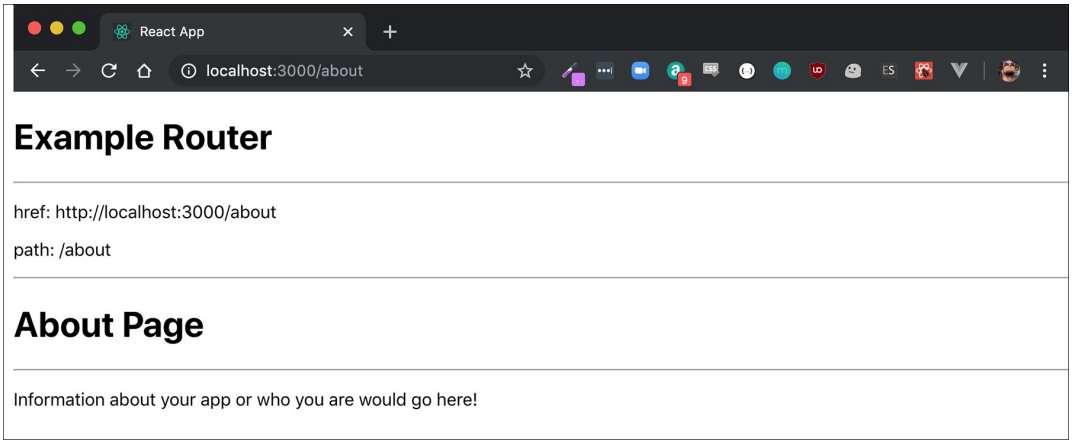

Figure 9.3: App showing the about page

Now that we understand the very basics of how React Router works behind the scenes, let's move on to using React Router. The importance of doing this yourself the first time is to understand what is going on under the hood. Once you demystify complicated concepts, you can learn them much faster.

BASICS OF REACT ROUTER

We have now looked at what we can accomplish through the browser's own built-in **`Location`** APIs. Imagine a library that is far more fully featured and built to accommodate the explicit purpose of making single-page applications way easier to navigate via browser mechanisms that the user is likely very comfortable with. The **`React Router`** library provides the functionalities that you require to make an application more interactive and intuitive. The library has gone through a few different revisions over the years, but the core of how to use it has remained similar across all iterations.

React Router provides several different built-in mechanisms for routing to components, handling default and error routes, and even more in-depth functionality such as dealing with authentication.

When we talk about React Router, a discussion of a few key components that will be imported and used while building the React Router project is necessary. So, let's get started:

- The first thing we need to import is **`BrowserRouter`**, which we will alias in our import statement as just Router. This is the top-level component inside which we need to wrap our routable code.
- Next, we will import **`Switch`**. You can think of the **`Switch`** component functioning in the same way as the **`switch`** statement used in the first exercise. This includes all the route decisions that need to be made while building an application.
- Finally, we will import **`Route`**. Think of **`Route`** as each of the case statements in the **`switch`** block from the first exercise. Each **`Route`** indicates a path to trap and render a particular component or bit of JSX code.

> **NOTE**
>
> The order of the **`Route`** components in our application is important. The program will start matching the routes will match from top to the bottom and will return the first route that is applicable. This means that if you have the default route (that is, the one without a pathname variable specified for it) above the about route, then you will never be able to reach the **`about`** route. Therefore, we have to be careful while setting the order of these components.

The best way to learn *React Router*, however, is to start building with it. So, let's jump right into our next exercise and start building a simple route-based app.

EXERCISE 9.02: IMPLEMENTING A SWITCH ROUTER

In this exercise, we will create a new app that mirrors the functionality we implemented in the previous exercise.

We will start by building our **`App`** component as a functional component and include the **`Homepage`** component first, and then, eventually, we will add the **`About`** component. We are going to use the order of **`Router -> Switch -> Route`** in order to switch between these components. Perform the following steps to achieve this:

1. Start off by creating a new React app, and call it router-example:

```
$ npx create-react-app router-example
```

2. Add React Router through the **`react-router-dom`** package using **`Yarn`**:

```
$ yarn add react-router-dom
```

3. Delete **`logo.svg`** and **`App.css`**, and then clear out the contents of **`App.js`**.

4. Add the **`About`** component from the previous exercise to this project. You can declare it inside of **`src/App.js`** for now:

```
import React from 'react';
const About = () => (
  <div className="About">
    <h1>About Page</h1>
    <hr />
    <p>Information about your app or who you are would go here.</p>
  </div>
);
export default About;
```

5. Build a quick **`Homepage`** component to represent our default route:

```
const Homepage = () => (
  <div className="Homepage">
  <h1>Homepage</h1>
  <hr />
  <p>This is our homepage.</p>
  </div>
);
```

6. First, add our **`import`** statements for React Router. We specifically need a few different **`import`** statements to account for how React Router works, as mentioned earlier:

```
import { BrowserRouter as Router, Switch, Route } from 'react-router-
dom';
```

7. Inside the **`App`** component, use the **`Switch`** router in order to switch between the **`Homepage`** and **`About`** components:

```
const App = () => (
  <Router>
   <Switch>
    <Route>
     <Homepage />
    </Route>
   </Switch>
  </Router>
);
export default App;
```

This should give us the following application to start, regardless of what URL we add:

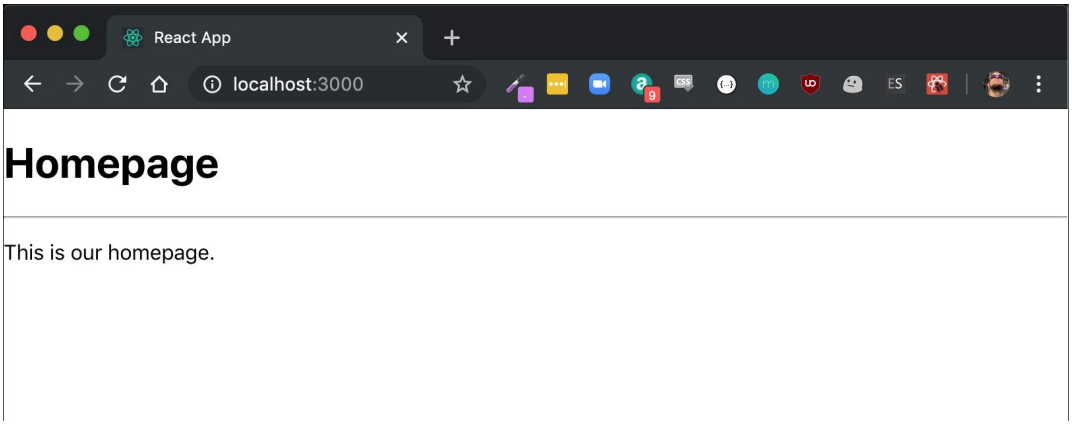

Figure 9.4: Homepage

8. Add an **`/about`** route that displays the **`About`** component we created earlier:

```
const App = () => (
  <Router>
  <Switch>
    <Route path="/about">
      <About />
    </Route>
    <Route>
        <Homepage />
    </Route>
  </Switch>
  </Router>
);
```

Here, we specify the path attribute for the route that sends the user to the **`About`** component and append the **`/about`** string in the URL Going to **`http://localhost:3000/about`** in our browser now will send us to the **`About`** component instead of the home page:

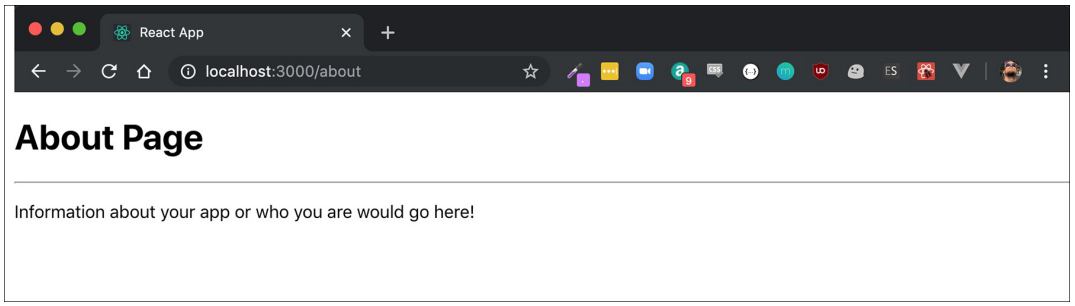

Figure 9.5: About Page

As you can see, the **`About`** page gets displayed when requested. You can see how easily we can use the **`Switch`** router in order to switch between the components.

> **NOTE**
>
> Routes are case-insensitive by default; so, if you went to **`http://localhost:3000/ABouT`**, it would still work and appropriately route you to the About component.

ADDING PARAMS TO YOUR ROUTES

Frequently, while working with routes, we might encounter a problem where we need to customize what is displayed on a page based on a little bit of information being passed into the page or path. A good example of this is with something like search pages, where you might have a URL such as **`https://search.com?q=search_term`**, where **`q`** is the URL parameter and **`search_term`** is the value of that parameter.

Now, if we want to customize the display result based on the **search_term** value, you will need to do a little more legwork than just the simple routing example we used before. First of all, you will need to include a new **import** statement from **react-router-dom** called **useLocation**. **useLocation** allows you to parse out more details about the current location in the browser and convert the data into variables. Using this, we can pull the query parameters out via the **useLocation** utility, specifically from the search property on **useLocation**. We can then build a new **URLSearchParams** object and use that to pull the specific parameter we want. For example, if we wanted to pull the **q** URL parameter out, we could do so via the following snippet:

```
import { useLocation } from 'react-router-dom';
const Search = () => {
  const query = new URLSearchParams(useLocation().search);
  const term = query.get('q');
  return (<div className="Search"></div>);
};
```

> **NOTE**
>
> In the preceding example, the call to **useLocation()**.search function happens from inside a React functional component, **Search**.

Let's now put this into practice by building a quick and small Search component using the preceding snippet of code.

EXERCISE 9.03: ADDING PARAMS TO ROUTES

In this exercise, we will create a **Search** component that will be designed to display a search results page. We will create a **results** list and design a search query. We will then append this query as a parameter to the URL. We will then match this with the items in the **results** list. In the case of a match with an item from the **results** list, we will display the matched items on the page. To do this, let's go through the following steps:

1. Start off by creating a new React app, and call it search:

```
$ npx create-react-app search
```

2. Add React Router through the **react-router-dom** package via **Yarn**:

```
$ yarn add react-router-dom
```

3. Delete **`logo.svg`** and **`App.css`**, and then clear out the contents of **`App.js`**.
4. Add our standard React Router **`import`** statements at the top:

```
import {
  BrowserRouter as Router,
  Switch,
  Route,
  useLocation
} from 'react-router-dom';
```

5. Add a quick **`Homepage`** component to represent our default route:

```
const Homepage = () => (
  <div className="Homepage">
    <h1>Homepage</h1>
    <hr />
    <p>This is our homepage.</p>
  </div>
);
```

6. Next, let's set up our main **`App`** component with all of the React Router code. We don't have a **`Search`** component yet, but We will be adding that next, so we're going to add the code for that now:

```
const App = () => (
  <Router>
    <Switch>
    <Route path="/search">
      <Search />
    </Route>
    <Route>
      <Homepage />
    </Route>
    </Switch>
  </Router>
);
```

7. Now, let's start building the **`Search`** component. We will start with a quick shell for a **`Search`** component and then flesh out the meat of it:

```
const Search = props => {
  const term = '';
  return (
    <div className="Search">
      <h1>Search Page</h1>
      <hr />
      Found results for {term}:
      <ul>
      <li>No results.</li>
    </ul>
    </div>
  );
};
```

8. Add a list of items that we can search across at the top of our project:

```
const Items = [
  'Lorem Ipsum',
  'Ipsum Dipsum',
  'Foo Bar',
  'A little black cat',
  'A lazy fox',
  'A jumping dog'
];
```

9. Next, add a search function that will perform the search across that list of items. This should take in term as the only argument, and if no search term is passed in, the entire list is returned. From there, we will change each list item to lowercase as we traverse through it and filter it down to only the items where the search term appears:

```
const doSearch = term => {
  if (!term) {
    return Items;
  }
  return Items.filter(
    item => item.toLowerCase().indexOf(term.toLowerCase()) !== -1
  );
};
```

10. Finally, we will flesh out the **`Search`** functionality using the code snippet we provided before this exercise, and map over each of the returned items from the search code in an enclosed **`<li>`** tag:

```
const Search = props => {
  const query = new URLSearchParams(useLocation().search);
  const term = query.get('q');
  const returned = doSearch(term);
  return (
  <div className="Search">
    <h1>Search Page</h1>
    <hr />
    Found results for {term}:
    <ul>
      {returned.map(t => (
        <li key={t}>{t}</li>
      ))}
    </ul>
  </div>
 );
};
```

11. Now, if we search for something such as **`ipsum`**, we should expect to see only the appropriate search terms returned to us:

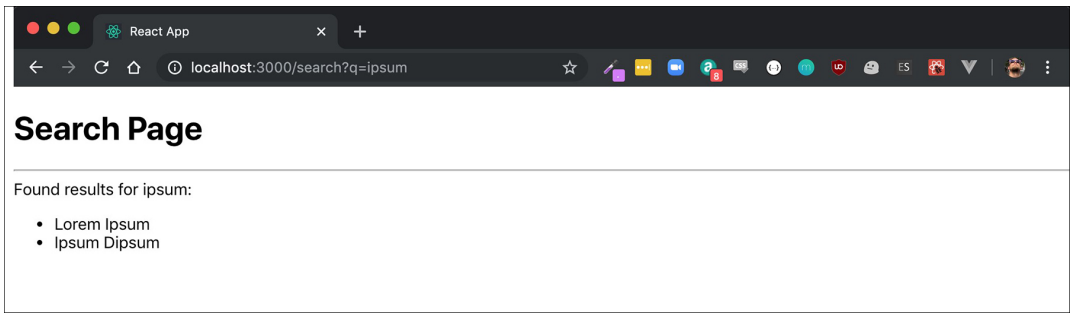

Figure 9.6: Search Page displayed

As you can see, when we add a parameter in the search query, **`q=ipsum`**, that matches an item from the results list created in *Step 8* of the preceding exercise, the page displays the items containing the word ipsum. (Note that the casing of the word is ignored.)

Let's move on to the next section, in which We will design components for instances where we might encounter a resource not found error when a user visits our site.

ADDING PAGE NOT FOUND FOR ROUTES

Sometimes, you will want to be able to display something meaningful to the user when they attempt to visit an incorrect URL for your site, instead of just showing the **`404`** error. This could be something funny, on-brand, or something memorable. However, in order to do something like this, you need to be able to catch those missed pages and display a **`NotFound`** component. One of the first things you will need to change is the previous **`default`** route that we had set up, which did not specify a pathname.

Try setting up your routes like this:

```
<Router>
<Switch>
  <Route path="/search">
  <Search />
</Route>
<Route>
  <Homepage />
</Route>
</Switch>
</Router>
```

Here, you can see that there is no room for anything to fall through when someone enters a bad route when a component is not found. It is often better to be more explicit while working with routes:

```
<Router>
<Switch>
  <Route exact path="/">
    <Homepage />
  </Route>
  <Route path="/search">
    <Search />
  </Route>
</Switch>
</Router>
```

Notice that, in the preceding example, the path for the Homepage component is now listed above the Search component and has a few new properties attached to it. Specifically, we have added the following properties:

- **`exact`**, which has a default value of true when specified this way.
- **`path`**, which points to **`/`**. This means it will **ONLY** match when the path is **ONLY `/`** with no variation. This prevents scenarios where it would catch everything else. Now we can add a **`404-display`** component.

We can create an explicit **`catch-all`** route by using **`path=`** to allow it to catch any other routes entered. Let's put this to the test in the next exercise.

EXERCISE 9.04: ADDING A NOTFOUND ROUTE

In this exercise, we will create a **`NotFound`** route in case a component does not have defined routes. This will allow us to display a custom **`404`** component. Let's go through the following steps:

1. Start off by creating a new React app called **`notfound`**:

```
$ npx create-react-app notfound
```

2. Add React Router from the **`react-router-dom`** package via **`Yarn`**:

```
 $ yarn add react-router-dom
```

3. Delete **`logo.svg`** and **`App.css`**, and then clear out the contents of **`App.js`**.
4. Add our standard React Router **`import`** statements at the top:

```
import { BrowserRouter as Router, Switch, Route } from 'react-router-
dom';
```

5. Add a quick **`Homepage`** component (this is from our previous exercise) to represent our default route:

```
const Homepage = () => (
  <div className="Homepage">
    <h1>Homepage</h1>
    <hr />
    <p>This is our homepage.</p>
  </div>
);
```

6. Additionally, add the **`About`** component, also from a previous exercise, since we should have a few different components to attempt to route to in order to make this more meaningful:

```
const About = () => (
  <div className="About">
    <h1>About Page</h1>
    <hr />
    <p>Information about your app or who you are would go here.</p>
  </div>
);
```

7. Now set up the routes. We will start off with our **`Homepage`** route (remember the exact **`path=/`** props) and the **`About`** route:

```
const App = () => (
  <Router>
    <Switch>
    <Route exact path="/">
      <Homepage />
    </Route>
    <Route path="/about">
      <About />
    </Route>
    </Switch>
  </Router>
);
```

8. Finally, let's add our **`NotFound`** route by adding a new route with a path of **`*`** at the bottom of our **`Switch`** route:

```
const App = () => (
  <Router>
    <Switch>
      <Route exact path="/">
      <Homepage />
      </Route>
      <Route path="/about">
      <About />
      </Route>
    <Route path="*">
```

```
          <h1>404 - Component Not Found</h1>
            <a href="/">Return Home</a>
        </Route>
      </Switch>
      </Router>
    );
```

9. Reload the page and try to create a URL with a string that will not match any values from the preceding list (also remember to test the two matched routes), and you should see the **`404`** page show up instead of the **`Homepage`** component:

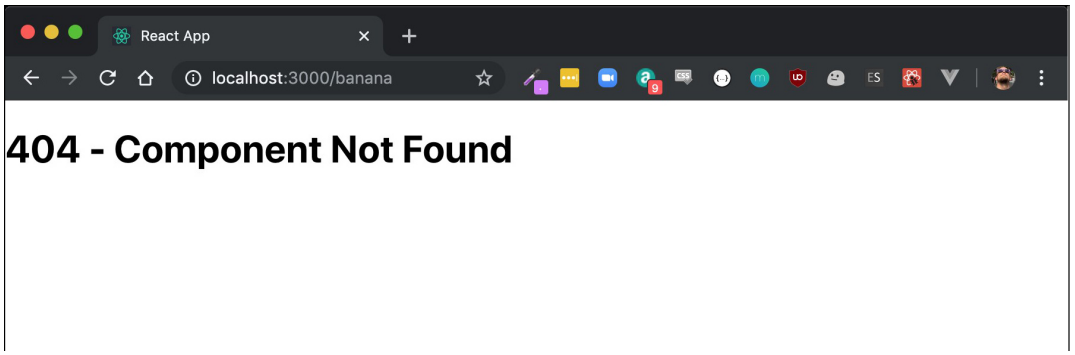

Figure 9.7: The NotFound component

The **`NotFound`** component has displayed successfully. Now, let's move on to the **`Nesting Routes`** section.

NESTING ROUTES

Occasionally, you will need to create routes that have the same parent URLs but different sub URLs. For example, you might visit an online store, which might have a full URL like this:

`https://somesite.com/store`

When you click on an item in a store, they might be listed like this:

`https://somesite.com/store/item/12345`

Here, the **`12345`** value at the end of the URL indicates the ID of the item you are looking at on that online store. The site will want you to still see the information that is pertinent to you in the context of the store, so it will keep the parent route as store, and the child route as the specific item (the **`/item/12345`** bit in the URL). This is an example of a nested route.

In React Router, dealing with nesting routes is a little bit more complicated than simple routes. The easiest way to handle it, even though it is slightly more verbose, is to place new **`Router -> Switch -> Route`** trees inside of your React components. Working with our previous exercise, if we wanted to add a nested **`Contact`** route to our **`About`** route, we would change our **`About`** component from a simple example to one that embeds the React Router components into the component itself.

The first thing you need to do is to wrap your **`About`** component in a **`Router`** component. Next, add a **`Switch`** statement inside of the component, followed by routes for each of the levels of nesting you want to include. We will also want to explore how we can use React Router's built-in **`Link`** component. The **`Link`** component acts like standard anchor tags in HTML, but it also helps you to preload necessary data for the component you are navigating to. It also does a better job of intercepting the navigation request in the browser so that only the component is refreshed instead of the entire page.

We also need to use something from React Router called **`useRouteMatch()`**. **`useRouteMatch()`** will return to us specifically two pieces of information that we need: the current path and the current full URL. We will need both to construct our nested paths and the links of the current path to avoid a situation where we change the about top-level path to something else and then have to go in and find every single nested route and fix those too.

Let's explore this further in our next exercise.

EXERCISE 9.05: NESTED ROUTES AND THE LINK COMPONENT

In this exercise, while building up more complicated navigation in your React apps, it is important to provide a means of navigation to move among the components in a simple way. We can use the **`Link`** component to provide React Router-specific navigation. You can copy the same project from the previous exercise, though, the steps to start a new project will be included here just in case. Let's get started:

1. Start off by creating a new React app and call it nested:

```
$ npx create-react-app nested
```

2. Add React Router from the **`react-router-dom`** package via Yarn:

```
$ yarn add react-router-dom
```

3. Delete **`logo.svg`** and **`App.css`**, and then clear out the contents of **`App.js`**.

4. Add our standard React Router **`import`** statements at the top, with the additional inclusion of **`Link`** and **`useRouteMatch`**, which we will use to construct our nested paths (we will talk about this a lot more in a later step):

```
import { BrowserRouter as Router, Switch, Route, Link, useRouteMatch
} from 'react-router-dom';
```

5. Add the **`Homepage`** component, the **`About`** component, and the **`App`** component with routing for each. The code for all three is included here:

`App.js`

```
15  <div className="About">
16  <h1>About Page</h1>
17  <hr />
18  <p>Information about your app or who you are would go here.</p>
…
55  const Homepage = () => (
56  <div className="Homepage">
57    <h1>Homepage</h1>
58    <hr />
59    <p>This is our homepage.</p>
60  </div>
61  );
```

`The complete code can be found here:` https://packt.live/2LqH4sp

6. Next, we are going to add a persistent navigation component called **`Navbar`**, which will contain our **`Link`** components. **`Link`** is just a wrapper around an **`anchor`** tag and involves React Router functionality. This allows for different routes to be navigated to within the context of your app, instead of via the browser. Each **`Link`** component we use will just have a property called to on it that tells us where the **`Link`** component needs to be pointed to:

```
const Navbar = () => (
  <div className="Navbar">
    <Link to="/">Home</Link>
     
    <Link to="/about">About</Link>
  </div>
);
```

7. Now, add that **`Navbar`** to our **`App`** component. It specifically needs to be inside of a router, or it will not work, so you need to place it above the **`Switch`** statement to make it persistent across each of the pages:

```
const App = () => (
  <div className="App">
    <Router>
      <Navbar />
      <Switch>
        <Route exact path="/">
        <Homepage />
      </Route>
      <Route path="/about">
        <About />
      </Route>
      <Route path="*">
        <h1>404 - Component Not Found</h1>
      </Route>
      </Switch>
    </Router>
  </div>
);
```

Our app should now resemble the following screenshot:

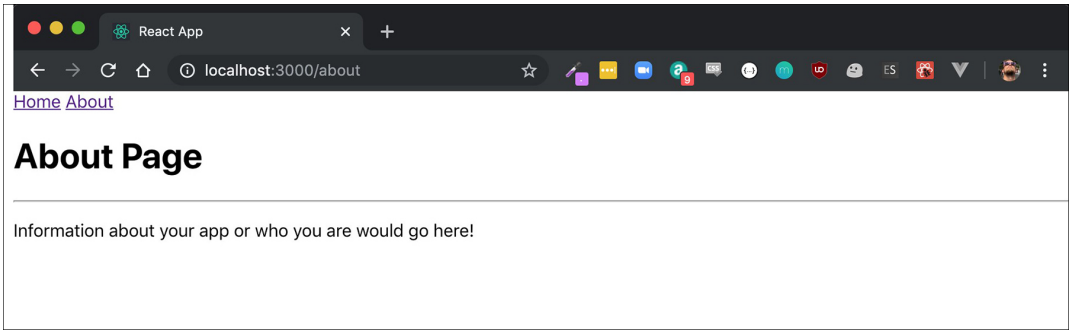

Figure 9.8: The About Page component

8. Let's start building our nested components now. We will add two new components: **`Bio`** and **`Contact`**. The **`Bio`** component will include a short biography about yourself and the **`Contact`** component will include the best way for someone to reach you:

```
const Bio = () => (
  <div className="Bio">
    <h2>Bio</h2>
    <hr />
    <p>I'm a pretty cool person.</p>
  </div>
);

const Contact = () => (
  <div className="Contact">
    <h2>Contact Me</h2>
    <hr />
    <p>Send me an email at test@test.com.</p>
  </div>
);
```

9. Now we are going to set up nested routing in our **`About`** component. The general design for this is that we want our **`/aboutpath`** to have two nested paths: **`/about/bio`** and **`/about/contact`**. These should display the **`About`** component and then additionally one of the nested routes if we visit one.
To build a nested route, we just place a **`Router -> Switch -> Route`** hierarchy into our component. We will start off by wrapping our **`About`** component inside of a **`Router`** tag:

```
const About = () => {
  return (
    <Router>
      <div className="About">
        <h1>About Page</h1>
        <hr />
        <p>Information about your app or who you are would go here.</
p>
      </div>
    </Router>
  );
};
```

10. We need to take advantage of the **useRouteMatch** import that we declared earlier. We will add this line to the top of our **About** component:

```
const { path, url } = useRouteMatch();
```

11. Now we can start building our custom routes. We will add a new **Switch** statement with an **<hr>** tag preceding it. Everything under the **<hr>** tag will represent the content of our nested routes. We will want to use the **path** variable we created before to construct our paths. Place this code under the **<p>** tag in your **About** component:

```
    <hr />
    <Switch>
      <Route path={`${path}/contact`}>
        <Contact />
      </Route>
      <Route path={`${path}/bio`}>
        <Bio />
      </Route>
    </Switch>
```

12. Next, let's create some sub-navigation for the **About** component to allow the user to easily move between each of those nested routes. This time, we will use the **url** variable we created from **useRouteMatch**, and we will place it above the **Switch** statement but below the content of the **About** component:

```
    <hr />
    <Link to={`${url}`}>About Home</Link>
     
    <Link to={`${url}/contact`}>Contact</Link>
     
    <Link to={`${url}/bio`}>Bio</Link>
```

This should give us a full **About** component with the following code:

```
const About = () => {
  const { path, url } = useRouteMatch();
  return (
    <Router>
      <div className="About">
        <h1>About Page</h1>
        <hr />
        <p>Information about your app or who you are would go here.</
p>
```

```
            <hr />
            <Link to={`${url}`}>About Home</Link>
             
            <Link to={`${url}/contact`}>Contact</Link>
             
            <Link to={`${url}/bio`}>Bio</Link>
            <hr />
            <Switch>
            <Route path={`${path}/contact`}>
              <Contact />
            </Route>
              <Route path={`${path}/bio`}>
                <Bio />
            </Route>
          </Switch>
        </div>
        </Router>
    );
  };
```

And a resulting component that should resemble the following screenshot:

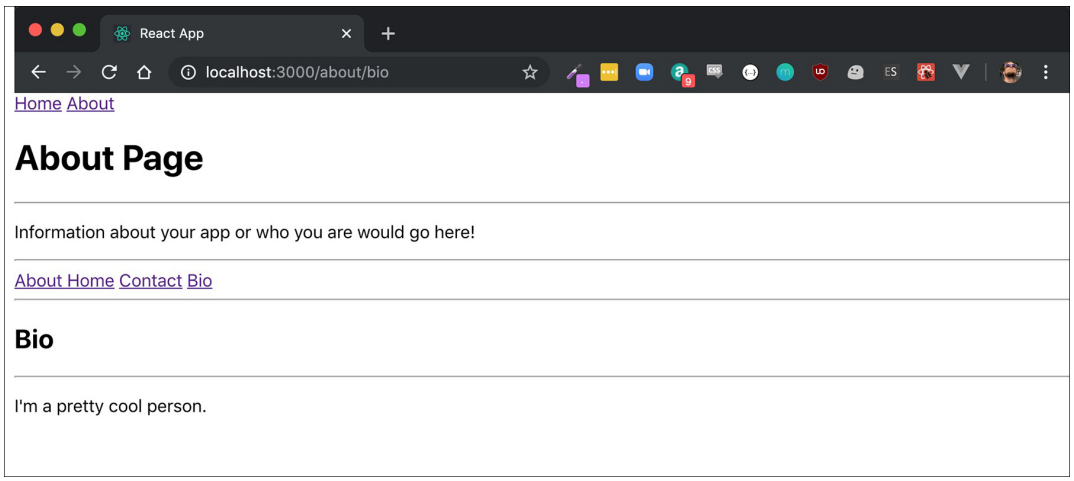

Figure 9.9: The About Page component

As you can see, the **`About`** and **`Bio`** components have been displayed successfully. We are now going to put everything we have learned so far together and perform the following activity.

ACTIVITY 9.01: CREATING AN E-COMMERCE APPLICATION

The goal of this activity is to construct a simple e-commerce application where we will implement several links or routes that could be used to navigate to different sections. We will apply nested **`404/component not found`** routes and URL queries.

In this **`MyStore`** we will have a few items displayed. The store app should contain a header and the list of items that are up for sale. You can either delegate the display of each item to another component or write them into the Store component. The Store component has a couple of requirements:

- If no item id is specified, display all of the items using the minimized item display.
- If an item id is specified and the item exists, display the larger version of that item. You will need to research and use a parameterized URL for this.
- If an item id is specified and the item does not exist, display an item **`NotFound`** message.
- If a bad nested route is specified, display an item **`NotFound`** message.

The following steps will help you to complete the activity:

1. Create a new React application using the **`npx`** command.
2. Inside **`App.js`**, include the necessary **`import`** statements: **`Router`**, **`Switch`**, **`Route`**, **`Link`**, and **`useRouteMatch`**.
3. Create a basic functional component, **`MyStore`**, and add to it the inventory items, **`Shoes`**, **`Backpack`**, and **`TravelMug`**.
4. Build a Store functional component and then, using **`useRouteMatch`**, build routes for each inventory item.
5. Create a **`HomePage`** component and a navigation bar inside each component.
6. Create nested routes for the **`Store`** component.
7. Create a **`NotFound`** component in case the item was not found in the store.
8. Create a functional **`Item`** component to show the details for each item.

The output should be as follows:

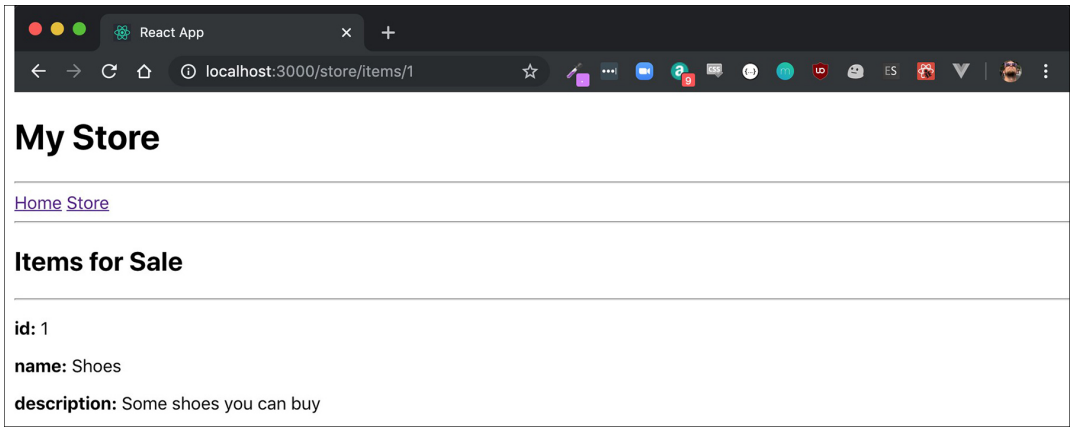

Figure 9.10: The MyStore app

You have successfully completed the activity.

React Router is a very powerful tool to have in your React toolbox. With it, you can create incredibly seamless and snappy web applications that will feel like they are practically native applications. Notice that every time you navigate around using the navigation bar created via **`Link`** components, the entire browser doesn't redraw the page for each different navigation element; instead, only the parts of the tree that need to be re-rendered will be re-rendered and the rest will remain the same.

This gives the user the feeling of navigation that feels natural, which is one of the best things you can give to your app. The more seamless everything feels, the more likely the user will be happy using it and recommend it to others as well. Putting this together with the techniques you have learned over the course of this chapter will give you the ability to construct some truly fantastic experiences on the web. In the next chapter, we will look into React Router in more detail.

> **NOTE**
>
> The solution to this activity can be found on page 672.

SUMMARY

Over the course of this chapter, we explored the basics of React Router and the various Router implementation strategies that you can take when you are just getting started. It is important to have a better means of natural navigation via a browser; this allows for a clean separation of components into their logical use cases and allows the user to use their back button in the browser. It is a better, cleaner separation for users and for developers, and now we have a great working knowledge not just of the library itself, but also a little insight into how it works behind the scenes as well.

In later chapters, we will explore React Router in greater detail, showcasing some of the more advanced functionality. We will be able to take our implementations with React Router from basic applications to the next level and build upon the foundations that we have established in this chapter.

10

ADVANCED ROUTING TECHNIQUES: SPECIAL CASES

OVERVIEW

This chapter will provide you with the knowledge on how to handle nested routes in React. You will be able to optimize your code by effectively creating nested `404` pages, URL parameters, and protected routes. This chapter will equip you with more advanced tools and techniques that are required to implement special use cases of routing, such as page not found, restricted routes, and nesting.

INTRODUCTION

In the previous chapter, we have introduced React Router and saw some practical examples of how to construct simple routes. We now understand what routing is and why React Router v4 is a solid choice when it comes to defining routes, and how to use links, exact matches, and transitions.

There are situations, however, that might require you to implement additional functionalities. For example, when we navigate to a page using deep links, links that can be used navigate to a specific indexed page in our application, we may need to pass a few parameters, such as tokens or IDs. In this chapter, we will learn how to handle URL parameters and how to retrieve them via route props. We will also learn how to restrict access to routes, either by preventing navigating out of the current view to prevent losing existing form data or handling and storing navigation into the view layer if the user has no permissions or is not authenticated. This is possible with the use of Higher-Order Components (HOCs) and some business logic rules.

When we complete this chapter, we will have a complete picture of how to use advanced routing techniques. We will complete some practical exercises for each technique in detail. We will practice our knowledge by implementing a complex routing solution with navigational rules, nested components, and more.

Let's get started by looking at a few scenarios that we might encounter while implementing React Router in a React application.

REACT ROUTER SPECIAL CASES

Let's look at a few special cases where we'll be using advanced routing techniques.

PASSING URL PARAMETERS BETWEEN ROUTES

URL parameters are variables that we define in the routes and which can take a special form. We will use those parameters to handle cases where we need to pass information into the route via the URL and to create SEO-friendly links such as the following:

```
/recipes/stews/meat
```

Another example of this is as follows:

```
/users/theo/profile
```

To define URL variables, we need to prepend them with a colon (**:**), followed by the name of the parameter. For example, the following strings are examples of valid parameters:

```
:id, :username, :email, :type, :search
```

However, the following strings are not valid parameters:

```
?id, -username, &email, type, *search*
```

When we use parameters like this, we just need a valid link to match that route. We can access the matched parameter by name using the **`useParams()`** function that's exposed by the library. This function returns an object containing the *key/value* pairs of the URL parameters of the current page.

Let's look at an exercise to practice the usage of URL parameters:

EXERCISE 10.01: CREATING URL PARAMETERS

In this exercise, we are going to create a new react app. Then, we will create a few routes defining the paths for each. We will practice the URL parameters. Perform the following steps:

1. Create a new React app:

```
npx create-react-app url-parameters
```

2. Go to the **`src/App.js`** file and delete **`logo.svg`**, **`App.css`**, and clear out the contents of **`App.js`**.

3. Import the required modules:

```
yarn add react-router-dom
```

4. Add the imports in **`App.js`**:

```
import React from 'react';
import {
  BrowserRouter as Router,
  Link,
  Route,
  useParams,
  useLocation,
  Switch,
} from 'react-router-dom';
```

5. Create a functional component called **App**:

```
function App()
{}
```

6. Inside **App**, create the links **Page1**, **Page2**, **Page3**, and **Page4**:

App.js

```
13  return (
…
19          <li>
20              <Link to="/">Page1</Link>
21            </li>
22            <li>
23              <Link to="/name">Page2</Link>
24            </li>
25            <li>
26              <Link to="/path/Theo">Page3</Link>
27            </li>
```

The complete code of this step can be found here: https://packt.live/30W7WJU

7. Use the **Switch** component as follows:

App.js

```
36        <Switch>
37          <Route exact path="/">
38            <Page1 />
39          </Route>
40          <Route path="/:id">
41            <Page2 />
42          </Route>
43          <Route path="/path/:name">
44            <Page3 />
```

The complete code of this step can be found here: https://packt.live/2YdQn6j

8. Inside **App**, create separate components for each links:

```
function Page1() {
  return <h3>Page1</h3>;
}
function Page2() {
  let { id } = useParams();
  return <h3>ID: {id}</h3>;
}
function Page3() {
  let { name } = useParams();
  return <h3>Hello { name ? `${name}` : 'stranger'}</h3>;
}
function Page4() {
```

```
    let { first, last } = useParams();
    return <h3>First: {first}, Last: {last}</h3>;
}
export default App;
```

The output is as follows:

Figure 10.1: Links created for separate pages

Let's understand what is happening in the preceding code in more detail:

- **`<Route exact path="/">`** is an exact route, so whenever we navigate to the base page, it will render the **`Page1`** component and there are no page parameters.
- However, when we click on the **`Page2`** link, which has a route definition of <Route path="/:id">, it will match the name parameter, so using the **`useParams`** call will return an object with a **`name`** property set.
- We will see the **`Page2`** component render as **`ID`**: name.
- Next, let's take a look at the **`Page3`** link. We think this should match **`<Route path="/path/:name">`** because it starts with the same prefix, right? Well, it turns out that it doesn't work like that. When we click on either the **`Page3`**, **`Page3 Missing Param`**, or **`Page4`** links, we'll notice that the **`Page2`** component renders instead. The reason why this happens is that React Router follows a top-down approach and when it matches the first route string, it does not continue looking. In our example, the route match is **`/:id`** and the actual string to match is **`/path/Theo`**, so it will match the path as the name parameter and render the **`Page2`** component.
- To overcome this issue, we just need to use the exact modifier on the second route like this:

```
<Route exact path="/:id">
```

- This way, we instruct React Router to only render this route if it exactly matches that path.
- Next, let's check the last two links. Once we click on the second to last link, the **`Page2`** component renders because we did not pass a parameter in the link. This will match the following path:

```
<Route path="/:id">
```

- Now, for the last link, we'll notice that it will render the **`Page3`** component. Again, this is for the same reasons that the library chose to render the **`Page2`** component when we didn't specify an exact match. So, we need to use the exact modifier on the second to last route like so:

```
<Route exact path="/path/:name">
```

By implementing these changes, we made all the routes navigate to the correct paths when we click on them.

URL parameters should not be confused with query parameters, which are a way to pass custom identifiers and variables in a route path. The main difference they have between query parameters is that they are key-value-based. For example, let's look at the following relative path:

```
/search?q=animals&c=cats
```

This specifies two query strings, one as q=animals and the other as **`c=cats`**.

Anything after the question mark character denotes a key-value list of query parameters, and React Router will not match them against any route.

In order to parse the query parameters from React Router, we can use the **`useLocation().search`** value, which returns the whole query string.

In our case, this would be **`?q=animals&c=cats`** string. As you may have noticed, it's not very useful like this. Fortunately for us, we can use the **`URLSearchParams`** interface by passing that string in the constructor:

```
const params = new URLSearchParams(useLocation().search)
```

Now, we can access the value of each parameter by key:

```
params.get('q') // "animals"
params.get('c') // "cats"
```

If we have multiple keys in the same search string, we need to use the **`getAll()`** method:

```
const params = new URLSearchParams("?q=cats&q=dogs")
params.getAll('q') // (2) ["cats", "dogs"]
```

Now that we have seen how URL parameters work, we will learn how to handle missing or unknown routes.

MATCHING UNKNOWN ROUTES WITH 404 PAGES

So far, we know that when we have a list of routes that we need to render exclusively, we need to use the **`<Switch />`** component so that whenever we match a location route, it will render only one component at a time.

EXERCISE 10.02: CREATING UNKNOWN ROUTES

In this exercise, we are going to create and practice the usage of unknown routes.

1. Create a new App:

```
npx create-react-app unknown-routes
```

2. Go to the **`src/App.js`** file and delete **`logo.svg`**, **`App.css`**, and clear out the contents of **`App.js`**.

3. Import the required modules:

```
yarn add react-router-dom
```

4. Add the necessary imports in **`App.js`**:

```
import React from 'react';
import {
  BrowserRouter as Router,
  Link,
  Route,
  useLocation,
  Switch,
  Redirect
} from 'react-router-dom';
```

5. Create a functional component **`App`**:

```
function App() {}
```

6. Inside **App**, create links to **Page1**, **Page2**, **Page3**, and **Page4** components:

App.js

```
18    <ul>
19      <li>
20        <Link to="/">Page1</Link>
21      </li>
22      <li>
23        <Link to="/page2">Page2</Link>
24      </li>
25      <li>
26        <Link to="/Page3?param=123">Page3</Link>
27      </li>
28      <li>
29        <Link to="/page4">Page4</Link>
30      </li>
31    </ul>
```

The complete code can be found here: https://packt.live/3bzavD8

7. Inside **App.js**, create separate functional components for each links:

```
function Page1() {
  return <h3>Page1</h3>;
}
function Page2() {
  return <h3>Page2</h3>;
}
function Page3() {
  let param = new URLSearchParams(useLocation().search);
  return <h3>Page3 { param ? `${param}` : ''}</h3>;
}
```

8. Create a **NoMatch** component in case a page was not found:

```
function NoMatch() {
  return <h3>404 Sorry!</h3>;
}
export default App;
```

We have a **NoMatch** component that will render in case of route not found. The following is a screenshot of the browser when this happens:

Figure 10.2: The page not found routes

In the preceding example, we define three routes with a unique page. Under the <Switch /> component, they will render individually for each unique path.

Note that we can use **`<Route path="*">`** or **`<Route >`** for the **`404`** path and that it always needs to be the last in order of the defined routes. Let's see what happens if we place it somewhere in-between the routes:

```
<Switch>
  <Route exact path="/">
    <Page1 />
  </Route>
  <Route path="/Page2">
    <Page2 />
  </Route>
  <Route>
    <NoMatch />
  </Route>
  <Route path="/Page3">
    <Page3 />
  </Route>
</Switch>
```

Here, the first two pages will render correctly, but **`Page3`** will render as **`NoMatch`** because the catch all rule precedes the **`Page3`** path.

The catch all rule will also work when we define nested routes. For example, if a unique path to a nested route is not found, then it will match any nested catch all rules first before matching a higher-level route.

There is also another issue. When we click on the **`Page4`** link, we can see that the page URL changes to **`/Page4`**. Sometimes, though, we want to redirect the user to a **`404`** path for the unknown route. In that case, we can assign a path for the **`404`** page and use a **`<Redirect />`** component to navigate to that route:

```
<Route path="/404">
  <NoMatch />
</Route>
<Redirect to="/404" />
```

This way is a little bit more flexible as it allows us to change the **`404`** page in one place if we have lots of nested routes that reference the same **`NoMatch`** component.

Speaking of nested routes, let's look at how we can use multiple hierarchies of routes within a component.

RENDERING NESTED ROUTES

There a lot of real-world use cases where we must render multiple components on each page. What's even more complicated is that many of those components may not change when we navigate from one route to another. These components can be nested within each other to form complex hierarchies. For example, we have the following routes:

```
/user/theo/profile
/user/theo/settings
```

Instead of defining two routes for **`/user/:name/profile`** and **`/user/:name/settings`**, we just define one for **`/user/:name`** and inside the matching component, we define more for profile, settings, or anything else.

Another example can be seen in the following illustration:

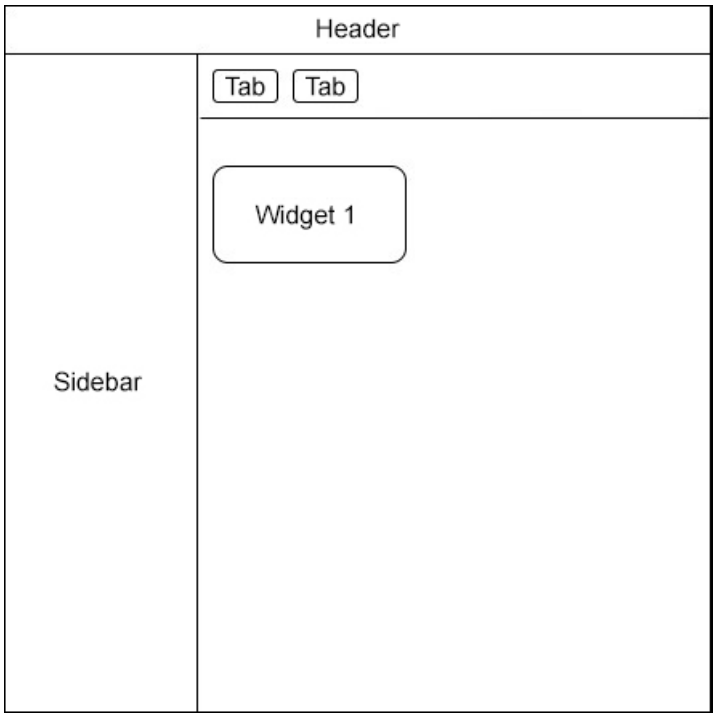

Figure 10.3: Web page

Here, we have a *Sidebar*, a *Header*, a *Tab* Holder with tabs, and a content area to display a widget. This could be a dashboard in a traditional web application. For some routes, for example, the initial login page, we wouldn't need to render the Sidebar or Header components, but for the main dashboard view and subsequent view, we always need them to render on those positions.

In addition, depending on the current tab component we click, a different Widget will be displayed, whether this is a map component or a Graph view. In that case, we only need to highlight the active tab and the selected Widget. We should be able to reconstruct this view based on a unique URL path, for example:

```
/dashboard/widgets/Widget1
```

The following is an alternative:

```
/dashboard/widgets/Widget2
```

To achieve this with React Router, since the routes are React components themselves, we can use them as the children of existing routes. Once a route is a child of another route, it will match itself when the parent route is matched.

Using nested routes is the recommended approach for reusing components between pages, preventing unnecessary re-renders and allowing code-splitting to work more effectively. The aim of this is to understand how to define and create nested routes in React Router.

Now, let's look at a practical exercise of how to define nested routes based on the dashboard mockup that we have.

EXERCISE 10.03: CREATING NESTED ROUTES

In this exercise, we are going to create nested routes. Perform the following steps to complete this exercise:

1. Create a new create -react- app:

```
npx create-react-app nested-routes
```

2. Go to the **`src/App.js`** file and delete **`logo.svg`** and **`App.css`** and clear out the contents of **`App.js`**.
3. Import the required modules:

```
yarn add react-router-dom
```

4. Add the following top-level routes. The **`Home`** component should render a basic string.
5. Add links to the **`Home`** and **`Dashboard`** components:

```
import React from 'react';
import {
  BrowserRouter as Router,
  Switch,
  Route,
  Link,
  useParams,
  useRouteMatch
```

```
} from "react-router-dom";


function App() {
  return (
    <div className="App">
      <Router>
        <div className="navbar">
          <Link to="/">Home</Link>
          <Link to="/dashboard">Dashboard</Link>
        </div>

          <Switch>
            <Route exact path="/">
              <Home />
            </Route>
            <Route path="/dashboard">
              <Dashboard />
            </Route>
          </Switch>
      </Router>
    </div>
  );
}
```

The **`Dashboard`** component should render two parts: a **`sidebar`** and a **`main`** content area.

6. Create a **`sidebar`** element:

```
<div className="sidebar">
        Sidebar
</div>
```

7. Create a main content area. This area should have also a **`Route`** component, like so:

```
    <Switch>
    <Route exact path={path}>
        <h3>Please select a widget.</h3>
      </Route>
```

```
          <Route path={`${path}/:widgetName`}>
            <Widget />
          </Route>
        </Switch>
```

8. Define the **`Widget`** component as follows:

```
function Widget() {
  let { widgetName } = useParams();

  return (
    <div>
      <h3>Widget: {widgetName}</h3>
    </div>
  );
}
```

9. Add a link for widgets:

```
<Link to={`${url}/map`}>Map</Link>
```

10. Add the following CSS so that you have a basic style for the page. An example CSS file is included in the source code that accompanies this chapter (https://packt.live/2y5B714).

    The complete code is as follows:

`App.js`

```
1  import React from 'react';
2  import {
3  BrowserRouter as Router,
4  Switch,
5  Route,
6  Link,
7  useParams,
8  useRouteMatch
9  } from "react-router-dom";
10  import './App.css';
11
12  function App() {
13  return (
14    <div className="App">
15      <Router>
16        <div className="navbar">
17          <Link to="/">Home</Link>
```

`The complete code can be found at:` https://packt.live/3bpNZwA

11. We imported an **`App.css`** file that contains common styles for the widgets and the dashboard elements. The code for this style is listed in the project repository files: https://packt.live/2y5B714.

    The output is as follows:

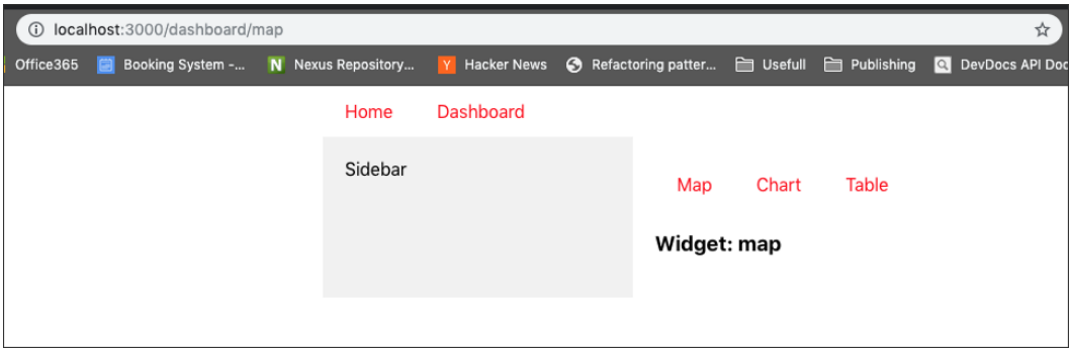

Figure 10.4: Final output

The preceding page will initially render the home page. When we click on the **`Dashboard`** link, we will see a **`sidebar`** component and some tab links for each widget. Then, when we click on each widget, the respective widget parameter name is passed to the **`Widget`** component. The following is a screenshot of that page:

Figure 10.5: Widget map

Here, we use the **`useRouteMatch`** hook, which will attempt to match the current URL in the same way that a **`<Route>`** can, though it will not render anything.

By using the **`useRouteMatch`** hook, we can extract the current path and the base URL from the returning match object and use it to define new routes.

React Router does not limit how many nested routes we can have. As long as the path uniquely matches the route component, we can have deeply nested trees. This is really useful when we need to expose deep links in our application where we need to have a unique link that will reconstruct a tree of components.

Now that we have seen how nested routes work, we will continue by looking at protected routes.

PROTECTING ROUTES

When we implement business requirements, often, we are asked to enforce conditional rules in regard to navigating to or from specific states. For example, when we have a form where the user has filled some fields in and then clicks on a navigational link to another page, we would like to tell the user that they will lose their current form data if they do. Additionally, we will come across cases where we want to prevent the user from navigating to certain pages until a condition is fulfilled; for example, the user needs to be logged in first to prevent unauthenticated page visits.

In general, we would like to control inbound and outbound transitions of the routes under certain conditions. Using React Router, there are a couple of ways we can do this. Let's explore the two different cases.

PREVENTING OUTBOUND TRANSITIONS

Preventing a user from navigating away from a page should be done in special cases as it may harm the UX if this is done inappropriately. For example, when a user is in a hurry and tries to click to another page and we present a prompt, then we need to be absolutely sure that this is crucial. Ideally, in those cases, we should not expose any links that navigate away from the current state.

That being said, React Router offers a **`<Prompt>`** component, which works by simply showing a native browser prompt dialog with a custom message. Here is an example:

```
<Prompt
  when={isFormDirty}
  message="You will lose your form data. Are you sure?"
/>
```

When your application enters a state where **`isFormDirty`** is **`true`**, this should prevent the user from navigating away.

EXERCISE 10.04: PROTECTED ROUTES

In this exercise, we are going to look at the usage of protected routes.

1. Create a new App:

```
npx create-react-app protected-routes
```

2. Go to the **`src/App.js`** file and delete **`logo.svg`**, **`App.css`**, and clear out the contents of **`App.js`**.
3. Import the **`App.css`** file we used in *Chapter 8*, *Introduction to Formik*.
4. Import the required modules:

```
yarn add react-router-dom formik
```

5. Import the necessary modules:

```
import React from 'react';
import { Formik, Form, Field } from 'formik';
import { Prompt, Link, Switch, Route, BrowserRouter as Router,
Redirect } from 'react-router-dom';
import './App.css';
```

6. Create validation functions for email and name:

```
function validateEmail(value) {
  let error;
  if (!value) {
    error = 'Email is Required';
  }
  return error;
}

function validateName(value) {
  let error;
  if (!value) {
    error = 'Name is Required';
  }
  return error;
}
```

7. Inside the **`App`** component, create link to the **`Feed`** component

```
function App() {
  return (
    <div className="App">
      <Router>
        <ul>
          <li>
            <Link to="/">Form</Link>
          </li>
          <li>
            <Link to="/Feed">Feed</Link>
          </li>
     <li>
            <Link to="/Dashboard">Dashboard</Link>
          </li>
        </ul>
```

8. Create the **`Switch`** component:

```
        <Switch>
          <Route path="/" exact children={<SignupForm/>} />
          <Route path="/feed" children={<div>Feed</div>} />
        </Switch>
      </Router>
    </div>
  );
}
```

9. Create the **`Formik`** component:

`App.js`

```
export const SignupForm = () => (
<div>
  <h1>Signup</h1>
  <Formik
     initialValues={{
       username: '',
       email: '',
     }}
    onSubmit={values => {
      // same shape as initial values
      console.log(values);
    }}
  >
```

`The complete code is available at:` https://packt.live/2AvyS8d

The output will be as follows:

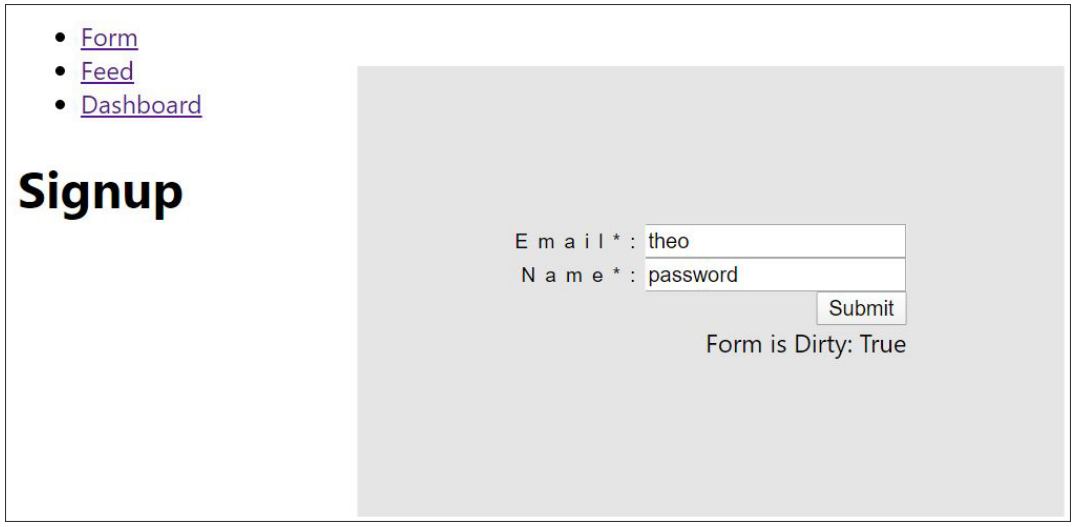

Figure 10.6: The localhost feed

Here, we have a typical Formik component with a few validations for email and name.

- First, we've used the dirty property, which we passed from the **`render`** function. This becomes **`true`** when we edit a field in the form. We also added a visual indication for this value.
- When the **`dirty`** value is **`true`**, the **`<Prompt>`** element is activated, but it will not show the prompt just yet. We need to navigate to another page for this to happen. So, if we click on the **`Feed`** link, we will be presented with some dialog to ensure we wish to proceed:

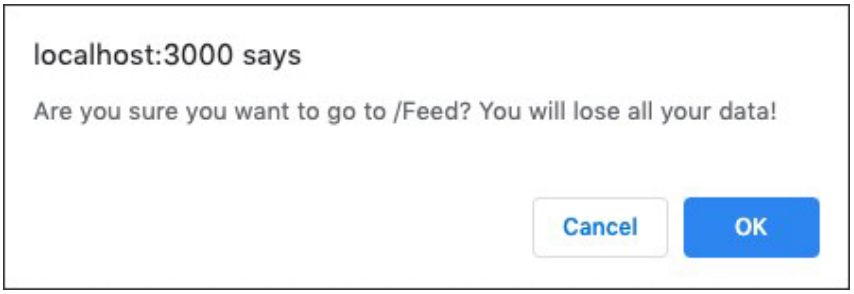

Figure 10.7: The localhost feed

There is also another way you can trigger the dialog message: by using the history package and any router type other than the **`StaticRouter`**. A **`StaticRouter`** is a router type that never changes location and is used mainly for testing.

We need to define a history object by passing a **`getUserConfirmation`** property that is a function with a message and a callback parameter. Then, inside this function, we can use a custom dialog or prompt, before calling the callback function. We also need to import a history object that lets us manage the history stack, navigate, and persist state between sessions. It is included with React Router upon installation.

Here is an example:

```
import {createBrowserHistory as createHistory} from 'history';
const history = createHistory({
    getUserConfirmation(message, callback) {
      const allowTransition = window.confirm(message);
      callback(allowTransition);
    }
  });

  <Router history={history}>
…
```

```
<Prompt
  message={location =>
    `Are you sure you want to go to ${location.pathname}? You will lose
all your
    data!`
    }
/>
```

Notice that we need to remove when condition from the **<Prompt>** component, otherwise the **getUserConfirmation** function will not trigger when we navigate elsewhere. In general, this way is more suitable for generic transition rules since the **getUserConfirmation** function will be called on every navigation event.

Now, let's learn how to prevent inbound transitions.

PREVENTING INBOUND TRANSITIONS

We can prevent inbound transitions by using protected routes. In other words, we can prevent the route from rendering if a condition is false. The easiest way we can do that is by using a Higher-Order Component (HOC) where we check the condition there. For clarity, an HOC is a function that takes a component and returns a new component, often by adding extra functionality.

For example, we can have the following **<IsAuthenticatedRoute>** component:

```
<IsAuthenticatedRoute exact path="/Dashboard" component={<div>Dashboard</
div>} />
```

Here, we have the following code:

```
import {Route, Redirect} from 'react-router-dom';

const authService = {
  isAuthenticated: function () {
    return false;
  }
};

const IsAuthenticatedRoute = ({ component: Component, ...rest }) => (
  <Route {...rest} render={(props) => (
    authService.isAuthenticated() === true
      ? <Component {...props} />
      : <Redirect to={{
        pathname: '/',
```

```
          state: { from: props.location }
        }} />
    )} />
);
```

Notice the usage of the condition inside the **<IsAuthenticatedRoute>** component. As long as **authService** returns false, the Component is not rendered. Instead, we render a **Redirect** component that navigates us back to the home screen.

The route also needs to be registered beforehand. What we mean by that is although this is a HOC, it's still a valid React Router Route, so we need to register it inside a **<Routes>** component.

When the route matches, the render props function will be called to check this with the authentication service. This way, any operation involving checking the authentication status is idempotent and it will perform the same computation each time it is called.

Now that we have a good grasp of some of the advanced React Router features, we can tackle a more complex activity.

ACTIVITY 10.01: CREATING AUTHENTICATION USING ROUTING TECHNIQUES

The aim of this activity is to get you to develop a list of routes that utilize what we have learned in this chapter. We will create a Login page that navigates to an **Enter your Security Token** page before authenticating and navigating to the main **Dashboard** page. We will have a specific set of requirements that we need to comply with. Our desired outcome will be a complete and smooth flow experience between pages, as shown in the following diagram:

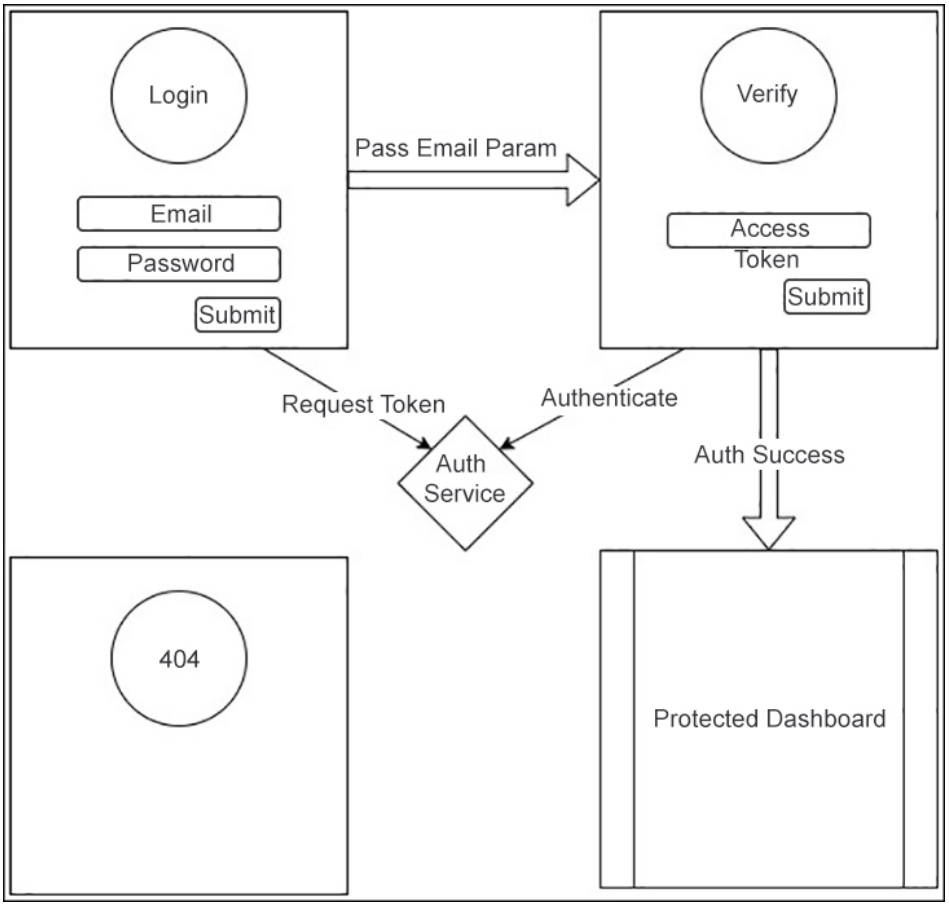

Figure 10.8: The flow diagram of the app

The following steps will help you complete the activity:

1. Create a **`404`** page that will handle unknown routes.
2. Create a Login page that is a React form with two fields, email and password. When we click on submit, the following events can occur:
3. Use the **`Auth Service`** to request an authentication token to our email box. For the purposes of this example, we can just hardcode some known list of codes.
4. Use the Auth Service to verify that the user password is correct using some hardcoded values for emails. If not, then do nothing.
5. If the password matches, then the page will immediately navigate to the Verify Token page, passing the email as a parameter.

6. Create a **`Verify Token`** page. This page needs to accept an email parameter and if that is missing, then it must redirect the user back to the login screen. This page also contains a form with one field, which is the **`Access`** token that was requested in the previous page. When we click on the Submit button, the following events happen:
7. We use the access token and email to the **`AuthService`** to verify that the user credentials are correct. Use the hardcoded tokens as an example. If the authentication succeeds, then make use of a temporary cookie or session storage or a variable to define that the user is **`isAuthenticated`**.
8. When the user is authenticated, then navigate to the main **`Dashboard`** page.
9. Create a **`Dashboard`** page that is only accessible when the user is authenticated. That is, when we deep link into that page and the user is not authenticated, then we redirect them back to the **`Login`** page.

The **`Dashboard`** page should be the same as the one we defined in the nested routes section of this article, with the addition of the **`Unknown Widget`** Route. That is, when we request an unknown widget, we need to show a message for that missing widget.

The final output should look like this:

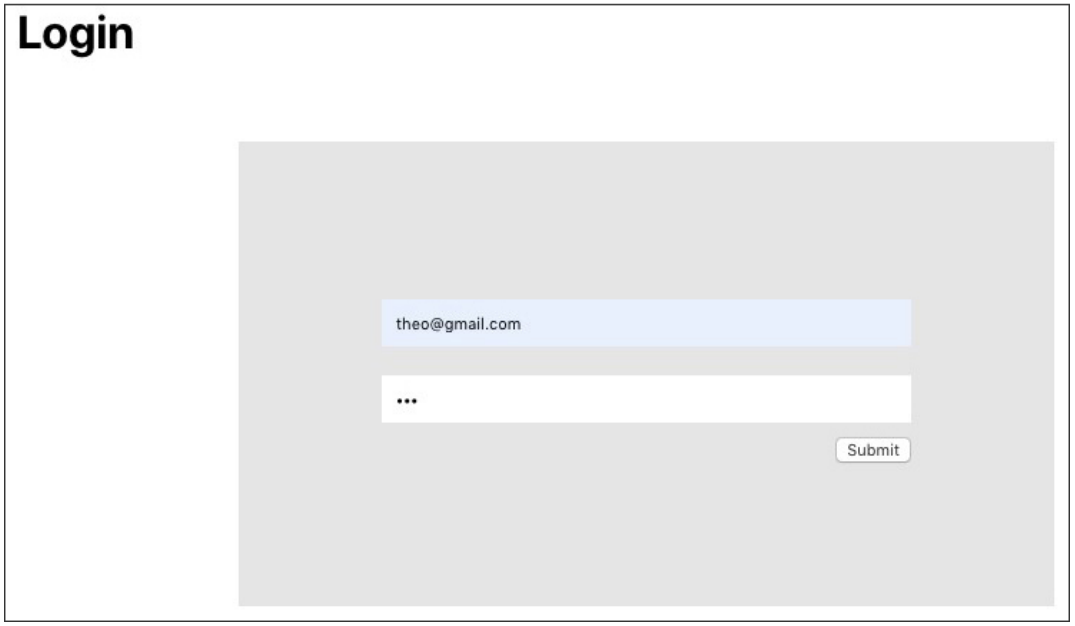

Figure 10.9: The final output of the app

> **NOTE**
>
> The solution of this activity can be found on page 679

SUMMARY

This concludes our exploration of this chapter. First, we learned quite a few things about how we can define **`404`** routes for unknown pages. We continued our journey by learning how to pass query and URL parameters and understood their inner differences.

We also spent some of our time understanding nested routes and created a simple dashboard page with inner routes for widgets. Then, we looked at how to prevent transitions from or out of the page by using protected routes and prompts.

We spent the majority of our time on this chapter's activity, where we had to design and implement a complex navigational flow from the **`Login to Verify`** pages and to the main **`Authorized Dashboard`** page. During that time, we had the chance to utilize most of the concepts that we learned about in this chapter and apply them in practice. In the next chapter, we will learn all about React Hooks, which are a new and modern way of reusing stateful logic between components.

11

HOOKS – REUSABILITY, READABILITY, AND A DIFFERENT MENTAL MODEL

OVERVIEW

This chapter will introduce you to React Hooks, enabling you to avoid wrapping your components in classes just to gain access to the state of a component. Using fewer abstractions will make the components lighter and will enhance the readability of the code. You will be able to optimize code while using Hooks in components efficiently. You will be able to create abstractions with Hooks that can be reused and so we do not need to rely on class-based components or component life cycle methods.

INTRODUCTION

In the previous chapters, we have seen how React handles state in class components. By the end of 2018, however, React developers had come up with a completely new API called Hooks, which changes how we manipulate the state of a component. This brings in massive syntax changes within the React framework. Even more importantly, this new method of state management can be used to manipulate state of the functional components.

Nothing in the life of a React component has a more significant effect than Hooks, which, of course, begs the question: what problems do Hooks solve that the former APIs could not? Previously, if we wanted to declare a state, we had to create a whole class and all the boilerplate code for that. Hooks, on the other hand, enable you to declare the state of a component with just one line. Hooks make React code more readable, maintainable, and reusable, while also making it a lot easier for newcomers to understand.

The Hooks library has been built with a newer version of ECMAScript; a script used to package JavaScript code for reuse. In this chapter, we will first take a look at the functionalities of Hooks. Later on in this chapter, we will look at the old style of writing code using class-based components and render props in React components and compare this method directly with Hooks and see how Hooks can improve the code you write.

Let's get started to see how you, as a developer, can benefit from embracing hooks.

HOOKS

In this section, we are going to introduce two of the popularly used React hooks: **`useState`** and **`useEffect`**. These are widely used and can solve most of our problems. The **`useState`** hook is used to initialize the state of a component and get access to a function that allows you to modify the state of that same component. The **`useEffect`** hook, on the other hand, is used when changes are made to the component, similar to the use case for **`componentDidMount`** or **`componentDidUpdate`** methods in class-based components.

> **NOTE**
>
> There are other types of hooks that come bundled with the React library. You can find a complete list of these at https://packt.live/3bCTh8d.

Let's dive right into these two particular hooks in more detail.

USESTATE

`useState` is the first type of hook that we are going to use. It gives us all the functionality that **`this.state`** and **`this.setState`** provide for class-based components. When we call **`useState`**, it will return an array where the first item in the array is the initial state of the component (which is what we pass in **`React.useState`**), and the second item is a function that acts identically as **`setState`** in a class-based component.

We can declare **`useState`** as following:

```
const [state, setState] = React.useState({ someFlag: false });
// or like this
const result = React.useState({ someFlag: false });
const state = result[0];
const setState = result[1];
```

The square brackets might seem intimidating, but they are exactly like *object destructuring* in JavaScript, except here we use arrays.

> **NOTE**
>
> Look at JavaScript de-structuring here: https://packt.live/2WxE2sS.

We have also discussed object de-structuring in *Chapter 7*, *Communication between Components*.

The first element of the arrays is the state itself, while the second element is a function that updates the state and, of course, triggers a re-render. We call this the **setter function**.

It can have two signatures; we can also say that the setter function can be overloaded. We will discuss the overloading of the setter function in more detail in *Chapter 12*, *State Management with Hooks*.

In this case, we will use the following code:

```
setState({ someNewState: true });
setState(prevState => ({ someNewState: !prevState.someNewState }));
```

The first example just takes a new state. For the second, you pass in a function with the previous state as the argument, and that function must return the new state. This is useful when our new state depends on the previous state. We will see the implementation shortly.

This behaves exactly like **`this.setState`** in class components. Now, let's work through a hands-on exercise to render an image using the **`useState`** hook.

EXERCISE 11.01: DISPLAYING AN IMAGE WITH THE TOGGLE BUTTON

In this exercise, we will display an image in a functional component. We will use an **`<img/>`** tag to render an image. Image tags require at least the source properties **`src`** and **`alt`** for accessibility. We will create a base component, and this will be provided to **`<Base/>`** and passed down as **`props`** later. To do so, let's go through the following steps:

1. Start by creating a new React project, which we will call **`imagehook`**, start the project, and take a look at the browser window that should have opened for you automatically:

```
$ npx create-react-app imagehook
$ cd imagehook
$ yarn start
```

2. Delete **`src/logo.svg`**. We won't be using this file, so we don't need to include it.

3. Replace the contents of **`src/App.css`** with the following:

```
img {
  width: 200px;
}
body {
  margin: 20px;
}
button {
  width: 200px;
  height: 50px;
```

```
  background: #4444ff;
  color: white;
  font-weight: bold;
  border: none;
  cursor: pointer;
}
```

You'll want to find an image to display in our component. For example, we could grab a nice picture of coffee beans from Unsplash, like the image we will use in this example: https://packt.live/3dMzzrl.

(Photo by Nadia Valko on Unsplash: https://unsplash.com/s/photos/coffee-beans?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText)

4. Replace the contents of the **`App`** component with the functional component below. This will set up our base UI, which we will refine through hooks later:

```
import React from "react";
import "./App.css";
const App = () => {
  const url = "https://images.unsplash.com/photo-1562051036-
e0eea191d42f";
  return (
  <div className="App">
    <img src={url} alt="Some coffee beans" />
    <br />
    <button>Toggle Image Display</button>
  </div>
  );
};
export default App;
```

This should give us our starting component UI, matching the following screenshot:

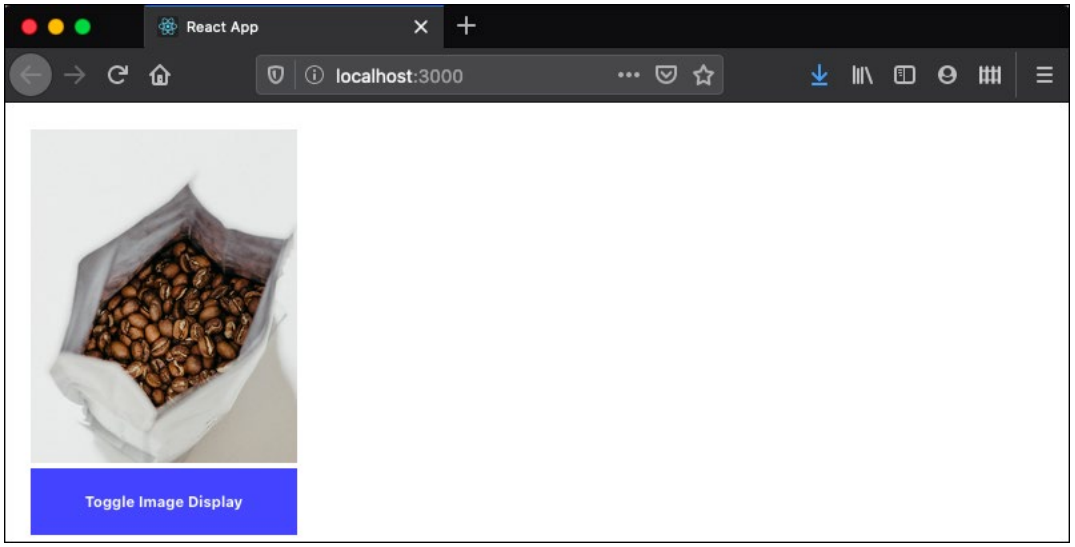

Figure 11.1: Toggle button

Let's expand this functional component a little more by making the image display toggleable and dependent on a **`displayImage`** Boolean value.

5. First, add the **`displayImage`** Boolean with a default value, false, and then change the JSX containing the **`<img>`** tag to be conditionally rendered when **`displayImage`** is **`true`**:

```
const App = () => {
  const url = "https://images.unsplash.com/photo-1562051036-
e0eea191d42f";
  const displayImage = false;
  return (
  <div className="App">
    {displayImage && <img src={url} alt="Some coffee beans" />}
    <br />
    <button>Toggle Image Display</button>
  </div>
  );
};
```

We are now ready to start adding React Hooks to this component. We will add a new line to the component that calls **`React.useState`**, pass in an initial value, **`false`**, and de-structure the result of that call into two new variables: **`displayImage`** and **`setState`**.

6. Use the following code

```
const [displayImage, setState] = React.useState(false);
```

Note that we are not using **`setState`** yet; we will be doing that soon. When your browser refreshes, you should no longer see the image on your component.

7. Next, we will need to implement a function that will toggle the **`displayImage`** flag via our **`setState`** function:

```
const toggleImage = () => setState(!displayImage);
```

8. Finally, hook it up by adding the **`toggleImage`** call to the button's **`onClick`** handler:

```
<button onClick={toggleImage}>Toggle Image Display</button>
```

Our final component code in **`src/App.js`** should now be the following:

`App.js`

```
1  import React from "react";
2  import "./App.css";
3
4  const App = () => {
5    const url = "https://images.unsplash.com/photo-1562051036-e0eea191d42f";
6    const [displayImage, setState] = React.useState(false);
7   const toggleImage = () => setState(!displayImage);
8  return (
```

The complete code can be found at: https://packt.live/2N9GEYi.

The output is as follows:

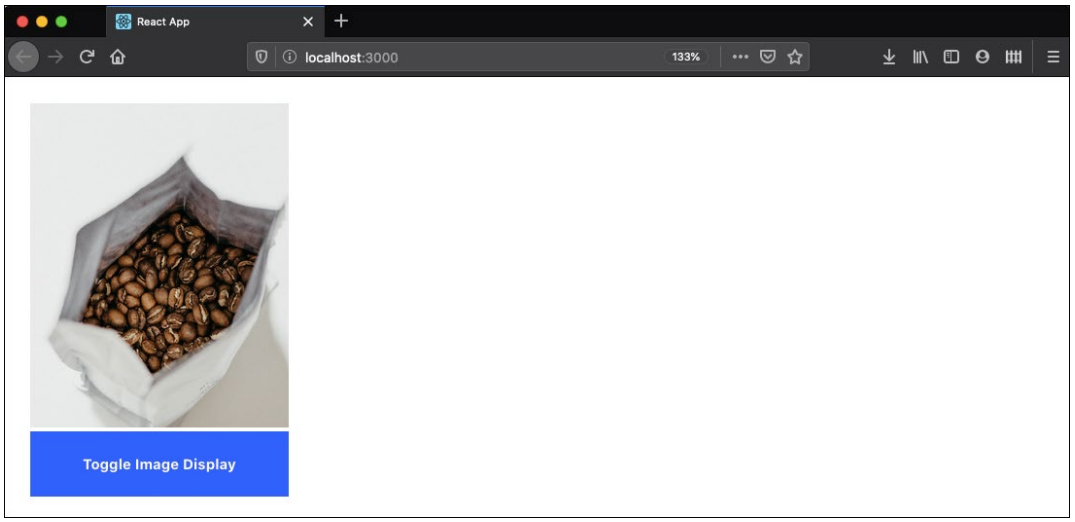

Figure 11.2: Final output of the Toggle button

Now, what would previously have been a complex abstraction has been reduced to only a function call. We don't need a complex constructor, multiple additional state modifying functions, or any other additional logic. It is close to our view layer; the order reads from top to bottom, which makes it super readable. Our JSX remained the same and no more wrappers got injected into the React tree. Similar to how we moved state modification away from class components, let's now see how we can move away from life cycle methods used in class components and use the **`useEffect`** hook instead.

USEEFFECT – FROM LIFE CYCLE METHODS TO EFFECT HOOKS

Hooks not only encompass new syntax in React but also require a different design pattern when it comes to developing the functional components in a React application. During the development cycle of a React application, components are continuously created, modified, and destroyed. Developers might want to use these events to call an API to modify the DOM elements, or maybe keep a log of information to display in the console.

As we've discussed previously, class components have life cycle methods to do this. If we wanted to do something like modifying the DOM or fetch some data by initiating a network request, we would use the life cycle methods, **`componentDidMount`**, **`componentDidUpdate`**, or **`componentWillUnmount`**. These life cycle events are tied to the insertion, updating, and removal of a given component. If you want to attach an event listener to a button in a form component, you could do that in **`componentDidMount`**. If you eventually need to remove it, you will use the **`componentWillUnmount`** method.

Other frameworks, such as Vue.js or Angular, have the concept of using life cycle methods as event callbacks. They provide callbacks to create a way to respond to events such as mouse clicks or mouse scrolls. However, with Hooks, there is a massive fundamental shift in the usual mental model. Instead of putting the code separately in each life cycle method, we group our logic and put it together into a functional hook called **Effects**.

Let's look at an example. We will create a class component, **`Comp`**, within which we will display our **`name`** prop when the component updates on mount. We will do this by implementing the life cycle methods, **`componentDidUpate`** and **`componentDidMount`**, as mentioned in *Chapter 4*, *React Lifecycle Methods*, as follows:

```
class Comp extends React.Component {
  componentDidMount(){
   console.log(this.props.name);
  }
  componentDidUpdate(prevProps){
  if(prevProps.name !== this.props.name) {
   console.log(this.props.name);
  }
  }
  render() {
  return <div>{this.props.name}</div>;
 }
}
```

If we want to capture similar functionality in a functional component as provided by these life cycle methods in a class component, we will need to use the **`useEffect`** hook. With the **`useEffect`** hook, we are going to take a different approach in terms of syntax. We will rewrite the **`Comp`** component, where we will create the logic first and will control when to run it like the following code:

```
const Comp = props => {
  React.useEffect(() => {
  console.log("name prop changed");
}, [props.name]);
return <div>{props.name}</div>;
}
```

As you can see from the preceding code, the **`useEffect`** hook takes two arguments:

- the first is the function that will be used to handle the effect.
- the other argument is an array of variables which will be used to determine whether we should call the function we passed in as the first argument.

In the preceding example, we have a React Effect we want to use, but we only want it to be executed when the **`name`** attribute in **`props`** gets updated, so we pass in **`[props.name]`** as our second argument. This effect (the first callback argument) will rerun only if the **`name`** prop changes. The logic is now encapsulated.

In a **`useEffect`** hook, the first argument is always a function where we put what we want to do (what we would have put into our life cycles for class-based components), and the second is an array of values. In that array, we set out our conditions when to perform effects. We don't need to rely on updates, mounts, and removals; we can create our own conditions. We can add multiple values to that array, which means that React will run that effect any time any of these values change. We can also leave this empty; in that case, the effect only runs on mount. We can also omit that array entirely in which case it will run every time the component re-renders.

The biggest difference between life cycle methods and **`useEffect`** is that with life cycle methods, you must think about which event is the right event to listen to and implement the correct logic for each. With **`useEffect`**, we think about the logic first and then control when to run it.

This has a few implications in terms of the readability of our code. For one, life cycle methods can be cluttered. If we have multiple things to do (such as logging the output and a network request), they will live inside the same function; they cannot be extracted.

In the following code, we are creating a class component, **`Comp`**, where we will perform multiple different actions when the component updates; the **`name`** prop will be displayed in the console and a network request will be initiated:

```
class Comp extends React.Component {
  componentDidMount(){
   console.log(this.props.name)
  }
  componentDidUpdate(prevProps){
    // this is one piece of logic
    if(prevProps.name !== this.props.name) {
      console.log(this.props.name);
    }
    // this is another
    fetch('https://jsonplaceholder.typicode.com/todos/1');
  }
}
```

In the preceding example, the **`fetch`** method used to initiate a network request has no connection with the **`console.log`** function, yet it still lives inside the same **`componentDidUpdate`** life cycle method in a class-based component.

`useEffect` focuses on the logic and allows you to couple logic with specific properties or attributes. This allows us to create one effect for the console and a different one for a network request, and we can control when they are called. This provides a cleaner and more elegant approach to writing different effects for different use cases.

Using **`useEffect`** hooks, we would rewrite the **`Comp`** component to look more like this:

```
const Comp = props => {
 React.useEffect(() => {
  console.log("name prop changed");
}, [props.name]);
React.useEffect(() => {
  fetch('https://jsonplaceholder.typicode.com/todos/1');
}, [props.name]);
return <div>{props.name}</div>;
}
```

Here, there are two separate hooks for the two different actions where the logic remains encapsulated. Our code is cleaner and the logic for each property is clear and easy to follow. Using this knowledge, let's build a component using **`useEffect`**.

EXERCISE 11.02: CREATING A LOGIN STATE USING USEEFFECT

In this exercise, we will utilize **`useEffect`** to create a component where we simulate logging in (and remaining logged in even if we refresh the page) through a combination of hooks and **`localStorage`**.

> **NOTE**
>
> **`localStorage`** is a part of the HTML5 specifications. With the help of **`localStorage`**, we can write values to our browser that will be persistent across re-renders. This constitutes unique persistent storage for our domain. If you would like to know more about **`localStorage`**, you can visit MDN for more information: https://packt.live/2Z1eOEF).

For this exercise, it is recommended to have Chrome developer panels open on the **`Application`** page. For that, you only need to right-click on your web page, click on **`inspect`**, and, on the tabs starting with elements, find **`Application -> Storage -> Local Storage`**:

1. Begin by creating a new React project, which we will call **`login-storage`**, start the project, and take a look at the browser window that should have opened for you automatically:

```
$ npx create-react-app login-storage
$ cd login-storage
$ yarn start
```

2. Delete **`src/logo.svg`**.
3. Replace the contents of **`src/App.css`** with the following:

```
body {
  margin: 20px;
}

button {
  width: 200px;
  height: 50px;
```

```
    background: #4444ff;
    color: white;
    font-weight: bold;
    border: none;
    cursor: pointer;
}
```

4. We will begin by creating the base UI without any of the logic attached to it yet. This will help us to understand how the UI will function later. Right now, the UI will just have a block of text telling us whether we are logged in or not and a button to **`log in/log out`**:

```
import React from "react";
import "./App.css";
const App = () => {
  return (
    <div className="App">
    <p>Logged Out</p>
    <button>Log In</button>
    </div>
  );
};
export default App;
```

Our early UI should resemble the following screenshot:

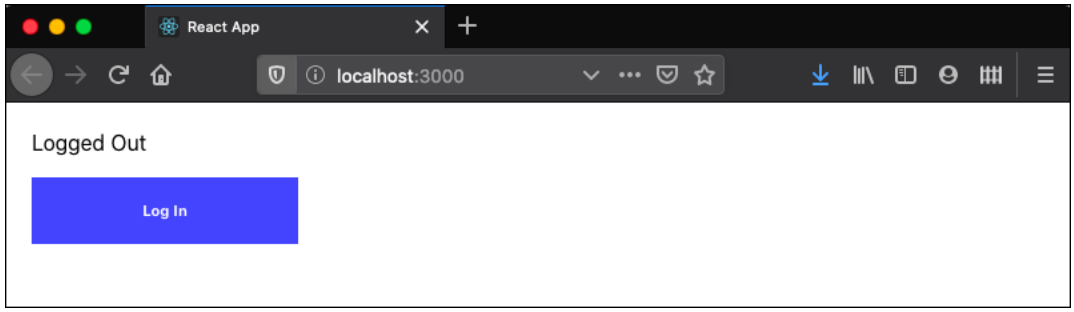

Figure 11.3: Component logged out

Now, we can incorporate the logic based on React Hooks.

5. Begin by using **`React.useState`** again to declare our initial state and store the results in **`loggedIn`** and **`setLoggedIn`**, respectively:

```
let [loggedIn, setLoggedIn] = React.useState(false);
```

6. Add some conditional logic to display **`Welcome Back`** if the user is logged in, or **`Logged Out`** if the user is logged out:

```
<p>{loggedIn ? "Welcome Back!" : "Logged Out"}</p>
```

7. Add a login condition to the button as well. There should be an **`onClick`** event handler that calls the **`setLoggedIn`** function and toggles the login state, while the text of the button should also be conditional based on whether the user is logged in:

```
<button onClick={() => setLoggedIn(!loggedIn)}>
  {loggedIn ? "Log Out" : "Log In"}
</button>
```

8. Now, we need to save the logged-in state to **`localStorage`**. We will use the **`localStorage.setItem(key, value`**) function to set a string key in the browser's local storage. This will allow the value of **`loggedIn`** to persist even if the browser refreshes. We want to store the **`loggedIn`** variable to **`localStorage`**. That would look something like this:

```
localStorage.setItem("loggedIn", loggedIn);
```

9. Use the preceding code to write our **`useEffect`** hook, which we only want to trigger when the **`loggedIn`** value is changed:

```
React.useEffect(() => {
  localStorage.setItem("loggedIn", loggedIn);
}, [loggedIn]);
```

The full component we have built so far should be the following:

```
import React from "react";
import "./App.css";
const App = () => {
  let [loggedIn, setLoggedIn] = React.useState(false);
  React.useEffect(() => {
    localStorage.setItem("loggedIn", loggedIn);
  }, [loggedIn]);
```

```
    return (
      <div className="App">
      <p>{loggedIn ? "Welcome Back!" : "Logged Out"}</p>
      <button onClick={() => setLoggedIn(!loggedIn)}>
        {loggedIn ? "Log Out" : "Log In"}
      </button>
      </div>
    );
  };
  export default App;
```

10. Next, use your browser's development tools to see what values are getting stored in **`localStorage`** and the output is as follows:

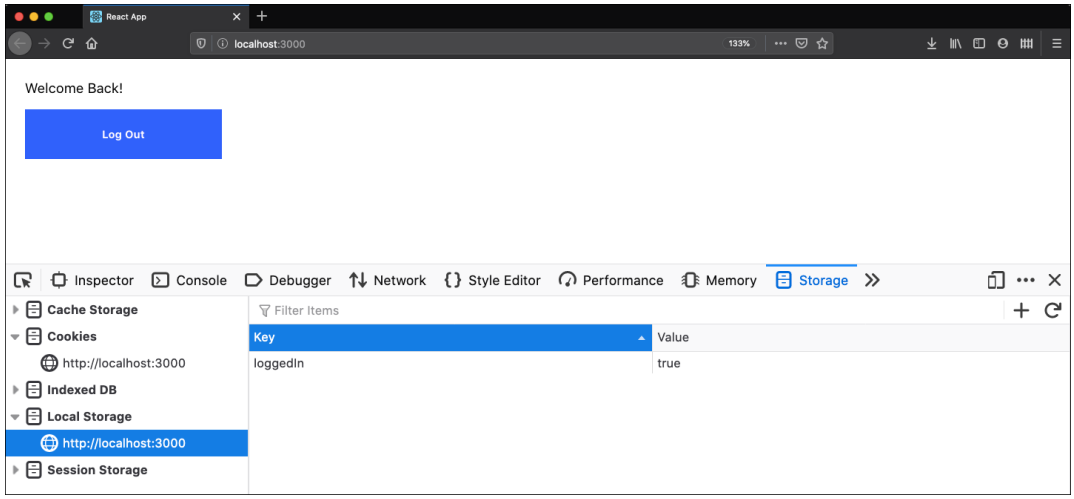

Figure 11.4: Opening browser development tools to see localStorage

When the app starts or we refresh the page, our program is not picking up the value from **`localStorage`**. Let's apply the same logic we used before.

11. Apply the same logic we used before and we will grab the value from **`localStorage`** with the help of the **`getItem`** function. It takes one argument: the identifier (or key) of the value. We could get the value out using the following code:

```
const loggedInFromLocalStorage = localStorage.getItem("loggedIn");

if (JSON.parse(loggedInFromLocalStorage) === true) {
  setLoggedIn(true);
}
```

    This returns a **`string`**, so we need to use **`JSON.parse`** to parse it to a **`boolean`**. If that is **`true`**, we set the state to **`true`** as if we clicked on the Log-n button. Let's place this in our code in a **`useEffect`** hook. We want to run this code on mount, so it runs once, and automatically return the value in the console if it is **`true`**.

12. Specify an empty array as the values to watch on the **`useEffect`** call to have our hook only affect when our component is mounted:

```
React.useEffect(() => {
    const loggedInFromLocalStorage = localStorage.
getItem("loggedIn");

    if (JSON.parse(loggedInFromLocalStorage) === true) {
      setLoggedIn(true);
    }
}, []);
```

The complete code for our component is as follows:

App.js

```
1  import React from "react";
2  import "./App.css";
3
4  const App = () => {
5  let [loggedIn, setLoggedIn] = React.useState(false);

6  React.useEffect(() => {
7    const loggedInFromLocalStorage = localStorage.getItem("loggedIn");
8
9    if (JSON.parse(loggedInFromLocalStorage) === true) {
10      setLoggedIn(true);
11    }
12  }, []);
```

The complete code can be found here: https://packt.live/2y5Crky.

The output will be as follows:

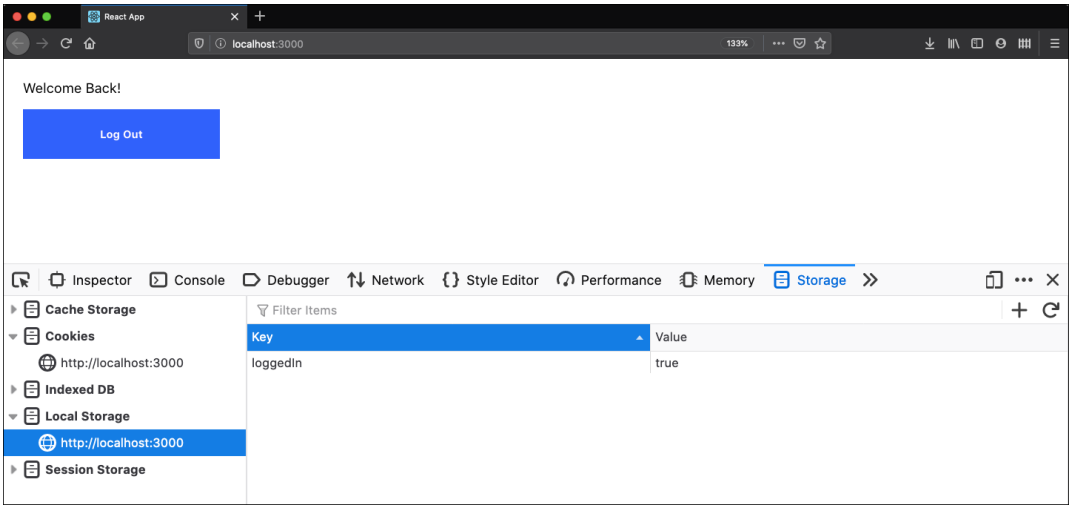

Figure 11.5: Welcome message

Even if we refresh our page, we are still logged in. Try it for yourself to verify whether it is working in the way you expect.

COMPARING USEEFFECT HOOKS WITH LIFE CYCLE METHODS

After seeing how we can be more precise in our code using effects compared to life cycles, let's have a brief overview, through a diagram, to see how the life cycle methods in the class components work, as shown in the following:

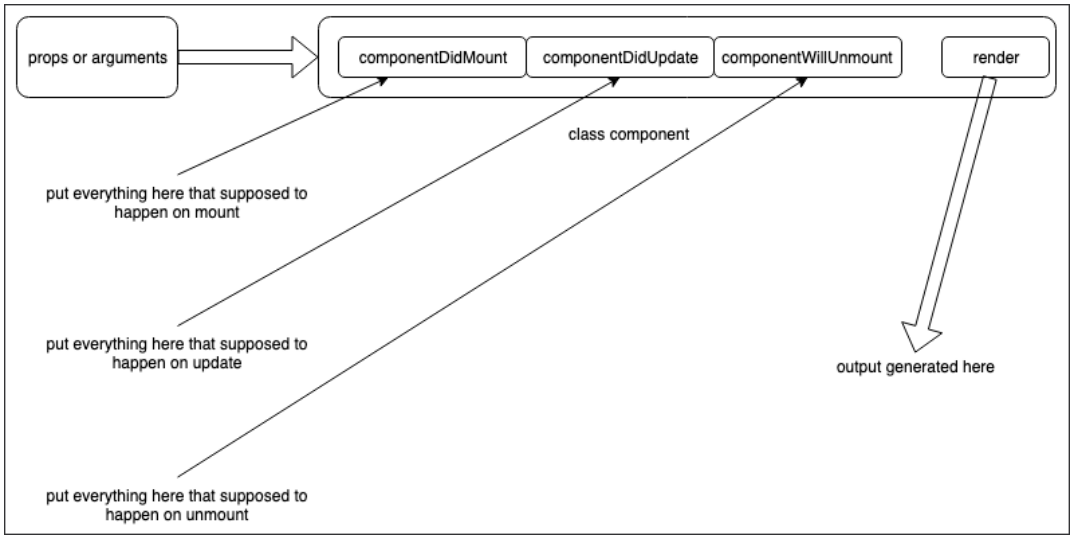

Figure 11.6: Life cycle methods

In classes, you have life cycle methods that tap into some state of that component. The problem with that model is that you may actually need to break each event into multiple different unrelated functions. For example, say you want to attach an event listener and, to avoid memory leaks, you want to remove the listener when it is no longer needed. What you must do in a class-based component is to attach it in **`componentDidMount`** and remove it in **`componentWillUnmount`**. This means that, now, both of those life cycle methods must track and have access to that listener. Instead of our code being grouped by context, it is grouped by life cycle methods.

Let's now see how Effect Hooks work by means of a diagram:

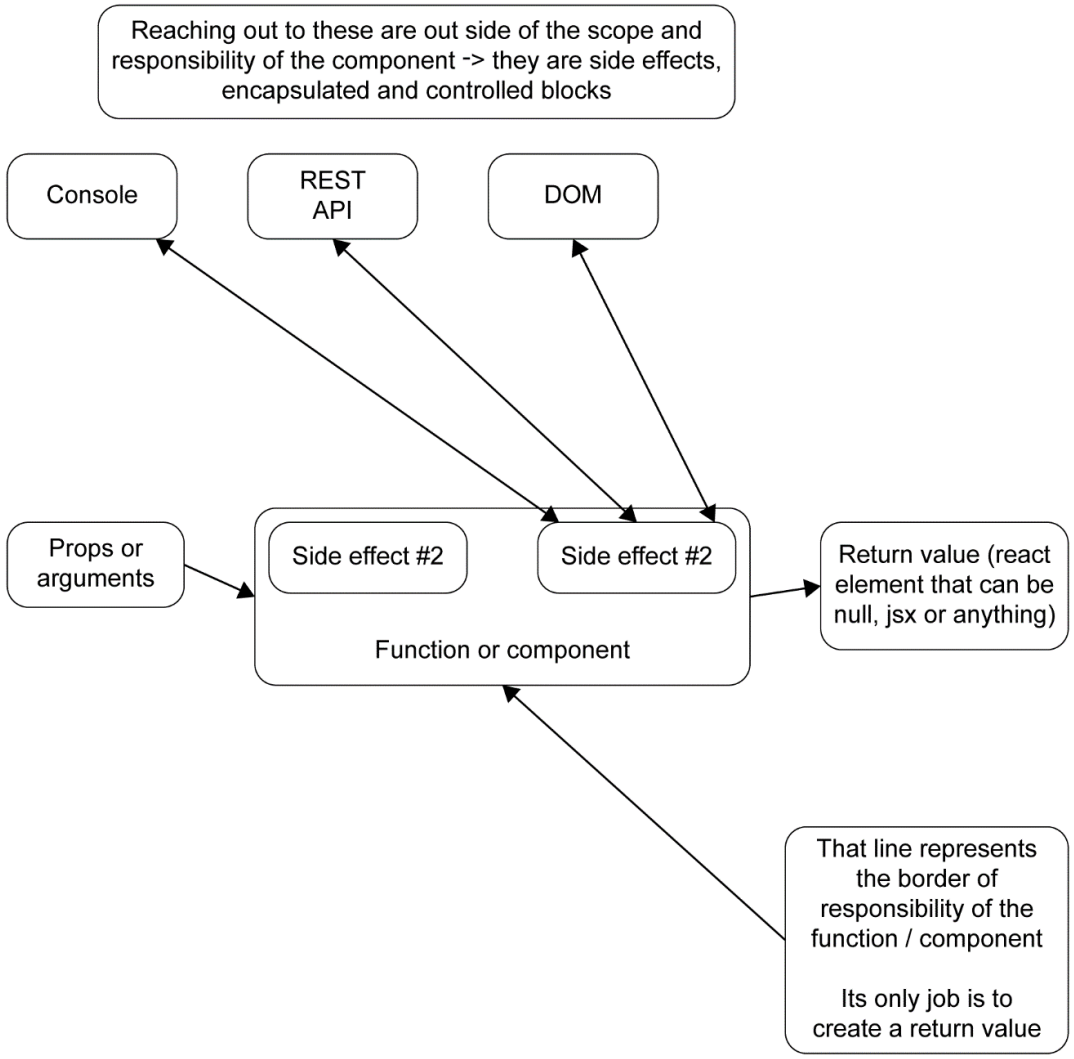

Figure 11.7: UseEffect hook

It is different with hooks. If we have an event listener, we put it to one **`useEffect`**. In that **`useEffect`** hook, we manage everything related to that specific context, and no unrelated code will live in that effect. If we need to do something unrelated, we instead create a separate effect for that. In the preceding diagram, we have two logical units and they are put into two separate effect hooks. With life cycle methods, they might have shared the same life cycle or life cycle methods depending on when they were required.

COMPARING HOOKS TO RENDER PROPS

Now that we know how to apply these hooks, let's take a quick look at why such a paradigm shift was required. The idea behind introducing hooks was mainly to improve developer experience. At React Conf 2018, the React team tried to address three main points:

- Make smaller components where logic is easier to understand
- Make components more reusable
- Limit the use of class-based components as they are harder to optimize

After practicing some exercises on how to start using the **`useState`** and **`useEffect`** hooks with functional components, it might be a good idea to see how hooks actually simplify the design pattern.

To do so, let's take the final component of *Exercise 11.01*, *Displaying an Image with the Toggle Button*, which contains code where we built an app that loads and shows an image by pressing a button.

We had a component called **`<App/>`** that rendered an **`<img/>`** tag:

```
const App = () => {
  const url = "https://images.unsplash.com/photo-1562051036-
e0eea191d42f";
  const [displayImage, setState] = React.useState(false);
  const toggleImage = () => setState(!displayImage);
  return (
  <div className="App">
  {displayImage && <img src={url} alt="Some coffee beans" />}
  <br />
  <button onClick={toggleImage}>Toggle Image Display</button>
  </div>
  );
};
```

We had a **`toggleImage`** function that hides the logic toggling the **`displayImage`** variable between **`true`** and **`false`** depending on whether you have a value inside it. We could further extract this out to its own function so that it is not bound to the **`<App />`** component, and then we can throw it in any other functional component like so:

```
const useToggle = initialValue => {
  const [value, setValue] = React.useState(initialValue);
  const toggle = () => setValue(prevValue => !prevValue);
  return [value, toggle];
};
```

The preceding code demonstrates what **`useToggle`** looks like. It's just a function that can be called in any functional React component. For example, using it back in our original App component, we would get code similar to the following:

```
const App = () => {
  const url = "https://images.unsplash.com/photo-1562051036-
e0eea191d42f";
  const [displayImage, toggleState] = useToggle(false);
  const toggleImage = () => toggleState(!displayImage);
  return (
  <div className="App">
  {displayImage && <img src={url} alt="Some coffee beans" />}
  <br />
  <button onClick={toggleImage}>Toggle Image Display</button>
  </div>
  );
};
```

It's not dramatically different, and it's even easier to reuse the simple toggle function going forward. Now, it's time to put all of this together and work through this activity, building a reusable Counter component.

ACTIVITY 11.01: CREATING A REUSABLE COUNTER

We will start by building our **`Counter`** component using render props and move the logic into Hooks over the course of this activity.

We will design a reusable click counter in this activity. We will remove the state (**`setState`**) first, convert it to a functional component, abstract the logic away, and control the component entirely through props. We should be able to set an initial number. Also, every time the value changes, it should be logged out. That function should be separate from the toggle logic.

The following steps will help you to execute this activity:

1. First, set up a standard component that you'll use to display the **`Counter`** component. This is usually our **`App`** component.
2. Next, you will want to set up the **`Counter`** component, which will just display the value for the counter.
3. Then, set up the state for the counter variable and the helper functions to increment, decrement, and reset the counter.
4. Set up a UI to increase, decrease, and reset the counter.
5. Make sure you keep your UI (presentation or view) layer separate from the counter through the use of **`useState`**.
6. Create a callback using the setter function on **`useState`**.
7. Create a logging hook using **`useEffect`** hooks.
8. Finally, make sure the counter hook is reusable by making it a custom hook called **`useCounter`**.

The final output should look like:

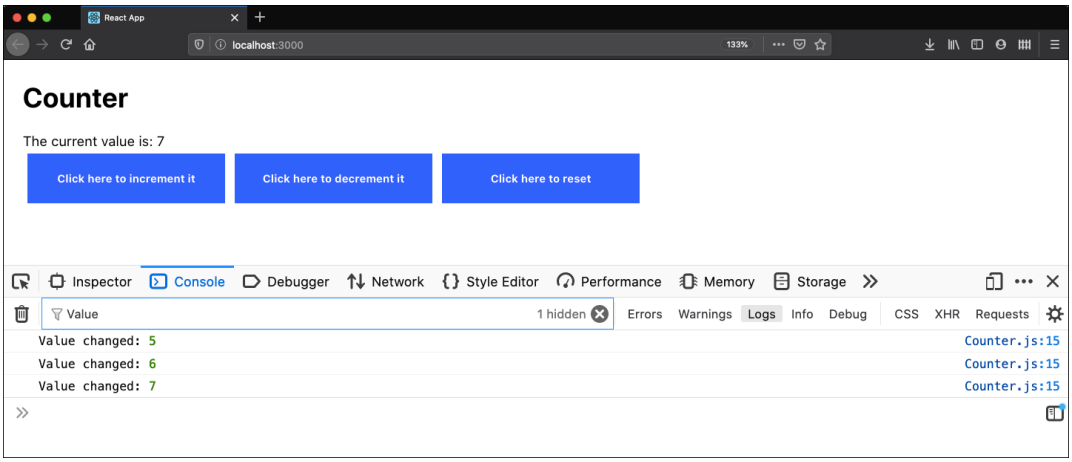

Figure 11. 8: Reusable Counter App

> **NOTE**
>
> The solution of this activity can be found on page 689

SUMMARY

This chapter introduced us to why hooks are better in a lot of ways than prior methods. The activity demonstrated how much easier it is to create an app using Hooks, and how much simpler it is to read and reason about what it actually does.

Over the course of this chapter, we have seen how React Hooks can allow us to write cleaner, more manageable code as our applications expand in complexity. The introduction of hooks allows us to think outside of classes and, even more importantly, allows us to throw away complex mental models of state management passing through multiple separate parent and child components.

In the next chapters, we are going to concentrate on effects, how to further use hooks, what basic hooks React provides, and how we can put them together.

12

STATE MANAGEMENT WITH HOOKS

OVERVIEW

This chapter will focus on a detailed approach to using Hooks. You will be able to use the useState and useReducer functions to manage state and decide when to use what. You can create and manage complex state transitions using these hooks. To avoid the most common bugs, you will be able to use the **`useEffect`** hook and the various techniques of performing cleanups in order to optimize your code. You can create custom hooks based on state and effect. By the end of this chapter, you will have a solid understanding of implementing hooks.

INTRODUCTION

We have seen in the previous chapter how Hooks, as compared to render props, can simplify a React app. The methodical approach of Hooks helps in creating easy-to-maintain and easily composable components, while keeping the React tree intact.

In this chapter, we are going to learn about state management in the functional components. The two most commonly used hooks that help us in making these components stateful are **`useState`** and **`useReducer`**. Over the course of this chapter, we will explore the trade-offs between the two and when to use which.

Since functional components cannot have life cycle methods to manage state, we had a discussion in the previous chapter about how the **`useEffect`** hook can perform identical functions, but without the risk of needing to group behavior together due to where it happens in the life cycle of the component. This has many advantages. For example, we can create separate and dedicated **`useEffect`** hooks to write to databases, to manipulate the Document Object Model(DOM), to write to the console, and more; we don't have to tie them to specific points during the component's life cycle.

However, while using the **`useEffect`** hook, we need to be careful about how we structure our code, or we might encounter some serious bugs. By far the most common causes of bugs with **`useEffect`** are due to the careless use of closures. Thankfully, React Hooks provides a solution to the problem of dangerous closures, specifically, another hook: **`useCallback`**.

Let's dive deeper and discuss what to look for when writing the **`useState`** hook to prevent the most common bugs from happening.

USESTATE HOOK: A CLOSER LOOK

As we have seen in the previous chapter, **`useState`** is the way to provide state to our functional components, similar to **`this.state`** and **`setState()`** in a class-based component.

As we already know, the state is always an object in classes. However, in Hooks, the state can be of any datatype. We used the **`useState`** hook in the previous chapter with primitive values such as number and Boolean. This introduces several complications later on when we rely on datatypes that are modified in a different way to primitive types, such as arrays or objects.

The first complication arises when we use setter functions with arrays, so let's explore that first.

SETTER FUNCTIONS ON ARRAYS

We have already become comfortable with using setter functions when using the **`useState`** hook in the previous chapter. As a refresher, let's return to the setter function for a while; the second argument of the **`useState`** array. We can pass a value directly to the setter function to set the state to that value (as we have seen in the previous chapter):

```
const [value, setterFunction] = React.useState(0);
// setter function takes a value directly
setterFunction(1);
```

Alternatively, it can also accept a callback, which can be helpful if we need to access the previous state of a component. In other words, a setter function takes a callback where the argument is the previous state value and then returns the new state, as in the following example:

```
// setter function takes a callback where the argument is the previous //
value and the return statement will be the new state
setterFunction(previousValue => previousValue + 1);
```

We can use the setter function to manipulate the state, just as we did with primitive values in *Chapter 11*, *Hooks – Reusability, Readability, and a Different Mental Model*. If we have a number in our state, we might want to increment it. With an array, however, this can be a bit more complicated.

After calling the push method, we returned the same instance of the stock from that function. If **`Object.is`** returns **`true`**, then React will not re-render. You can see in your console when you press the **`add item`** button that Object.is returns the value **`true`**.

If we modify a variable structurally (that is, we mutate it), we must return a new object. The good news is that there are a lot of ways to do this in JavaScript. In ES6, one of the ways we can do this is through a new feature called spreading.

Spreading is telling JavaScript to grab the contents of an array (the elements of the array) or an object (the fields of the object), add something new to it, and then return a new array or object:

```
const arr = [1,2,3];
const obj = {name: "Joe", age: 30};
const newArr = [...arr]; // [1,2,3]
const newObj = {...obj}; // {name: "Joe", age: 30};
```

Structurally, **`arr`** and **`newArr`** are the same (they have the same content), just like **`obj`** and **`newObj`**. But when we check whether they are referentially equal (that is, they have the same memory location), this is what happens:

```
Object.is(arr, newArr); // false
Object.is(obj, newObj); // false
```

In a nutshell, we need to return a new array that is a copy of the previous array with new data added to it; otherwise, the state setter function will not work the way we expect it to. Let's explore this with a real-world example in our first exercise for this chapter.

EXERCISE 12.01: ARRAY MANIPULATION USING HOOKS

In this exercise, we will see how to correctly manipulate arrays using Hooks. We will continue with the **`setStock`** function that we began with in the previous example. We will still add a new item in an array, and we will see how to overcome the issue where nothing happened when we clicked the button:

1. Start off by creating a new React app:

```
$ npx create-react-app hook-arrays
```

2. Go to the **`src/App.js`** file and delete **`logo.svg`**.
3. Replace the contents of **`src/App.css`** with the following:

```
.App {
  margin: 20px;
}

button {
  width: 200px;
  height: 50px;
  background: #4444ff;
  color: white;
  font-weight: bold;
  border: none;
  cursor: pointer;
}
```

4. Replace the contents of **`src/App.js`** to get an empty component to work with:

```
import React from "react";
import "./App.css";

const App = () => {
  return (
  <div className="App">
    <p>Stock</p>
    <button>Add New Item</button>
  </div>
  );
};
export default App;
```

5. Next, we want to set up our state with **`React.useState`**. We will set up a **`stock`** state and a **`setStock setterFunction`**. The initial value for this will be an empty array. This will be inside your **`App`** component:

```
const [stock, setStock] = React.useState([]);
```

6. Display a **`<p />`** element for each string in the stock:

```
{stock.map((s, i) => (
  <p key={i}>{s}</p>
))}
```

7. Create an **`addStock`** function inside our **`App`** component that we will call **`setStock`** and add to the previous value. Remember that setter functions will take the previous state as an optional argument in their definitions:

```
  const addStock = () => {
  setStock(prevStock => {
    console.log("Adding new item...");
    prevStock.push("New Item");
    return prevStock;
  });
  };
```

8. Hook the **`addStock`** function to our button's **`onClick`** event handler:

```
    return (
    <div className="App">
    {stock.map((s, i) => (
      <p key={i}>{s}</p>
    ))}
      <button onClick={addStock}>Add New Item</button>
    </div>
    );
```

When we click on the button, we should see the **`Adding New Item..`** statement show up in Console, but nothing gets added to our **`stock`**. Don't worry; this is intentional. We are going to fix this up in the next step. Our UI (with the console) at this point should look something like this:

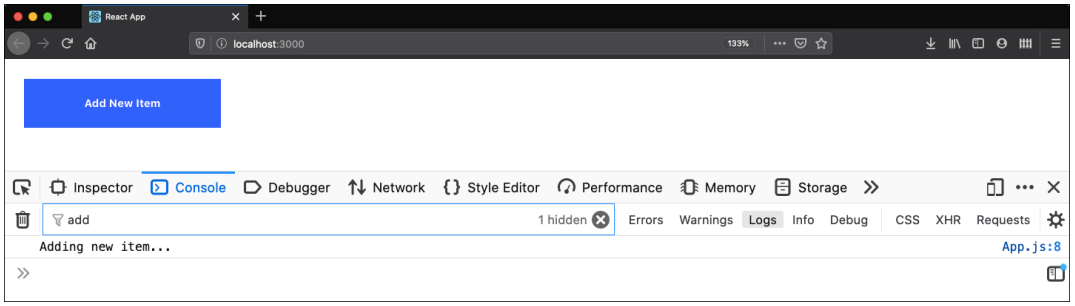

Figure 12.1: App console

This is an erroneous behavior we mentioned previously regarding modifying arrays or objects.

Knowing this catch with the hook and the **`Object.is`** method (this is used in React to check whether two objects are equal), let's modify the setter function to return a new array instead of mutating the old one. In our case, **`prevStock`** is an array, so we need to return a new array by declaring it with the array literal (**`[]`**) and then we add the contents of the **`prevStock`** array to it.

9. Change our **`addStock`** function to use the array spread operator to copy the array, add a new item to the end of it, and then return a new array:

```
const addStock = () => {
setStock(prevStock => {
console.log("Adding new item...");
return [...prevStock, "New Item"];
});
};
```

As you can see, we return a completely new array with previous stock in it. While there are other methods to create a new array, the important point is that we must always return a new item when we mutate something. This will ensure that React re-renders the component.

The output is as follows:

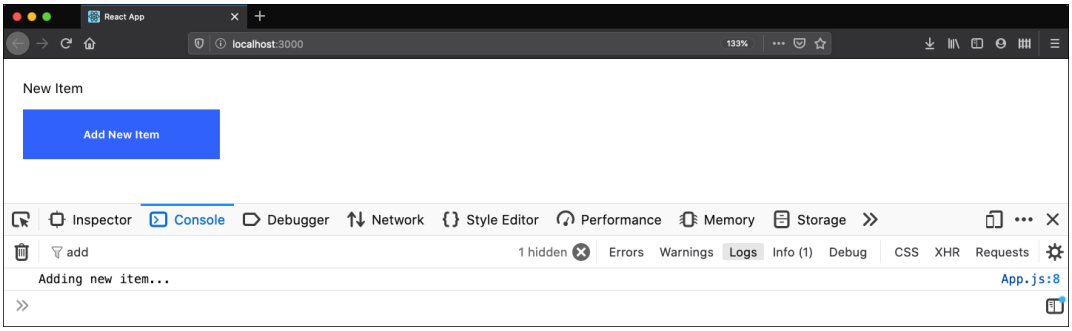

Figure 12.2: App console

As we can see, in the Console, we still have the message, and we have the new item in our UI display as well.

EQUALITY AND IMMUTABILITY FOR OBJECTS IN REACT

As mentioned previously, **`useState`** only triggers a re-render if the previous and current states are not equal. Under the hood, React uses **`Object.is`** to determine whether two things are equal.

Let's see how it behaves with objects. In JavaScript objects, arrays, and functions are essentially objects, so they behave the same way if we put them into **`Object.is`**. For example, let's take a look at the following code:

```
Object.is({}, {}); // false
const item = {
 price: 4,
};
Object.is(item, item); // true
Object.is(item, {price: 4}); // false
```

What we can see here is that **`Object.is`** does not care whether object has the same content (structurally equal); only if that is the same object in memory (reference equality). That means that objects are subject to the same setter function that arrays are subject to, so keep that in mind if you design a state that relies on either objects or arrays.

LIMITATIONS OF USESTATE

`useState` is a great way to define a state, but it does have its limitations. The setter functions can grow too big if the logic gets complicated, plus you have to make sure that you return new items in every setter function using the spreading syntax. Also, if one state depends on another, you can end up writing tedious logic to make sure everything is correct in all of your dependencies.

When dealing with complex interactions, we need to define multiple pieces of states and managing logic can get out of control. We can prevent this state logic from getting too out of control through the use of **`useReducer`**.

USING THE USEREDUCER HOOK

Before we can start utilizing the second hook that React offers for state management, let's go through the fundamental difference as to how **`useState`** and **`useReducer`** approach state transitions (rather, how we can set a new state). With **`useState`**, we concentrate on how to transition each individual piece of state. With **`useReducer`**, we concentrate instead on the entirety of the state and setting logical rules for state manipulation.

Now, take a look at this pseudocode (this is provided as an example; don't try to run this):

```
const Thermostat = () => {
  const [temperature, setTemperature] = React.useState(20);
  return (
  <>
  <div>current temperature is {temperature}</div>
  <button onClick={() => do(INCREMENT)}>increase temperature</button>
  <button onClick={() => do(DECREMENT)}>decrease temperature</button>
  </>
  )
}
```

Here, the design is separated out into each individual piece of state. We just care about what state we need and later rely on something happening to that state, which is a very unguided approach. For example, in the preceding code snippet, we are using a **`temperature`** state, but what if we put another value such as a string into that state? We have no good way to prevent that from happening and effects on the state are scattered all over the place.

Let's look at the approach that **`useReducer`** takes:

```
const Thermostat = () => {
  const [state, dispatch] = React.useReducer(
  (prevState, action) => {
    if(action.type === "inc") {
    return {temperature: prevState.temperature + 1};
  }
  if(action.type === "dec") {
    return {temperature: prevState.temperature - 1};
  }
  return prevState;
  },
  { temperature: 20 }
  )
  return (
  <>
  <div>current temperature is {state.temperature}</div>
```

```
  <button onClick={() => dispatch({type: "inc"})}>increase temperature</
button>
  <button onClick={() => dispatch({type: "dec"})}>decrease temperature</
button>
  </>
  )
}
```

Let's examine what is happening here:

- First, we have the same array de-structuring as we had with **`useState`**. Again, we call the first element state and the second dispatch (we could have called it *do* just as easily).
- The first element is the **`state`** element. That serves the same purpose as the first argument did with **`useState`**.
- The second is much more exciting: dispatch is a function that can take any kind of value as an argument. Again, because of conventions, we usually pass an object to it with at least one field: type.
- As you can see, that is what happens in the **`onClick`** handlers. **`dispatch ({type: "inc"})`** tells the reducer what to do but doesn't concern itself with how to modify the state otherwise.
- Now, you might say, fair enough, but at some point, we need to specify the *how* Let's explore how to answer that in the reducer function.

REDUCER FUNCTION IN REACT

The first argument to **`useReducer`** is a reducer function. This is called with the previous state and action variables. Let's look at the following code:

```
const [state, dispatch] = React.useReducer(
  (prevState, action) => {
  if(action.type === "inc") {
  return {temperature: prevState.temperature + 1};
  }
```

So, in the case of our preceding example, action is **`{type: "inc"}`** or **`{type: "dec"}`**.

The second argument to **`useReducer`** is the initial state. In the preceding example, it's an object. Now, recall that the setter function can be called with a callback where we have access to the previous state and must return a new state. The reducer function is very similar to that. It receives the previous state and must return a new state. The only difference is that it also receives the action. To define the action simply, action is the value you call your **`dispatch()`** function with. Let's take a look at the following diagram to see the difference in approach for **`useState`** and **`useReducer`**:

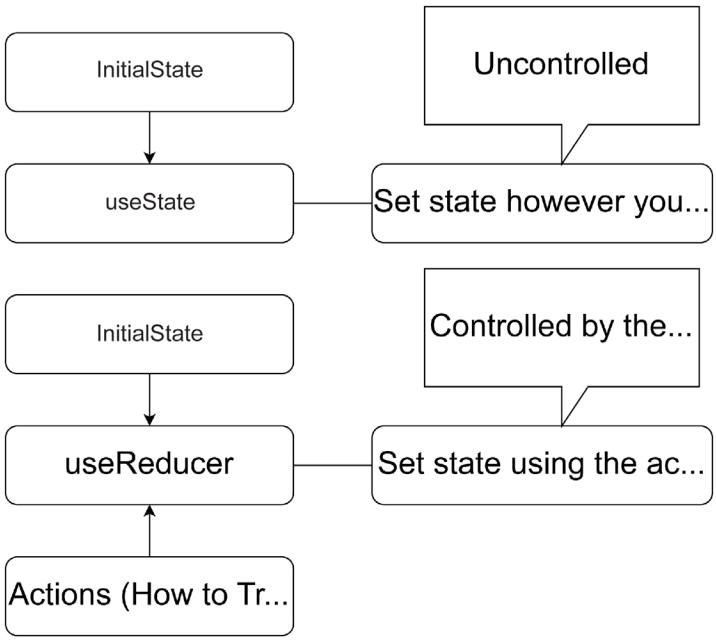

Figure 12.3: Difference in approach for useReducer and useState

Now, the reducer has all the information it needs: the previous state and what needs to be done. With **`if`** statements, we can check the action type and, based on that, we can lower or increase the temperature.

Watch how, in both cases of the **`inc`** and **`dec`** action types, we return a new object and, in the default case, we return the previous state. That is because **`useReducer`** must abide by the same equality laws as **`useState`**: if we mutated something and want our component to re-render, we must return a new item. If we want to pass on a re-render, we can simply return the previous state.

EXERCISE 12.02: USEREDUCER WITH A SIMPLE FORM

We have explored **`useReducer`**, but let's put it into actual practice. In this exercise, we are going to demonstrate the versatility of **`useReducer`** to create simple, reusable hooks that could be used to design multiple different forms with multiple different rules:

1. Start off by creating a new React app:

```
$ npx create-react-app form-hooks
```

2. Go to the **`src/App.js`** file and delete **`logo.svg`**.
3. Replace the contents of **`src/App.css`** with the following:

```
.App {
  margin: 20px;
}

button {
  width: 200px;
  height: 50px;
  background: #4444ff;
  color: white;
  font-weight: bold;
  border: none;
  cursor: pointer;
}

input[type="text"] {
  height: 40px;
  width: 200px;
  margin: 5px;
}

ul.errors > li {
  color: red;
}
```

4. Replace the contents of **`src/App.js`** to get an empty component to work with:

```
const App = () => {
  return (
    <div className="App">
    <h1>Sample Form</h1>
  </div>
  );
};
```

5. Next, we will want to set up our first simple form, **`UsernameForm`**. Create **`src/UsernameForm.js`** and in it, we will add a text input, a header, and a validation button:

```
import React from "react";

const UsernameForm = () => {
  return (
    <div className="UsernameForm">
    <h3>Username Form</h3>
    <ul className="errors">
    <li>Sample Error</li>
    </ul>
    <input type="text" placeholder="Username" />
    <button>Validate</button>
  </div>
  );
};

export default UsernameForm;
```

6. Return back to **`src/App.js`** and import the **`UsernameForm`** component. Add the **`UsernameForm`** component to your **`App`** component:

```
import React from "react";
import "./App.css";

import UsernameForm from "./UsernameForm";

const App = () => {
  return (
  <div className="App">
```

```
    <h1>Sample Form</h1>
    <UsernameForm />
    </div>
    );
};

export default App;
```

Our component will start off looking like this:

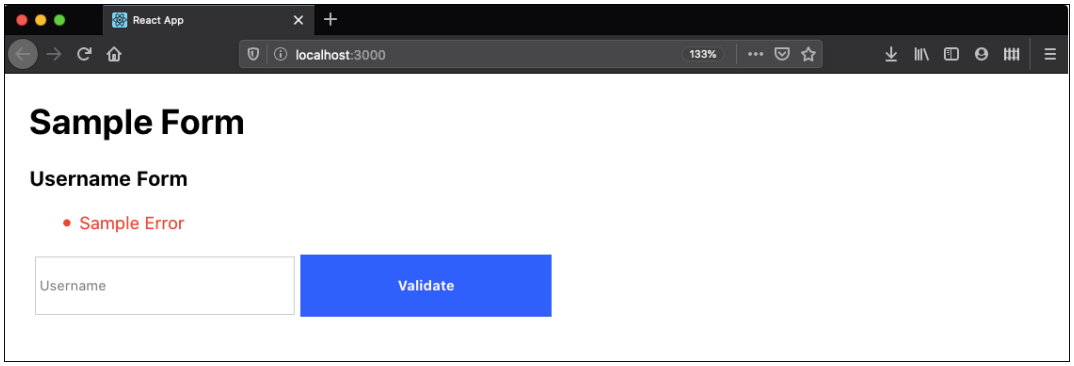

Figure 12.4: Sample Error

7. Let's start hooking up the **`UsernameForm`** component to an actual **`useReducer`** hook. We will create a new file, **`src/FormHooks.js`**, and, in that file, we will create a new **`useForm`** custom hook that will call **`useReducer`** to set up our **`state`** and **`dispatch`** function and return those. Right now, we will not have a specific action type that we will be listening to, so we will just set up a default case for our **`switch`** statement. In addition, our initial state will be a value (a blank string) and errors (an empty array):

```
import React from "react";

const useForm = () => {
  const [state, dispatch] = React.useReducer(
  (state, action) => {
  switch (action.type) {
  default:
  return state;
   }
  },
  {
  value: "",
```

```
    errors: []
    }
    );

    return { state, dispatch };
};

export { useForm };
```

8. Let's add our first action handler to our reducer. We will add a block in our **`switch`** statement to handle a **`change_value`** action type. This will use the object spread operator to return a modified object with a new value:

```
case "change_value":
  return { ...state, value: action.value };
```

9. We're actually going to be using this with every single form we ever create, so let's create a helper function called **`changeValue`** that will look in a standard JavaScript event object structure for the new value, which will be found in **`event.target.value`**. We will also need to change our **`return`** statement to return this new helper function:

```
  const changeValue = event => {
  const newValue = event.target.value;
  dispatch({ type: "change_value", value: newValue });
  };

  return { state, dispatch, changeValue };
```

10. Return to **`src/UsernameForm.js`** and let's start using the **`useForm`** custom hook in our component. We will need the state and the **`changeValue`** helper to start:

```
const { state, changeValue } = useForm();
```

11. Let's start using these values. We will begin by displaying all of the errors for the username form at the top:

```
<ul className="errors">
  {state.errors.map((error, index) => (
  <li key={`e-${index}`}>{error}</li>
  ))}
</ul>
```

12. Next, we will need to hook up the input to call the **`changeValue`** function, as well as use the state's value in the value prop of the input field:

```
<input
  type="text"
  placeholder="Username"
  onChange={changeValue}
  value={state.value}
/>
```

13. Without something to add or display errors, this isn't very helpful. Let's head back over to **`src/FormHooks.js`** and modify our **`useForm`** custom hook to have an **`add_error`** action type that it will handle that will modify the array of errors in the state:

```
case "add_error":
  return { ...state, errors: [...state.errors, action.error] };
```

14. Let's also create a generic **`validateInput`** function that will take in a function as its only argument and execute it against the value in the state. This will act as our primary validation method and will allow us to avoid having to expose the direct dispatch function.

    In this generic validation function, we will call the function and see what it returns. If it does not return anything, we assume that the validation passed. If it does return something, we assume it's an error message that we will dispatch with the **`add_error`** action type:

```
  const validateInput = fn => {
  const error = fn(state.value);
  if (error !== null) {
  dispatch({
  type: "add_error",
  error: error
  });
  }
  };

  return { state, changeValue, validateInput };
```

    We are now ready to hook this new function up to our **`UsernameForm`** component.

15. Return to **`src/UsernameForm.js`**, where we will add a **`validate()`** function to the component that will handle the different validation scenarios that our **`UsernameForm`** component cares about:

```
const validate = () => {
validateInput(v =>
  v.length < 3 ? "Username must not be under 3 characters!" : null
);
validateInput(v =>
  v.toLowerCase() === "test" ? "Cannot use a test username!" : null
);
};
```

16. As the last step for this form, we will add a call to validate to our button:

```
<button onClick={validate}>Validate</button>
```

17. Finally, visit **`http://localhost:3000`** in your browser and you should see the finished UI as follows:

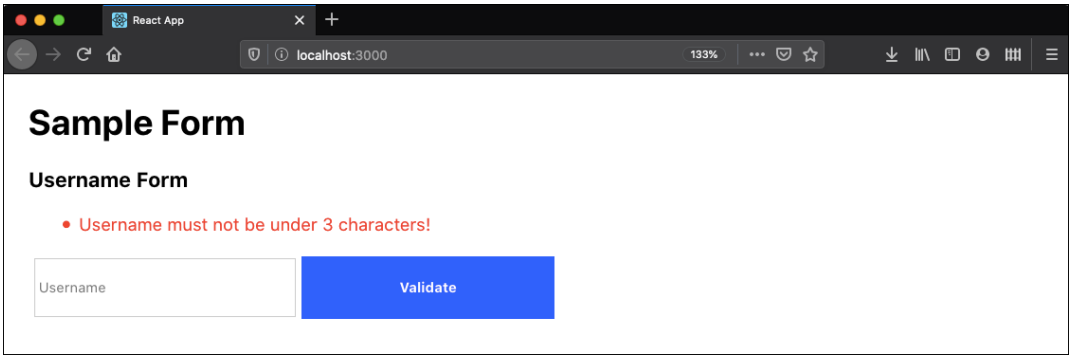

Figure 12.5: Sample Form

18. We're not 100% done yet. Right now, we have a bug where, every time you click on the **`Validate`** button, the form will add the same error message over and over. This is happening because we are not clearing out our errors in between each validation, so we need to go back to **`src/FormHooks.js`** and add a block to our switch statement to handle a **`clear_errors`** action type, which will set **`errors`** in the state to an empty array.

In addition, we will want to expose a helper **clearErrors** function in our return statement that **UsernameForm** will be able to use. The complete **src/FormHooks.js** file is as follows:

FormHooks.js

```
3  const useForm = () => {
4  const [state, dispatch] = React.useReducer(
5  (state, action) => {
6  switch (action.type) {
7    case "change_value":
8  return { ...state, value: action.value };
9    case "add_error":
10  return { ...state, errors: [...state.errors, action.error] };
…
22  const changeValue = event => {
23  const newValue = event.target.value;
24  dispatch({ type: "change_value", value: newValue });
25  };
26  const validateInput = fn => {
27  const error = fn(state.value);
28  if (error !== null) {
29  dispatch({
30  type: "add_error",
…
35  const clearErrors = () => {
36  dispatch({ type: "clear_errors" });
37  };
38
39  return { state, changeValue, validateInput, clearErrors };
};
```

The complete code of this file is found here: https://packt.live/2Z4dy3I

19. Finally, return to **src/UsernameForm.js** and add the **clearErrors** helper function to our **useForm** call, and add it as the first line in the **validate** function:

```
  const { state, changeValue, validateInput, clearErrors } =
useForm();
  const validate = () => {
  clearErrors();
  validateInput(v =>
  v.length < 3 ? "Username must not be under 3 characters!" : null
  );
  validateInput(v =>
  v.toLowerCase() === "test" ? "Cannot use a test username!" : null
  );
  };
```

The output is as follows:

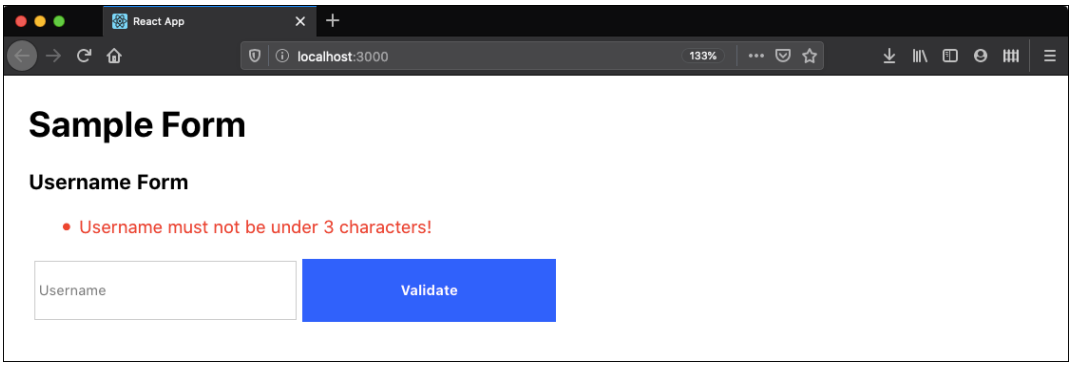

Figure 12.6: The SampleForm app

That's it. Our app is bug free and we have now used **`useReducer`** pretty extensively.

EFFECTS WITH CLEANUP

When you use the **`useEffect`** hook, one thing you need to be aware of is how your components are cleaning up after themselves. You can very easily get yourself into a scenario where your effects end up creating issues for you down the line. For example, what if you create an on-mount effect that sets up some sort of subscription or listener?

You need to make certain that the listener is being cleaned up appropriately, or you might run into a nasty memory leak that might end up being very difficult to diagnose and clean up later. The good news is that adding a cleanup to a React **`useEffect`** statement is incredibly simple.

Typically, the syntax for **`useEffect`** is:

```
React.useEffect(function, [props to watch]);
```

However, there is actually a little more to the syntax. The function that we pass in the preceding statement can actually optionally define a **`return`** statement, which will function as a cleanup function that will be run either when the watched property changes again or when a component is unmounted (if **`[]`** is specified as the prop to watch). For example, the following snippet sets up a subscription, and then the cleanup happens (pay close attention to this: it is a big hint for the activity you are about to undertake):

```
const ChatWindow = () => {
  React.useEffect(() => {
  const chatService = new ChatService();
  chatService.subscribe();


  return () => {
  chatService.unsubscribe(); // React will run this if the chat window
component
   is removed
  };
  }, []); // React will run this effect on mount and run the cleanup on
unmount
};
```

If you don't do this, **`chatService`** component in the preceding example, whatever functions happen in **`chatService.subscribe()`** will **continue** to run even after the component has been removed! This means that you can create a scenario easily where you keep adding new **`chatService`** subscriptions and never clean up the defunct ones.

With the knowledge gained from the chapter, let's step into the activity for this chapter:

ACTIVITY 12.01: CREATING A CHAT APP USING HOOKS

In this activity, we are going to demonstrate the right way to build an app using the **`useState`**, **`useEffect`**, and **`useReducer`** hooks, and remove any possible memory leaks along the way through the correct use of **`useEffect`** cleanups.

The best way for us to do this is by creating a chat app that will receive messages periodically when a user "joins" the chat and stops receiving the messages when the user closes the chat component.

First, let's analyze the technical requirements of this app before you get to building it. There are a few pieces of state we have to have in order to build that app:

- We must have a state for messages and whether the user is chatting.
- The messages must update when the user is listening, but not update when the user closes the window.
- We must show a log statement in the Console when the user joins the chat or when they leave.
- We must have a button to close the chat window, a button to clear the chat window (remove all messages), and a message display for all of the messages we have received.
- We must not introduce a memory leak by properly cleaning up our subscriptions.
- The user should receive a new message when appropriate every second.

The end result should be something like this. When the user first opens up the app:

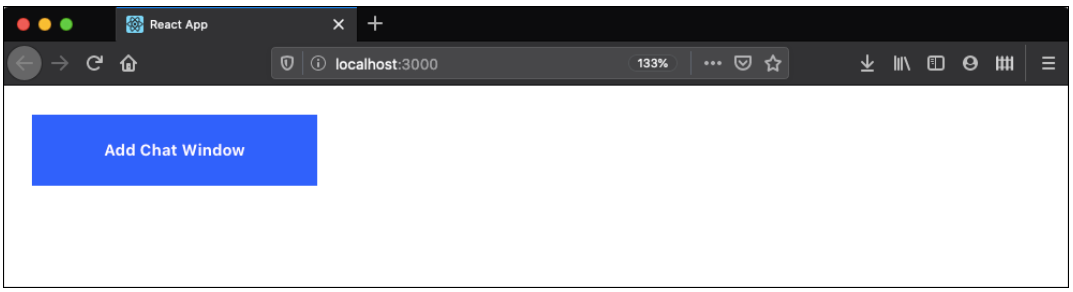

Figure 12.7: Chat window

When the user clicks **`Add Chat Window`**:

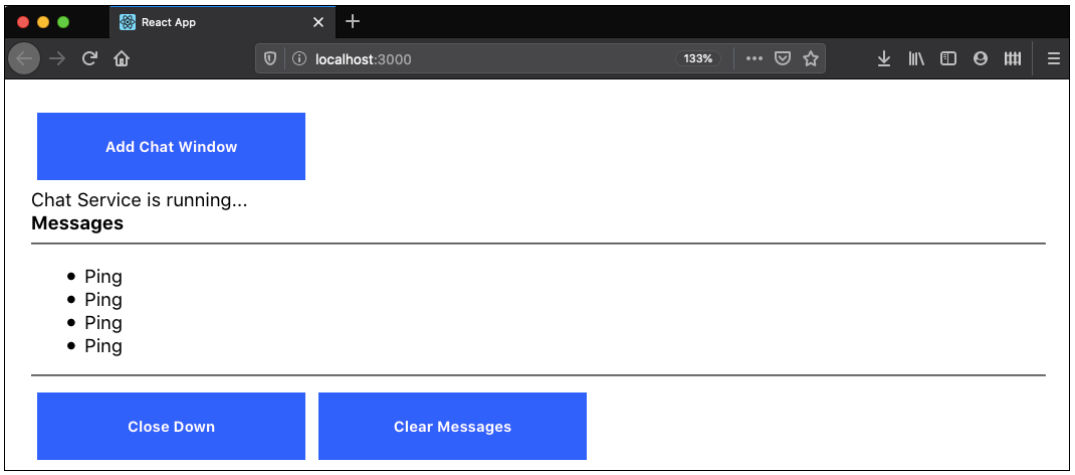

Figure 12.8: Chat window showing the ping messages

And when the user clicks **`Close Down`**:

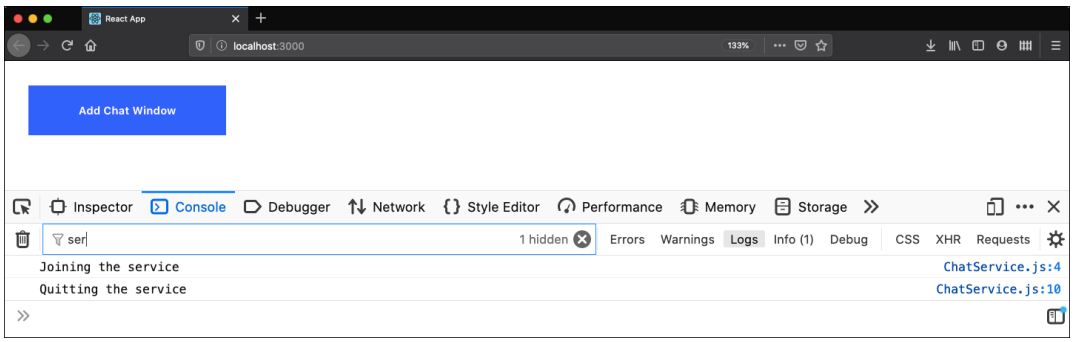

Figure 12.9: Chat window

The following steps will help you to complete the activity:

1. Start a new React application.
2. Set up the stylesheet you want to use.
3. Create your chat history window.
4. Create an initial reducer state for joining/leaving the chat.

5. Add more events to the reducer for adding a message to chat and clearing the messages from the chat.
6. Set up an interval timer that will dispatch an event to the reducer to add a message.
7. Add your buttons for adding the chat window, hiding the chat window, and clearing the chat messages.
8. Hook all of the buttons up and carry out any tweaks until you have the UI above.

> **NOTE**
>
> The solution of this activity can be found on page 697.

If you have been following along, you should now have a nice lean example of using **`useReducer`** and **`useEffect`** with cleanups. Our app is functional, has nice subscriptions to a chat service, and, more importantly, it has no class components and no memory leaks.

SUMMARY

This chapter provided us with a more in-depth look at state management in a less ad hoc basis through **`useReducer`**. **`useReducer`** gives us a lot of helpful state management via using functionalities such as actions and payload. Our state modifications can be controlled and managed in a way that would be difficult to implement using only the **`useState`** Hooks.

We also learned the proper way to clean up after ourselves to prevent the **`useEffect`** hook from getting us into a bad scenario where we subscribe to events and listeners and then never unsubscribe from them. You can experiment with this in your activity by removing the cleanup from your **`useEffect`** hook and seeing the result on your app.

In later chapters, we are going to dive even deeper into hooks, focusing on how to deal with shared context via Hooks to build great components with shared services.

13

COMPOSING HOOKS TO SOLVE COMPLEX PROBLEMS

OVERVIEW

This chapter will enable you to compose and use hooks with the Context API. You will be able to create hooks for app state management using Context API for UI-related components and to create well-encapsulated and scalable abstractions to handle and manage the app state. By the end of this chapter, you will have a solid understanding of how to use component libraries and logical units in hooks to solve complex problems.

INTRODUCTION

In the previous chapter, we saw the detailed approach of using Hooks inside a component. We extracted the component into a separate and unique hook so that we can reuse it elsewhere across multiple components. Then, we went on and externalized the logic and created our own custom hook that lived outside the component.

We have seen how every hook has its own utility and we have figured out how to apply them to suit our needs; and all of this without using class-based components, which can easily grow out of control if we want to modify or reuse them. When we build programs and applications, however, we realize that they are not isolated systems. We need to think about their global state management, dependencies, theming, and so on; for example, when you set up a theme in your project and you most likely want to use it throughout your entire app. That theme object gives consistency in every component. Alternatively, if you have an app that requires authentication, you might want the logged-in user to be able to access your entire app. Simply storing your state in one component provides a scalable solution this time.

In this chapter, we aim to look into each of these scenarios and we'll see how Hooks can come to our rescue. We'll use hooks and see how they can solve complex problems that we might face while designing a full-blown React application.

CONTEXT API AND HOOKS

Having to pass down props through multiple different levels of components can cause the following issues:

- While creating maintenance in your app, if you make a new prop, you need to pass it down the whole line. You might need to do the same if some component is changed or removed.
- It can create unnecessary noise if we do not know what props are intended for while creating a component and what is just a traveler.
- It creates a dependency on the component that is passing down the props. The component that just passes down the prop knows too much by simply playing the messenger.

To handle these issues, React typically allows access to something called **`Context`**, which we've briefly looked into in the previous chapters. React also provides a useful hook for us to be able to access the context of the app called the **`useContext`** hook.

USECONTEXT HOOK

useContext is a React hook that takes a React **context** object as an argument and gives us back the value provided. We can then save that value to a variable and access its fields associated with it. This variable prop can be accessed anywhere with the help of the **useContext** hook.

Let's take a look at how this would work in a simple example, where we'll have multiple-nested levels of React components. We are going to create three components, **<A />**, **<B />**, and **<C />**, and embed one component within the other, where **<A/>** will be the topmost component while **<C/>** is the lowest-level component. We'll put the information that we need in our **<C />** component, the lowest-level component, without having to pass props all the way down:

```
const InfoContext = React.createContext();
const C = () => {
  const value = React.useContext(InfoContext);
    return value.info;
  };
const B = () => {
  return <C />;
};
const A = () => {
  return <B />;
};
```

The context's **Provider** component can be used to provide the value to all React child components below this. In this case, the **<A/>** component uses the **Provider** component to set the value:

```
<InfoContext.Provider value={{ info: "important" }}>
  <A />
</InfoContext.Provider>;
```

In the preceding case, **<A/>** and **<B/>** are both now just rendering some JSX; the magic happens in **<C/>**. We mentioned earlier that **InfoContext.Provider** provides the value. We can use the **useContext** hook, set in the **<C/>** component, to retrieve it. Since value.info is a string, we can render it.

Let's practice this while implementing this in a shopping cart app. Shopping carts are pretty well-known and well-understood components that could be used in any sort of online store, which makes it easy for us to define how they work and how their interactions with the rest of the app should be defined. This is something that could be taken care of easily.

EXERCISE 13.01: ADDING CONTEXT TO THE SHOPPINGCART APP

In this exercise, we will create a **`ShoppingCart`** example. The **`<ShoppingCart/>`** component needs to be designed in isolation; what happens here should not be known in other parts of the app. For example, if we are going to modify the items in the cart, this should only be available to this component while the rest of the components should not have access to modify that state. That seems like a perfect use case for context. With our knowledge of **`createContext`** and **`useContext`**, let's refactor our **`<ShoppingCart />`** app.

In this exercise, the **`<Pay />`** component receives **`items`** and **`setItems`** as props. Secondly, **`<App/>`** renders **`<Cart/>`** and **`<Checkout/>`** directly:

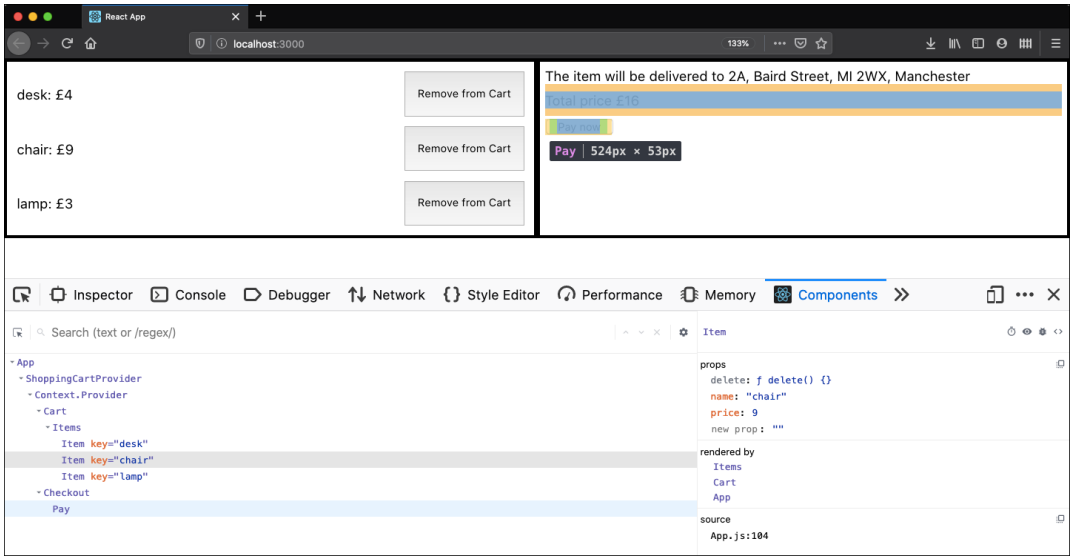

Figure 13.1: Final output of the shopping cart

1. Start off by creating a new React app:

```
$ npx create-react-app shopping
```

2. Go to the **src/App.js** file, delete **logo.svg** and **App.css**, and clear out the contents of **App.js**. Then, we will need to start off with an initial component in **src/App.js**:

```
import React from "react";
const App = () => {
  return (
    <div className="App">
    <h1>Shopping App</h1>
  </div>
  );
};
export default App;
```

3. Let's create a context that holds all the information about the **<ShoppingCart />** component and we'll call it **ShoppingCartContext**. This will go outside your **App** component definition in **src/App.js**:

```
const ShoppingCartContext = React.createContext();
```

4. Create the items in the shopping cart: **desk**, **chair**, and **lamp** with their prices in the App component. Add this to your **App** component:

```
const [items, setItems] = React.useState([
{ name: "desk", price: 4 },
{ name: "chair", price: 9 },
{ name: "lamp", price: 3 }
]);
```

5. Create the **createTransaction** function. This will be responsible for putting together the final transaction for everything added to the cart. We will start off just by verifying that the transaction is not equal to **0**. This will go inside your **App** component in **src/App.js**:

```
const createTransaction = amount => {
  if (amount !== 0) alert("transaction was created with " + amount);
};
```

6. Wrap everything the **<App/>** component returns in **<ShoppingCartContext.Provider/>**. That way, everything that is rendered inside **<ShoppingCartContext.Provider/>** will have access to what we put in the **<ShoppingCartContext.Provider/>** value prop:

```
return (
  <ShoppingCartContext.Provider value={[items, setItems]}>
  <div style={{ display: "flex", justifyContent: "space-between" }}>
  <Checkout createTransaction={createTransaction} />
  </div>
  </ShoppingCartContext.Provider>
);
};
```

This time, we put our state (**items**) and setter function (**setItems**) inside, wrapped in an array. We got rid of the props on **<Cart/>** and only **createTransaction** remains on the **<Checkout/>** component. We no longer need to pass items and **setItems** down as props.

7. Now, we will need to create a **<Checkout/>** component, which will start off being a bit empty (we'll add to it later):

```
const Checkout = props => {
return (
  <div
  style={{
  border: "3px solid black",
  padding: 5,
  flexGrow: 1,
  flexShrink: 1,
  flexBasis: "100%"
  }}
  >
  </div>
  );
};
```

8. Create a **`Pay`** component to handle when we need to pay for anything in our app. This will display a button with a little **`Pay now`** title on it, and when it is clicked on, it will try to call the **`createTransaction`** function we created earlier in the exercise. We will be passing this down directly via props:

```
const Pay = props => {
let totalPrice = 0;
return (
  <>
  <div style={{ margin: "8px 0" }}>Total price £{totalPrice}</div>
  <button
  title="Pay now"
  onClick={() => props.createTransaction(totalPrice)}>
  Pay now
  </button>
  </>
  );
};
```

9. Next, we need to add the **`Pay`** component to our **`Checkout`** component. This is where the **`createTransaction`** prop will get passed down:

```
const Checkout = props => {
return (
  <div
  style={{
  border: "3px solid black",
  padding: 5,
  flexGrow: 1,
  flexShrink: 1,
  flexBasis: "100%"
  }}
  >
  <Pay createTransaction={props.createTransaction} />
  </div>
  );
};
```

The **`<Checkout />`** component only received the **`createTransaction`** function as a prop. This means that we can only pass that to the **`<Pay />`** component.

10. Modify the **<Pay />** component to use the **useContext** code. This is a great example of where we can use **useContext** to pass services and allow each component in a hierarchy to opt into the data and information that it requires instead of having to drill props down entire chains of components.

    We will pass the **items** state and **setItems** setter function down and write a little logic on top of that **items** state that will allow us to calculate the total price of each item in our shopping cart.

    We will also call **setItems** with an empty array in our **pay now** button:

App.js

```
37  const Pay = props => {
38  const [items, setItems] = React.useContext(ShoppingCartContext);
39
40  let totalPrice = 0;
41
42  items.forEach(item => {
43  totalPrice = totalPrice + item.price;
44  });
```

The complete code of this step can be found here: https://packt.live/2WXoDAK

11. Let's build the **Cart** component now:

```
    const Cart = () => {
      return (
      <div
      style={{
      border: "3px solid black",
      padding: 5,
      flexGrow: 1,
      flexShrink: 1,
      flexBasis: "100%"
      }}
      >
      </div>
      );
    };
```

    This component will be responsible for storing all of the items that we add to the cart and displaying them to the user. Right now, we will leave this empty; we're going to be placing the items in this by building out an **<Items/>** component (that stores multiple **<Item />** components).

We will build those in a later step.

12. Return to the **App** component in **src/App.js** and add back the reference to **Cart**:

```
return (
<ShoppingCartContext.Provider value={[items, setItems]}>
    <div
    className="App"
  style={{ display: "flex", justifyContent: "space-between" }}
  >
  <Checkout createTransaction={createTransaction} />
  <Cart />
  </div>
</ShoppingCartContext.Provider>
  );
};
```

13. Remember, in the **Cart** component we discussed that we'd need a component to store the list of inventory items and then an **Item** component for each. Let's build out an **<Item />** component used to display each individual item as well:

App.js

```
78  const Item = props => {
79  return (
80  <div
81    className="Item"
82    style={{
83    display: "flex",
84    justifyContent: "space-between",
…
91    <p>
92    {props.name}: {props.price}
93    </p>
94    <button onClick={() => props.delete(props.name)}>Remove from Cart</button>
95  </div>
```

The complete code of this step can be found here: https://packt.live/2T4odYo

14. Build out an **<Items />** component that will store and display each inventory item that we can purchase in the store:

```
const Items = () => {
  const [items, setItems] = React.useContext(ShoppingCartContext);
  return items.map(item => {
  return (
  <Item
  key={item.name}
```

```
      name={item.name}
      price={item.price}
      delete={() => {
        setItems(prevItems => {
        return prevItems.filter(i => i.name !== item.name);
        });
      }}
      />
  );
  });
  };
```

This will use the **`items`** state and **`setItems`** setter function from the context again, and just use a looping display of **`Item`** components.

15. Hook everything up by adding the **`Item`** component to the **`Cart`** component now:

```
const Cart = () => {
  return (
    <div
    style={{
    border: "3px solid black",
    padding: 5,
    flexGrow: 1,
    flexShrink: 1,
    flexBasis: "100%"
  }}
  >
  <Items />
  </div>
);
};
```

We talked about how the best way to use the Context API is to create well-defined logical units. To take full advantage of this strength, we should be smarter about where we store the state, and that means not storing it directly in the **`App`** component, as that would make our state less reusable.

This means that **<ShoppingCartContext.Provider />** and the **useState** hook should not be in the **<App/>** component; instead, we'll build our own **ShoppingCartProvider** component and use that as a wrapper component, allowing us to keep our logic centralized and logically grouped.

16. Let's create a **ShoppingCartProvider** component that will store everything related to the shopping cart:

```
const ShoppingCartProvider = props => {
  const [items, setItems] = React.useState([
  { name: "desk", price: 4 },
  { name: "chair", price: 9 },
  { name: "lamp", price: 3 }
  ]);

  return (
    <ShoppingCartContext.Provider value={[items, setItems]}>
    {props.children}
  </ShoppingCartContext.Provider>
  );
};
```

17. Finally, replace the JSX in **App**, which wraps everything in **ShoppingCartContext.Provider** with our new wrapper component, **ShoppingCartProvider**:

```
const App = () => {
  const createTransaction = amount => {
  if (amount !== 0) alert("transaction was created with " + amount);
  };
  return (
  <ShoppingCartProvider>
  <div
  className="App"
  style={{ display: "flex", justifyContent: "space-between" }}
  >
  <Cart />
  <Checkout createTransaction={createTransaction} />
  </div>
  </ShoppingCartProvider>
  );
};
```

Looking at the new version of our **`App`** component, we can see how much cleaner our code is. We don't need to store the **`useState`** call inside our **`App`** component; we allow the context provider to store everything it needs.

The output is as follows:

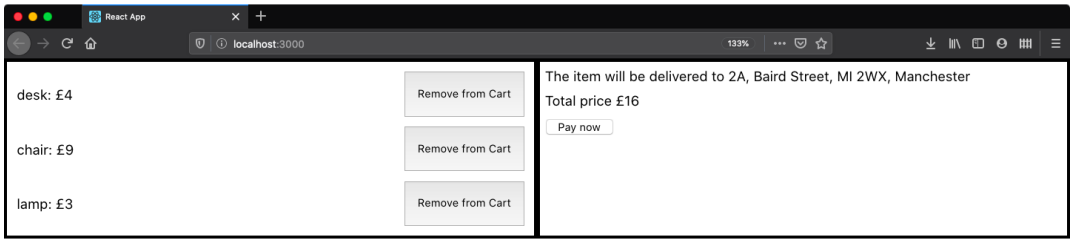

Figure 13.2: Output showing the app

As we can see from the preceding exercise, it is clearly defined how the **`Cart`** and **`Items`** components can both take advantage of the context in a logical way. They are in the hierarchy under the **`ShoppingCartProvider`** component, and thus have access to its context. This makes it incredibly simple to reason about our components without having to deal with messy prop chains traveling through multiple levels of components.

PROPS AND CONTEXT

After seeing context in action with the **`useState`** and **`useContext`** hooks, this is a good place to briefly talk about props versus context.

When to use props and when context? There are no hard and fast rules here, but, as a general guideline, use props for customizing React components and use context if you want to use services within a React component.

A better way to think about this is on the basis of where you might have something affect just the component itself versus something that might be a shared thread amongst multiple different components. Context makes sense with components that will need to share states or information at multiple different stages along the component hierarchy, but props really only make sense (and are best used) when they're just going to be used in one component, and not passed down in a chain to each of its children. Look at the example in the previous exercise: we use context to manage the cart/items components that multiple components need access to but need to be able to opt in to at each component individually. The props we rely on are pieces of data that customize each individual component – the name or price of the **`Item`** component, for example.

PROPS FOR CUSTOMIZATION

Let's talk about when we would want to use props for customization. A hypothetical **<Text/>** component can appear all over your app, but in some places, you want a small size, whereas somewhere else, you want the color to be red. Your props are the API to make your **<Text/>** customizable.

In one particular place, it may be like this:

```
<Text size="small" color="violet">
  greetings
</Text>
```

In another, the following is what we may want:

```
<Text size="large" color="red">
  greetings
</Text>
```

There is another example that we are actually going to use and build a bit later. Imagine in the **<Checkout />** component that we can have an address (such as a shipping address to send the package to):

```
<Address
  streetName={user.address.streetName}
  houseNumber={user.address.houseNumber}
  postalCode={user.address.postalCode}
  city={user.address.city}
/>
```

The output will look like this:

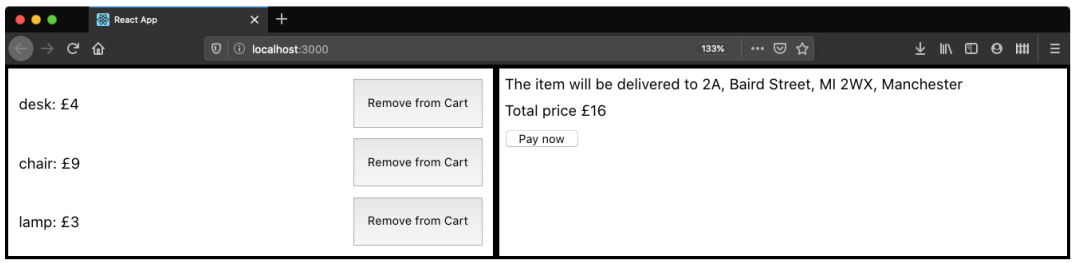

Figure 13.3: Address component

This component could be used elsewhere, for example, in your profile, where you can save multiple shipping and billing addresses. To make the **`Address`** component reusable and customizable, we can expose props on them, such as **`postalCode`**, **`country`**, **`street`**, and **`city`**.

Therefore, the general rule when it comes to customizing a component is that we should use props.

ANOTHER EXAMPLE: THEMING AS A SERVICE

Imagine you have a theme set up in your app where all the colors are set up for you. That way, you, as a developer, no longer have the option to pass color to your text. The color is defined by the theme, but you might need certain variants in relation to the text, such as the primary or secondary text. The following code snippet is an example of a theme context using **`createContext`**:

```
import React from "react";
const ThemeContext = React.createContext();
const theme = {
  primary: "blue",
  secondary: "red"
};
const Theme = props => {
  return (
  <ThemeContext.Provider value={theme}>
  {props.children}
  </ThemeContext.Provider>
  );
};
const Text = props => {
  const theme = React.useContext(ThemeContext);
  let color = theme.primary;
  if (props.variant && theme[props.variant]) {
  color = theme[props.variant];
  }
  return <p style={{ color }}>{props.children}</p>;
};
```

```
const App = () => {
  return (
  <Theme>
    <div className="App">
    <Text variant="primary">Some Primary Text</Text>
    <Text variant="secondary">Some Secondary Text</Text>
    </div>
  </Theme>
  );
};
export default App;
```

The resulting component should look like this:

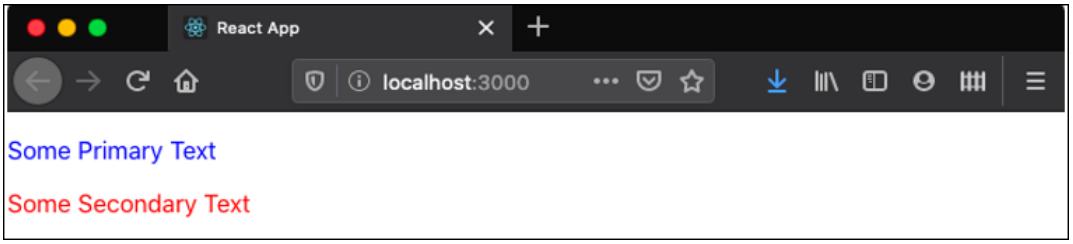

Figure 13.4: Shopping cart provider app

This time, theming is a service, and thus suitable for context; if it were a one-time customization, a prop would be a better fit.

Let's see what is happening in the preceding code above. We wrap our **`<App />`** component with the **`<Theme />`** component, which we set up as the context provider (similar to **`ShoppingCartProvider`** in the previous exercise), making that **`Theme`** context available to any components underneath it. We use it in two **`Text`** components, but we could use it in any component that requires access to the **`Theme`** component, and it allows the **`Text`** components to access the **`Theme`** context regardless of where they are in the app's hierarchy of components.

Imagine if every single component that had to use the theme had to pass these down as a prop chain, drilling through multiple levels; every component declaration and JSX would be messy, cluttered, and borderline impossible to read clearly.

We have one more exercise worth running through regarding our shopping cart. Using the same code from our previous exercise (*Exercise 13.01*, *Adding Context to the ShoppingCart App*), we are going to build on that to showcase how to properly build context and use it with **`useEffect`**.

> **NOTE**
>
> Remember that **`useEffect`** exists as our replacement for component life cycle methods in class-based components. We're going to stick entirely with functional components, so to do that, we'll need to use **`useEffect`** instead of something like **`componentDidMount`**.

EXERCISE 13.02: CREATING CONTEXT USING USECONTEXT AND USEEFFECT

In this exercise, we are going to create an **`AuthService`** component that will simulate getting a logged-in user and will store the mailing addresses for the user. This will be similar to when you visit an e-commerce site and it needs to get the context of where to ship items from your user account. These are typically loaded asynchronously, so they are a good candidate to load via **`useEffect`**, instead of tying the entire page load to waiting for the user data to be loaded. This user and their addresses are going to come from our clever use of context and **`useEffect`**; we will build all of this through an **`AuthService`** component. Let's go through the following steps in order to do so:

1. Let's begin by building our **`AuthService`** component. We will create a new file for this, **`src/AuthService.js`**, and we will build our context using **`React.createContext`**:

```
import React from "react";
const AuthContext = React.createContext();
```

2. Next, create our **`AuthService`** wrapper; this will be our context wrapper, just like we have created in previous exercises and examples. It will require access to a user state and **`setUser`** setter function, so we will use **`React.useState()`** for that and start off with a null value. We will also need to export both **`AuthService`** and **`AuthContext`** since these will be what other components will require later:

```
const AuthService = props => {
  const [user, setUser] = React.useState(null);
  return (
```

```
    <AuthContext.Provider value={user}>{props.children}</AuthContext.
Provider>
    );
};
export { AuthService, AuthContext };
```

3. Now, let's set up our call to **`setUser`**, which we'll wrap inside a JavaScript **`setTimeout`** call to simulate a longer async call. We will set our **`useEffect`** to execute upon component mounting, which we do by adding an empty array as the watched properties for the effect.

   For **`setTimeout`**, we will use **`2000`** as the value (remember that **`setTimeout`** and **`setInterval`** expect a timer value in milliseconds, so that would be equivalent to **2** seconds).

   We'll use a few sample addresses that we'll set as the user data as well. Let's now start writing the rest of this **`AuthService`** component:

`AuthService.js`

```
5  const AuthService = props => {
6  const [user, setUser] = React.useState(undefined);
8  React.useEffect(() => {
9  setTimeout(() => {
10  console.log("Time to set the user");
11  setUser({
12  addresses: [
13    {
14    postalCode: "M1 2WX",
15    city: "Manchester",
16    streetName: "Baird Street",
17    houseNumber: "2A",
18    main: true
```

`The complete code of this file can be found here:` https://packt.live/2T24sAw

4. Let's add the **`AuthService`** component to our **`<App />`** component. The **`AuthService`** component should be available in our entire app, just like **`<ShoppingCarProvider />`**.

5. Return to **`src/App.js`**, include the **`AuthService`** imports, and import both **`AuthService`** and **`AuthContext`**, and then wrap our **`ShoppingCartProvider`** component inside the **`AuthService`** component. First, the **`import`** statement:

```
import { AuthService, AuthContext } from "./AuthService";
```

6. And then our **App** component changes:

```
const App = () => {
  const createTransaction = amount => {
  if (amount !== 0) alert("transaction was created with " + amount);
  };
  return (
  <AuthService>
  <ShoppingCartProvider>
  <div
    className="App"
    style={{ display: "flex", justifyContent: "space-between" }}
  >
    <Checkout createTransaction={createTransaction} />
    <Cart />
  </div>
  </ShoppingCartProvider>
  </AuthService>
  );
};
```

7. Now, let's create the **<Address />** component. Create a new file, **src/Address.js**, and then create a simple component that will display **Address** information to the user:

```
import React from "react";
const Address = props => {
  return (
  <div className="Address">
  <div>{`${props.city}, ${props.postalCode}`}</div>
  <div>{`${props.houseNumber}, ${props.streetName}`}</div>
  </div>
  );
};
export default Address;
```

8. We have the **AuthService** and the **<Address />** components in place, so now it's time to combine them in the **<Checkout />** component.

   First, we grab the user by using the **useContext** hook and pass **AuthContext** as an argument. In the **AuthService** component, we initially set the user to **null** and use the **useEffect** hook combined with **setTimeout** to set the user to an object after 2 seconds.

   This means that the user returned from our **useContext** hook is not going to be available for **2** seconds, which is why we check whether the user is available when we try to find the main address. If the user is set to a non-null value, we grab the user's addresses (which is an array) and return the main address with the help of the **find** method:

```
const Checkout = () => {
  const user = React.useContext(AuthContext);
  const mainAddress = user && user.addresses
          ? user.addresses.find(address => !!address.main)
  : null;
  return (
  <div
  style={{
  border: "3px solid black",
  padding: 5,
  flexGrow: 1,
  flexShrink: 1,
  flexBasis: "100%"
  }}
  >
  {mainAddress ? <Address {...mainAddress} /> : "No addresses
found!"}
  <Pay />
  </div>
  );
};
```

   The next step is to implement the **settings** page.

9. Let's create **`src/Settings.js`**, where we'll need to import the **`AuthContext`** and **`Address`** components from elsewhere. In this component, we'll use the same user context we've used before, and then we'll just display a few headers indicating that it's the address displayed by the **`Settings`** component:

`Settings.js`

```
6  const Settings = () => {
7  const user = React.useContext(AuthContext);
8
9  return (
10  <div style={{ marginTop: 24, textAlign: "center" }}>
11  <h1>User Settings</h1>
12  <h2>Addresses</h2>
13  <hr />
14    <div
15      style={{
16      display: "flex",
17      justifyContent: "space-between",
18      padding: 16
19    }}
20  >
21  {user
22    ? user.addresses.map((addr, index) => (
23    <Address key={`address-${index}`} {...addr} />
24    ))
25    : "loading..."}
```

The complete code of this step can be found here: https://packt.live/3dLqlw8

10. Finally, we are ready to add the **`Settings`** component to our **`App`** component. Up at the top, add an **`import`** for the **`Settings`** component:

```
import Settings from "./Settings";
```

11. And finally, add the **`Settings`** component to the **`App`** component:

```
const App = () => {
  const createTransaction = amount => {
  if (amount !== 0) alert("transaction was created with " + amount);
  };
  return (
    <AuthService>
    <ShoppingCartProvider>
    <div
```

```
        className="App"
        style={{ display: "flex", justifyContent: "space-between" }}
        >
        <Cart />
        <Checkout createTransaction={createTransaction} />
      </div>
      <Settings />
      </ShoppingCartProvider>
      </AuthService>
      );
  };
```

The end result here is that while our component is loading, we will see the following UI:

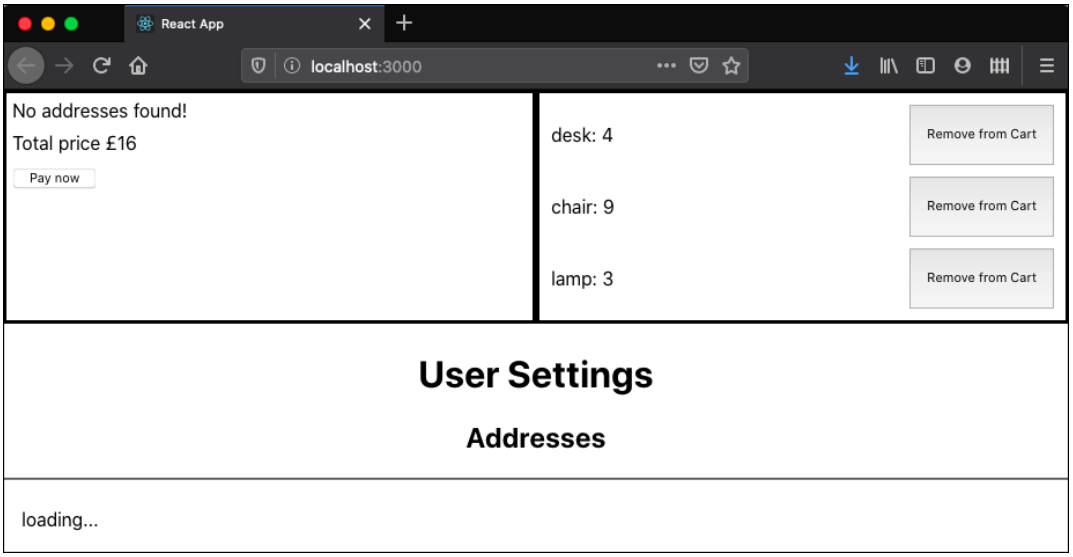

Figure 13.5: Output while the component is still loading

And then, after the component finishes loading, we'll see the addresses displayed correctly:

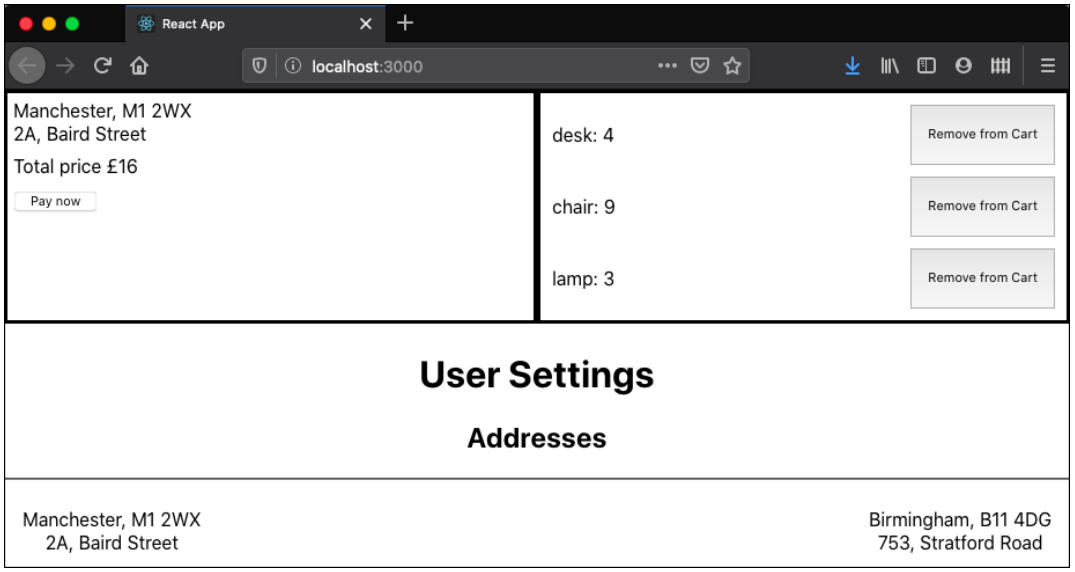

Figure 13.6: Output after the component is loaded

With our working application, we have put **`useState`**, **`useContext`**, and **`useEffect`** together to create the kind of complicated component where we previously had to rely on class components and all of the different life cycle methods. Knowing all of this, we are ready to apply this knowledge and tackle our final activity for Hooks.

ACTIVITY 13.01: CREATING A RECIPE APP

In this activity, we're going to build a **`Recipe`** app that will display two sections: on the left-hand side, we'll have our list of recipes to choose from, and on the right, we'll display the recipe to the user. We will use **`useEffect`** and **`setTimeout`** to simulate the effect of loading the recipes from a server, and we'll use **`useState`** to set up the recipe state. We will also use **`useContext`** and **`createContext`** to build the **`Recipe Context`** that will provide access to the recipes in each component.

The following steps will help you to complete the activity:

1. Create the **`App`** component and give it the same flexbox styling we have used throughout this chapter.
2. Create the **`RecipeList`** component.

3. Create the **`Recipe`** component.
4. Add the **`RecipeList`** and **`Recipe`** components to the **`App`** component.
5. Set up a **`RecipeContext`** and **`RecipeService`** wrapper that will load a list of recipes after **2** seconds and create a state, both for the list of recipes and for the selected recipe. Remember to display a loading message in the recipe list while that **2** seconds is loading.
6. Expose the recipe list context and the set **`recipe`** setter function in our **`RecipeList`** component. This should only use **`RecipeContext`**.
7. Display the selected recipe when a recipe is clicked in the **`RecipeList`** component. This should use everything from **`RecipeContext`**.

The output should appear as follows:

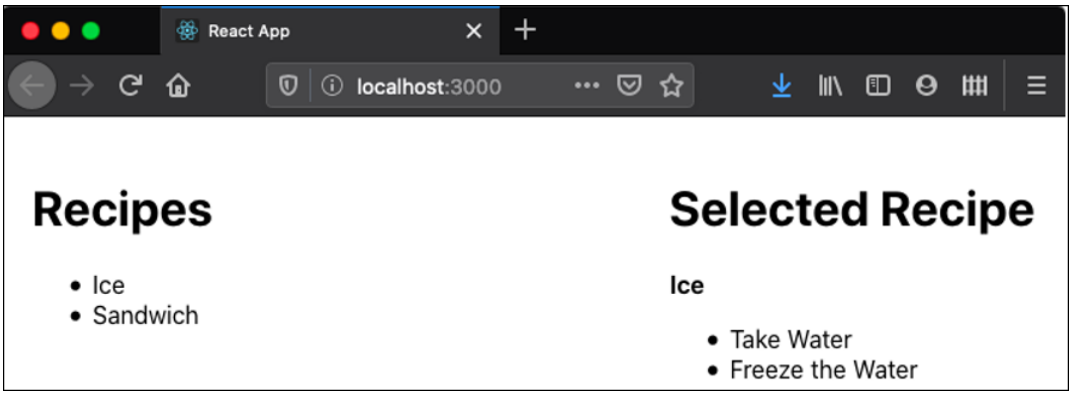

Figure 13.7: Final component output

> **NOTE**
>
> The solution of this activity can be found on page 707.

You have now successfully completed this activity.

SUMMARY

In this chapter, we have successfully built a complicated app using the **`useState`**, **`useEffect`**, and **`useContext`**/**`createContext`** hooks. We have learned about the importance of each and what sorts of problems these hooks provide solutions for. Most importantly, we have completely avoided class-based components in the entire process of implementing these complex components, each requiring states and data load/data modification for each event.

Through **`useEffect`**, we have learned how to modify the context to expose a global context to each component that needs to use it without having to resort to deep prop chaining. The data concerns for each component are now optimized, without any additional bloat in our component definitions, which is the ideal state for us. Our components are clean, lean, and very reusable.

In the next chapter, we will explore various external APIs in React in order to fetch data from the servers.

14

FETCHING DATA BY MAKING API REQUESTS

OVERVIEW

This chapter will introduce you to the various ways to fetch data by making API requests in React. You will do some hands-on exercises on fetching data from servers using RESTful APIs, the Fetch API, and Axios. We will compare these three with various parameters and we will see how Axios is better than the rest. By the end of this chapter, you will have a good understanding of why the Axios API is preferred when we make API requests in React.

INTRODUCTION

Most modern apps receive data from servers and dynamically update the content. For example, a weather application will make requests to receive today's weather data from its servers. With React apps, it is also essential to communicate with servers in order to get the requested data to display content dynamically based on the user's interaction with the app.

In the previous chapters, we have seen how two components communicate with each other in React using render props and hooks. In this chapter, we will look at multiple ways to communicate with the outside world using APIs in React.

We will discuss the commonly used RESTful API, usually described as a web service that implements REST architecture (a convention for building **`HTTP`** requests with **`HTTP`** methods). After initiating a server request using the **`REST API`**, we will test the server by making API requests with Postman, including **`GET`**, **`POST`**, **`PUT`**, **`PATCH`** and **`DELETE`** requests. After you understand all the essential parts of making API requests, we will write a simple app to receive data from a server.

We will also discuss different ways of requesting data from servers dynamically in React, including **`XMLHttpRequest`** and the **`Fetch API`**. We are also going to learn about a third-party library called **`Axios`** that helps you to fetch data from servers in a better way than the native **`Fetch API`**.

Before beginning our discussion on the various APIs in React, let's first look at the basics of making HTTP requests to interact with the server.

RESTFUL API

When we develop any application with React or JavaScript, we almost always need to request data from a server to dynamically update content inside our application. For example, any social media platform, such as Twitter, Facebook, LinkedIn, Instagram, or YouTube, updates some parts of their content on the page by requesting data from their servers. The content on those platforms is never static.

To request data from a server, there are several web services you can use, such as **SOAP**, **WSDL**, and **REST**. In this chapter, we are going to focus on the popular web service **REST** (or RESTful). We are going to talk about what a RESTful API is and how it works and will look into a few practical examples.

REST stands for Representational State Transfer, and it is an architectural style of web architecture with six constraints that allows us to request data via the HTTP protocol using, for instance, **GET** and **POST**. When something is RESTful, it means it follows the REST constraints, and you can think of RESTful as the adjective of REST. Therefore, a RESTful API can be defined as an API that uses the HTTP protocol to request using **GET**, **POST**, and **DELETE**.

When the client needs to talk to the server to receive or save data, these requests happen using the HTTP protocol, which exposes services from the server, and then the client can directly call the services by sending HTTP requests. This is where REST comes into play.

> **NOTE**
>
> There is another popular API-based declarative query language called GraphQL. GraphQL is a syntax for querying APIs by allowing clients to specify what data it needs, and it can aggregate data from multiple places easily. To find out more, please check out its documentation page with JavaScript at https://graphql.org/code/#javascript.

FIVE COMMON HTTP METHODS

In the following section, we will look at five common HTTP methods that you must provide when you request data through a RESTful API. The following table summarizes the five common HTTP methods – CRUD (Create, Read, Update, Delete):

HTTP Methods	CRUD	Description
GET	Read	Fetch all or any data
POST	Create	Add new data
PUT	Update	Update existing data entirely
PATCH	Update	Update existing data partially
DELETE	Delete	Delete data

Figure 14.1: Header table

A lot of developers get confused with the **`PUT`** and **`PATCH HTTP`** methods, so let's take a closer look at them.

PUT VERSUS PATCH

Both **`PUT`** and **`PATCH`** update existing data but **`PUT`** entirely updates the existing data whereas **`PATCH`** partially updates the existing data. To better explain, let's have a look at an example.

Let's say we have this **JSON** data:

```
{
  "name": "Ryan",
  "age": "29",
  "city": "Sydney",
  "fav_language": "React"
}
```

When we **PUT** the new data:

```
{
  "age": "30"
}
```

The outcome will only have the **age** and nothing else:

```
{
  "age": "30"
}
```

Now, if we apply this update with the **PATCH** method, the result will still contain other details but will only update **age**:

```
{
  "name": "Ryan",
  "age": "30",
  "city": "Sydney",
  "Fav_language": "React"
}
```

Thus, as we can see from the example, the **PUT** method overwrites the entire content and the **PATCH** method only updates the specified property.

Therefore, wherever we want to make minor modifications to a resource, we use **PATCH**, whereas **PUT** is used when we want to completely rewrite the data in a resource.

Now, when you make a network request, you must have often noticed the status messages that get displayed on the web page when a request is completed. Let's take a look at the various types of HTTP status codes and messages that get returned.

HTTP STATUS CODES

REST APIs respond to a request with a status code to inform developers of the status of the request. There are **40** standard status codes, but the following table shows you the 7 most common status codes you will see with REST APIs:

Status Codes	Description
200 OK	Requested data and successfully found it.
201 Created	Successfully created a resource.
204 No Content	Successfully deleted a resource.
400 Bad Request	Passed bad data to the server.
401 Unauthorized	Passed invalid Auth token to the server.
404 Not Found	Resource not found.
500 Internal Server Error	The server encountered an error and needs to be requested again later.

Figure 14.2: HTTP status codes

You must have seen **404** quite often when there is a failure in retrieving the requested data. This simply means the resource was not found at the defined web address. For example, when an Axios **GET** request successfully fetches data, the status code shown on the console is **200**, which means the data was successfully retrieved.

ACCEPT HEADER AND CONTENT-TYPE HEADER

When we request data from a server, we include **HTTP** headers in a **REST** request to tell the server the format of data the client is expecting. Among many headers, **Accept** and **Content-Type** are the two main headers for formatting the data properly, and we will look closer look at these two headers.

The difference between the **`Content-Type`** and **`Accept`** is:

- **`Accept`** is used by HTTP clients when requesting data from a server and tells the server what data format the client is expecting.
- The server will send back the response including the **`Content-Type`** header telling the client what format the response is returned with.

To look at the details of headers, open the **`DevTools`** (by pressing *F12*), go to the **`Network`** tab, and refresh the page to request the data. Let's write the following code in the Console:

```
const xhr = new XMLHttpRequest();
// assign json
xhr.responseType = 'json';
// initialize the request
xhr.open('GET', 'https://jsonplaceholder.typicode.com/posts/1');
// send request to the server
xhr.send();
```

We will see something like this on the screen:

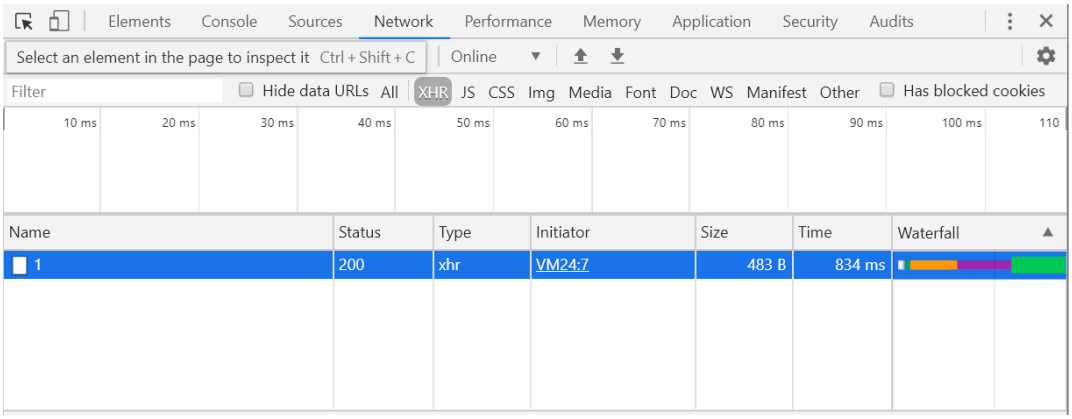

Figure 14.3: Checking network in Chrome DevTools

When you click **`XHR`** and **`1`** in the **`Name`** panel, you will see the **`Headers`** tab. Clicking on it gives you all the header details you have included with your request.

Under **`Response Headers`**, you will see the **`content-type`** header. Here, you should get

`application/json; charset=utf-8`

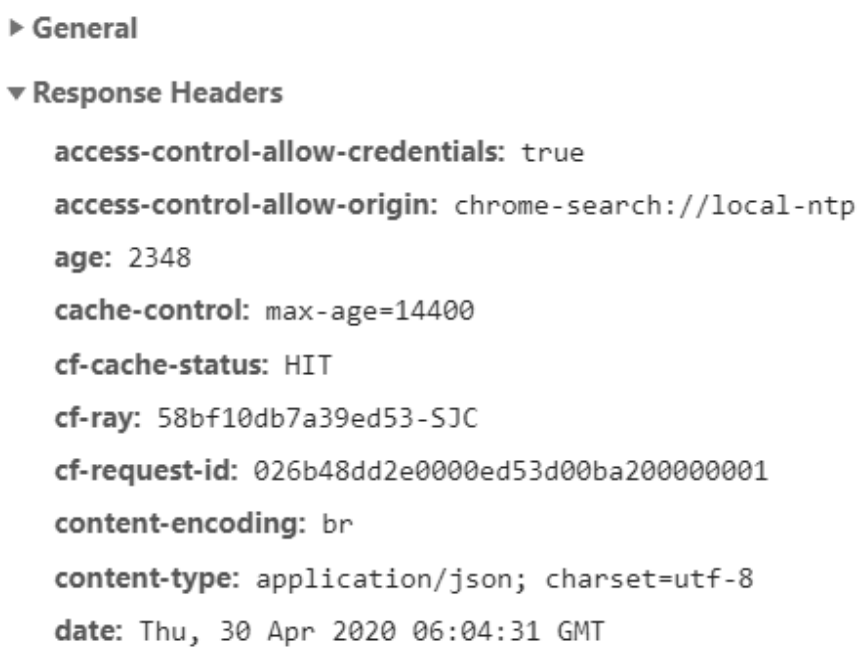

Figure 14.4: Content-type header

Under the **`Request Headers`**, there will be **`Accept`** header with **`*/*`** which is any MIME type as we can see in the following figure:

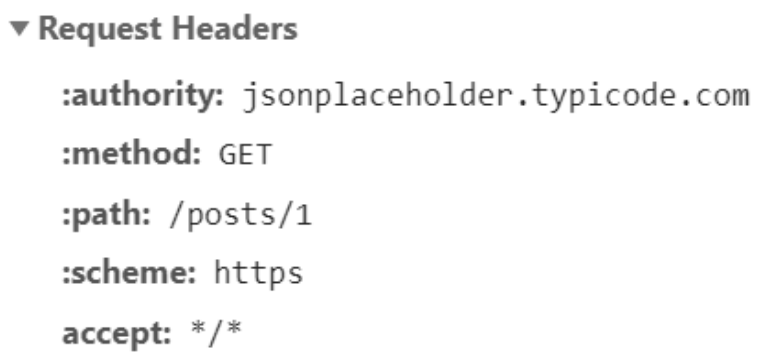

Figure 14.5: Accept header

From *Figure 14.5*, we can see that the client is giving the server a hint that the client is expecting the response in the format of **`*/*`** and if the server cannot handle the format, it'll return anything it can respond with. However, it is up to the server to handle it. In *Figure 14.4*, the **`content-type`** header has returned with **`application/json`**. When the content-type header is included in the request to the server and when the server handles it correctly, the response should come back with the content-type the client had asked for.

Giving the server a hint about the data format gives it a chance to properly handle the data format and return the response in the expected format.

With all this knowledge, let's dive right into the different ways of requesting data from the server.

DIFFERENT WAYS OF REQUESTING DATA

In most applications, it is very common to request data from a server and display it on a web page dynamically. In modern web applications, clients such as browsers request data from a server and only load content partially on the page. That prevents reloading the entire page to display the new data or content and provide a better user experience. For example, Tumblr is a social networking website with a huge database running in the backend. However, when we fetch data, there isn't a lot of loading time involved. The reason for this is the partial loading of the content. As the user scrolls down a Tumblr page, the content is loaded dynamically. Let's look at the following diagram:

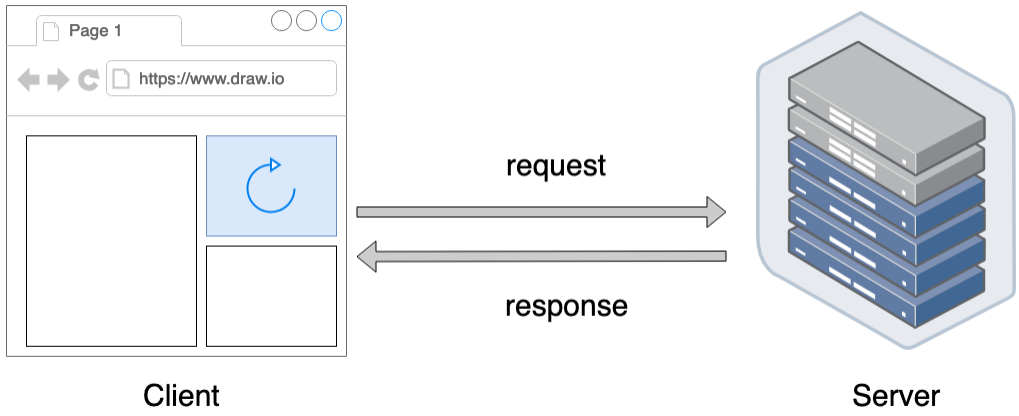

Figure 14.6: Requesting data for partial update on the browser

We can see in the previous diagram that only partial data is being loaded onto the web page. The different methods used to fetch data from a server are:

- **`XMLHttpRequest`**
- **`Fetch API`**
- **`Axios API`**

Throughout the next section, we are going to learn about the different ways to request data and decide which one we are going to use.

XMLHTTPREQUEST

`XMLHttpRequest` is used to fetch data in the form of not only **`XML`** but also **`JSON`**, **`HTML`**, and **`plain text`**. **`XMLHttpRequest`** is made using an **`XMLHttpRequest`** (**`XHR`**) object. An **`XMLHttpRequest`** object allows us to transfer data between a web browser and a web server.

In this section, we will look at the basics of requesting and receiving data from a server using **`XMLHttpRequest`**.

> **NOTE**
>
> XMLHttpRequest (XHR) was created by Microsoft in the late 1990s.

Let's see how to initiate **`XMLHttpRequest`**:

1. The first step is to create an instance of **`XMLHttpRequest`** object:

```
const xhr = new XMLHttpRequest();
```

2. Then, to specify the type of data contained in the response, we can use the **`responseType`** property. The supported values are an empty string, **`arraybuffer`**, blob, document, JSON, and plaintext.

   To use **`JSON`**, add the property to the instance and assign the value:

```
xhr.responseType = 'json';
```

3. Once we have created the instance of the **XHR** object, we need to set the connection to a server by using the **open()** method. The **open()** method requires two mandatory parameters, which are a method such as **GET**, **POST**, **PUT**, or **DELETE**, and the URL to send the request to:

```
xhr.open('POST', 'https://yourdomain.com/api/posts/new');
```

   If you need to set any HTTP headers, call the **setRequestHeader** method with the header and value. The **setRequestHeader** method must be called after **open()** but before **send()**:

```
xhr.setRequestHeader('Content-Type', 'application/json');
```

4. It's time to send a request. To send a request, we need to use the **send()** method with the body of the data:

```
xhr.send(JSON.stringify({
  title: 'this is a title',
  body: 'this is body'
}));
```

   When sending data to a server, the format of the data should be a string. The **JSON.stringify()** method converts a JavaScript object to a string.

5. To get the data, we can call the load event through the standard **addEventListner()** API to **XMLHttpRequest**. We use the load event because the event is waiting for all the resources on the page is finished loading before it gets fired:

```
xhr.addEventListener('load', res => {
  console.log(res);
});
```

   If we put all the code in one place, it should look like this:

```
const xhr = new XMLHttpRequest();
//
xhr.open('POST', 'https://yourdomain.com/api/posts/new');
xhr.setRequestHeader('Content-Type', 'application/json');

xhr.send(JSON.stringify({
  title: 'this is a title',
  body: 'this is body'
}));
```

```
xhr.addEventListener('load', () => {
  console.log(xhr);
});
```

The response depends on what the server returns, but in **JSON** format, it would look like this:

```
{
  id: 12,
  title: 'this is title'
}
```

Now that we have seen an example implementation of an **XMLHttpRequest**, let's do an exercise to see how to use **XMLHttpRequest** to fetch data from a server.

EXERCISE 14.01: REQUESTING DATA USING XMLHTTPREQUEST

In this exercise, we are going to use **XMLHttpRequest** to request data from **JSONPlaceholder**, http://jsonplaceholder.typicode.com/, a dummy URL to test the data. This allows us to use a fake online REST API to test requesting data with all the **HTTP** methods. We are particularly going to request the details of a post that has the **ID** number **1**.

The endpoint URL we are going to request is https://jsonplaceholder.typicode.com/posts/1.

1. Open the **DevTools** in your browser in the Console panel
2. Instantiate **XMLHttpRequest()** using the **new** keyword:

```
const xhr = new XMLHttpRequest();
```

The **new** keyword is used to instantiate an object.

3. As we want to receive the data in JSON format, we will add the **responseType** property and assign **json**:

```
const xhr = new XMLHttpRequest();
// assign json
xhr.responseType = 'json';
```

4. Initialize the request with the **`open()`** method. The first argument we want to set is **`GET HTTP`** method, and the second argument is the endpoint URL we are going to request data from:

```
const xhr = new XMLHttpRequest();
// assign json
xhr.responseType = 'json';
// initialize the request
xhr.open('GET', 'https://jsonplaceholder.typicode.com/posts/1');
```

5. Send the request to the server:

```
const xhr = new XMLHttpRequest();
// assign json
xhr.responseType = 'json';
// initialize the request
xhr.open('GET', 'https://jsonplaceholder.typicode.com/posts/1');
// send request to the server
xhr.send();
```

6. Let's receive the data with the load event. To get the data, we can access the response property from the result:

```
const xhr = new XMLHttpRequest();
// assign json
xhr.responseType = 'json';
// initialize the request
xhr.open('GET', 'https://jsonplaceholder.typicode.com/posts/1');
// send request to the server
xhr.send();
// receive data
xhr.addEventListener('load', () => {
  console.log(xhr.response);
});
```

7. Open up the **DevTools** in your browser and in the Console panel, enter the code. Let's execute the code in the browser. In a few moments, you will receive the data back from the server as follows:

```
> const xhr = new XMLHttpRequest();
  // assign json
  xhr.responseType = 'json';
  // initialize the request
  xhr.open('GET', 'https://jsonplaceholder.typicode.com/posts/1');
  // send request to the server
  xhr.send();
  // receive data
  xhr.addEventListener('load', () => {
    console.log(xhr.response);
  });
< undefined
                                                                    VM23:10
  {userId: 1, id: 1, title:
  "sunt aut facere repellat provident occaecati excepturi optio reprehenderit", body:
  "quia et suscipit↵suscipit recusandae consequuntur …strum rerum est autem sunt rem eveniet arch
  }
    body: "quia et suscipit↵suscipit recusandae consequuntur expedita et cum↵reprehenderit mol…
    id: 1
    title: "sunt aut facere repellat provident occaecati excepturi optio reprehenderit"
    userId: 1
  ▶ __proto__: Object
```

Figure 14.7: Receiving data using XMLHttpRequest on Chrome DevTools

As you can see, we received the **userId**, **id**, and **title** from the dummy REST API URL. Also, it is to be noted that sending simple data requires going through many steps with **XMLHttpRequest**. In the next section, we will discuss how to send the same data with the **Fetch API** and compare it with **XMLHttpRequest**.

FETCH API

The Fetch API, introduced in 2015, is a promise-based API (more on promise-based APIs in *Chapter 15*, *Promise API and async/await*.) that allows us to send requests to a server, quite similar to what **XMLHttpRequest** does, but there's a bit of a difference, as we'll see in a moment.

Let's write the same request we did with **XMLHttpRequest** in the last section with the **Fetch API** and then compare the two APIs to find out the benefits of the **Fetch API** compared to **XMLHttpRequest**.

To send data to a server, we need to call the **`fetch()`** function, which is available in the global window scope, so it is accessible from anywhere in your JavaScript code. In the **`fetch()`** function, we are going to specify the URL to send the data to and some optional information, such as the method, the body of data, and the headers, like so:

```
fetch('https://yourdomain.com/api/posts/new', {
  method: 'POST',
  body: JSON.stringify({
    title: 'this is a title',
    body: 'this is body',
    Header: {
      'Content-Type': 'application/json'
    }
  })
});
```

To get the response sent back from the server, we can make use of promises. We will discuss promises in more detail in *Chapter 15*, *Promise API and async/await*, but simply put, promises are used to tell you three states based on the asynchronous communication or actions with the server:

- **`Fulfilled`**: The action succeeded.
- **`Rejected`**: The action failed.
- **`Pending`**: The action is pending; it's not fulfilled or rejected yet.

When the action is either **`fulfilled`** with a value or **`rejected`** with an error, the promise's **`then()`** method is called. So, when we call the **`then()`** method for the first time, it will return a **`response`** object. The **`response`** object has several methods, and one of them is **`json()`**. In order to convert the response to JSON format, we need to call the **`json()`** method and call **`then()`** method again. In the second **`then()`** method, we will get the response as a JavaScript object. Let's add the **`then()`** method to the preceding **`fetch()`** function:

```
fetch('https://yourdomain.com/api/posts/new', {
  method: 'POST',
  body: JSON.stringify({
    title: 'this is a title',
    body: 'this is body',
    Header: {
      'Content-Type': 'application/json'
    }
```

```
  })
}).then(function(res) {
  return res.json();
}).then(function(data) {
  console.log(data);
});
```

Let's go through an exercise now to see the Fetch API in action.

EXERCISE 14.02: REQUESTING DATA USING THE FETCH API

In this exercise, we are going to make use of the Fetch API to request the same data as we did in *Exercise 14.01*, *Requesting Data using XMLHttpRequest* with **`JSONPlaceholder`** (http://jsonplaceholder.typicode.com/). To do so, let's go through the following steps:

1. Open up the **`DevTools`** in your browser and go to the Console panel.
2. Add the URL as the first parameter to request data from the endpoint using the **`Fetch API`**. As this is a simple **`GET`** request, we don't need to add any additional parameters:

```
fetch('https://jsonplaceholder.typicode.com/posts/1')
```

3. Call the **`then()`** function twice to receive the data to get the response and to get the data in the JavaScript object format. Put together, the code should look like this:

```
fetch('https://jsonplaceholder.typicode.com/posts/1')
  .then(function(res) {
    return res.json();
  }).then(function(data) {
    console.log(data);
  });
```

4. Run it and see the result received from the server. It is dependent on your internet speed, but when you receive data from the server it'll look like *Figure 14.5*; the data contains **`body`**, **`id`**, **`title`**, and **`userId`**:

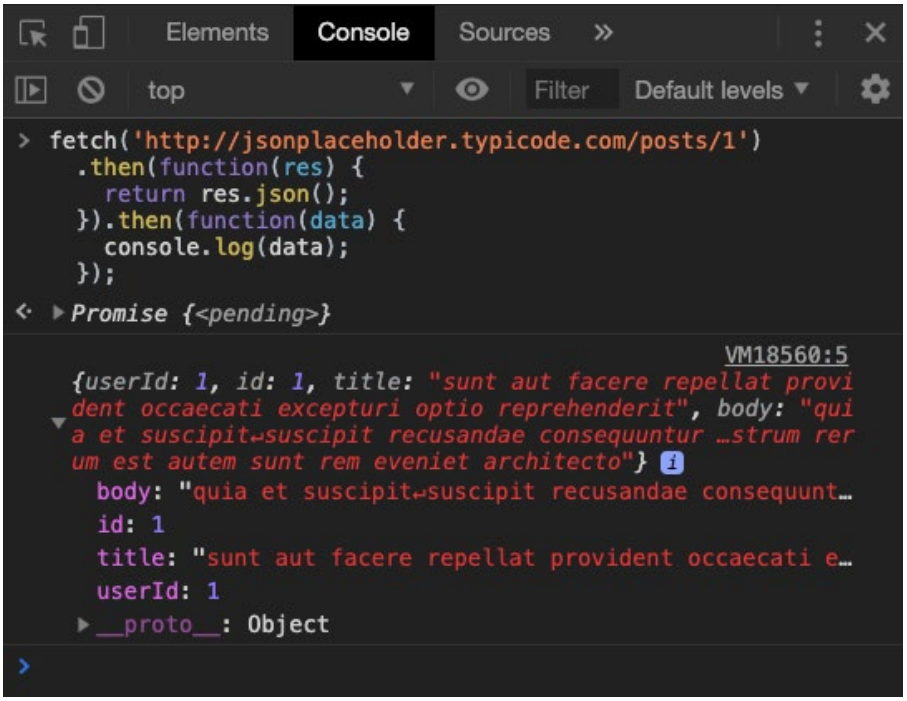

Figure 14.8: Receiving data using the Fetch API on Chrome DevTools

We have now seen how to request data from a server using **`XMLHttpRequest`** and the **`Fetch API`**. A notable difference between them is that the **`Fetch API`**, being promise-based, makes your code much more manageable and maintainable than **`XMLHttpRequest`**. Lastly, we are going to look at another method of performing HTTP requests called **`Axios`**.

AXIOS

`Axios` (https://packt.live/2WtWceQ) is a popular JavaScript library that is a promise-based HTTP client. It has many features, including transforming request and response data and canceling requests. We will use this API with React in order to fetch data from the server.

As **`Axios`** is a third-party library, we need to install it first. Once it's installed, we will be ready to make a request to a server. Let's look at a basic example using **`POST`**:

```
axios({
  method: 'POST', // it could be any other HTTP methods such as GET or
DELETE
  url: 'https://yourdomain.com/api/posts/new',
  headers: {'Content-Type': 'application/json'},
```

```
  data: {
    title: 'this is a title',
    body: 'this is body',
  }
});
```

In the preceding code, we are requesting the server to accept the data (title and body) and store it in the server. **`Axios`** also provides shortcut so that we can condense the preceding code like so:

```
axios.post('https://yourdomain.com/api/posts/new', {
  title: 'this is a title',
  body: 'this is body',
}, {
  headers: { 'Content-Type': 'application/json' }
});
```

Once the **`HTTP`** request is made, a response will be returned from the server containing information such as **`data`**, **`status`**, **`statusText`**, **`headers`**, **`config`**, and **`request`**. Among them, what we really want to get is the **`data`**, which contains the response provided by the server.

> **NOTE**
>
> To find out more details on each piece of information, please refer to the Axios documentation page, https://packt.live/3iPsQA4.

To receive the data, we can use the **`then()`** method:

```
axios.post('https://yourdomain.com/api/posts/new', {
  title: 'this is a title',
  body: 'this is body',
}, {
  headers: { 'Content-Type': 'application/json'}
})
.then(function(res) {
  console.log(res);
});
```

The **`then()`** method returns the data, specifically the **`title`** and the **`body`** of the response. Let's look at the following exercise to learn how to request data using Axios.

EXERCISE 14.03: REQUESTING DATA USING AXIOS

In this exercise, we are going to use Axios to request data from a server. The endpoint URL we are going to use is https://jsonplaceholder.typicode.com/posts/1, the same as in the Fetch API exercise. As we need to include the Axios library, for this exercise, we are going to use **`CodePen`**, which provides a development environment for the frontend work:

1. Go to https://codepen.io/ and create a new Pen.
2. Add the **`Axios`** library. To include the library, click the **`Settings`** button at the top and select the JavaScript menu in the pop-up window. In the JavaScript section, type **`axios`** in the search field. Click on **`Axios`** and the **`axios`** library will be automatically included. Click the **`Save & Close`** button:

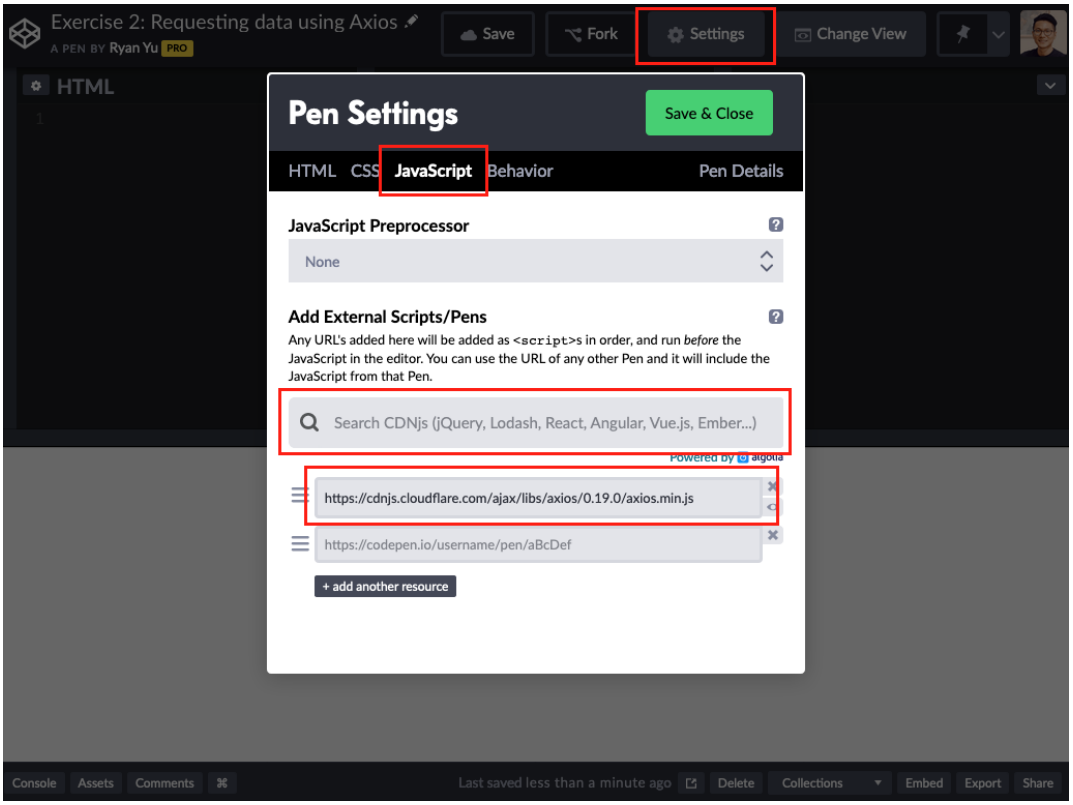

Figure 14.9: Adding the Axios library

`CodePen` allows us to add HTML, CSS, and JavaScript. For this exercise, we are only going to add JavaScript with **`Axios`**.

3. In a JavaScript editor, write the code to request the data from the endpoint using **`Axios`**:

```
axios.get('https://jsonplaceholder.typicode.com/posts/1')
  .then(function(res) {
  console.log(res);
});
```

4. Open up the **`DevTools`** in your browser and see if you are receiving data such as **`data`**, **`status`**, and **`headers`**:

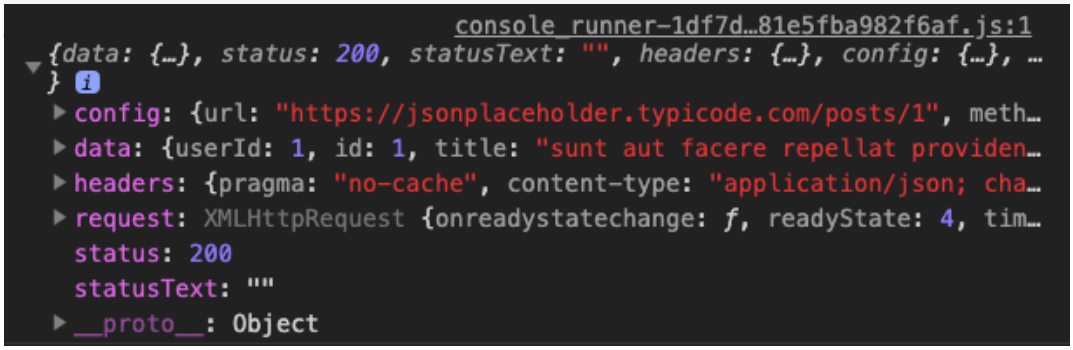

Figure 14.10: Receiving data using Axios

5. Call **`res.data`** as we want to get the data rather than any other information, and it will give us the data directly containing **`body`**, **`id`**, **`title`**, and **`userId`**:

```
axios.get('https://jsonplaceholder.typicode.com/posts/1')
  .then(function(res) {
  console.log(res.data);
});
```

The output is as follows:

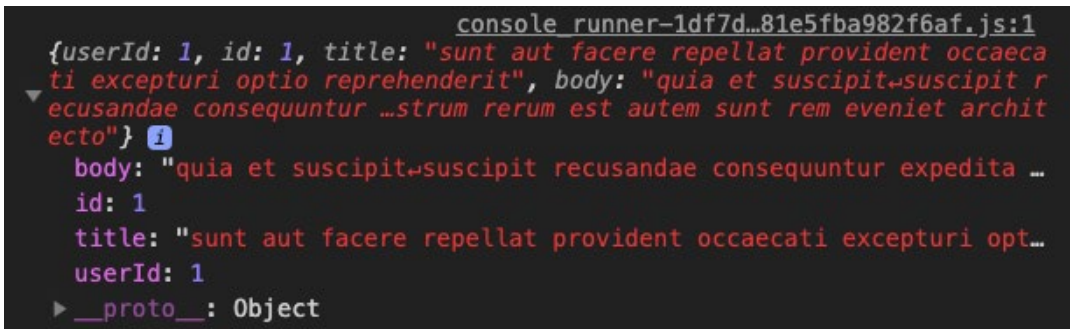

Figure 14.11: Receiving only data

Now that we have discussed three different ways to request data from a server, **`XMLHttpRequest`**, the **`Fetch API`**, and **`Axios`**, let's compare them. We will see that **`Axios`** is preferable to the other APIs.

COMPARISON OF XMLHTTPREQUEST, THE FETCH API, AND AXIOS

In the previous sections, we have learned how to fetch data from a server in several ways, **`XMLHttpRequest`**, the **`Fetch API`**, and **`Axios`**. In this section, we are going to compare the differences between the three main methods based on the following factors:

- Ease of syntax
- Supporting promises
- Browser support

We will discuss which method we want to use:

- **Ease of syntax**: When we compare the syntax in the previous section, the **`Fetch API`** and **`Axios`** provide much easier syntax than **`XMLHttpRequest`**. The syntax of **`Fetch`** and **`Axios`** is cleaner and simpler to understand and also ensures fewer lines of boilerplate code.
- **Supporting promises**: The **`Fetch API`** and **`Axios`** provide promises. We are going to talk about promises in more depth in *Chapter 15*, *Promise API and async/await*, but briefly, they allow you to chain functions, which helps us avoid callback hell.

  Callback hell is an anti-pattern that consists of multiple nested callbacks. With **`XMLHttpRequest`**, if we wanted to request data based on the previous request, it would end up looking like this:

```
const postXHR = new XMLHttpRequest();

postXHR.open('GET', 'https://yourdomain.com/api/post/1');
postXHR.send();

postXHR.onload = function() {
  const postInfo = this.response;
  const authorXHR = new XMLHttpRequest();
```

```
  authorXHR.open('GET', `https://yourdomain.com/api/autor/${postInfo.
authorId}`);
  authorXHR.send();

  authorXHR.onload = function() {
const authorInfo = this.response;
const countryXHR = new XMLHttpRequest();

countryXHR.open('GET', `https://yourdomain.com/api/
country/${authorInfo.countryCode}`);
countryXHR.send();

countryXHR.onload = function() {
  const countryInfo = this.response;

  console.log('Country:', countryInfo.name);
}
  }
}
```

As you can see, it is quite hard to read, and if any errors occur, it is hard to debug as we cannot pinpoint which callback the error is coming from.

With the **`Fetch`** API and **`Axios`** being promise-based, we can avoid callback hell. The preceding code can be updated as follows with promises:

```
fetch('https://yourdomain.com/api/post/1')
  .then(res => res.json())
  .then(getPostInfo)
  .catch(err => console.log(err));

function getPostInfo(res) {
  fetch(`https://yourdomain.com/api/autor/${res}`)
  .then(res => res.json())
  .then(getAuthorInfo)
  .catch(err => console.log(err));
}

function getAuthorInfo(res) {
  fetch(`https://yourdomain.com/api/country/${res}`)
  .then(res => res.json())
  .then(getCountryInfo)
  .catch(err => console.log(err));
```

```
}

function getCountryInfo(res) {
  const countryInfo = this.response;

  console.log('Country:', countryInfo.name);
}
```

By chaining functions with promises, the syntax is more intuitive, and it is much easier to debug.

- **Browser support**: Both **`XMLHttpRequest`** and the **`Fetch API`** have browser-native features. That means we do not need to install any third-party libraries to use them. As you can in the following figure (https://caniuse.com/#feat=xhr2), XHR was supported by most browsers in 2019:

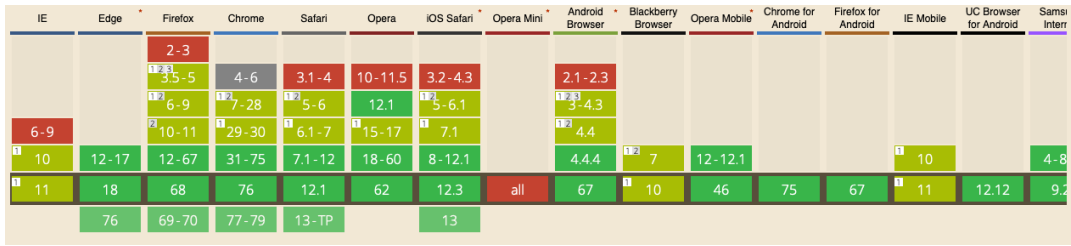

Figure 14.12: Browser support for XMLHttpRequest

The **`Fetch API`** is also supported by most browsers, except some old browsers such as IE, Blackberry, and Opera Mini.

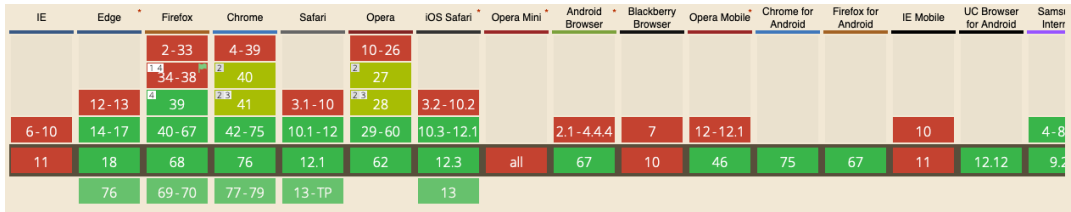

Figure 14.13: Browser support for the Fetch API

Unlike **`XMLHttpRequest`** and **`Fetch`**, **`Axios`** is a JavaScript library. Therefore, to use **`Axios`**, we need to install it, as mentioned earlier. The browser support **`Axios`** provides is quite good as it is supported by all the major browsers, including IE 11.

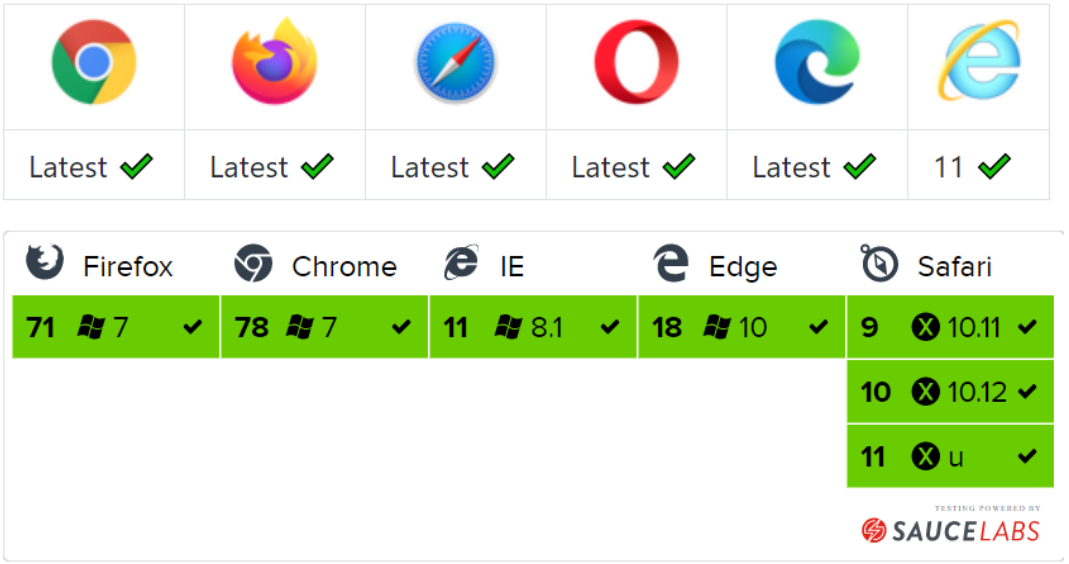

Figure 14.14: Browser support for Axios

We have compared **`XMLHttpRequest`**, **`Fetch`**, and **`Axios`**, and of the three, **`Axios`** provides the most benefits when carrying out HTTP requests.

Apart from the three main features that we have discussed, other helpful features include:

- Better response handling
- Better error handling
- Allowing many requests at once
- Supporting canceling requests
- Client-side support for protecting malicious exploitation of a website

Among those benefits **`Axios`** has to offer, let's further compare **`Axios`** and **`Fetch`** based on the **`response handling`** and **`error handling`** functionalities.

AXIOS VERSUS THE FETCH API

When we compare the **`Fetch API`** and **`Axios`**, they look quite alike. However, let's discuss the features mentioned previously to see how **Axios does better than the Fetch API**.

BETTER RESPONSE HANDLING

In *Exercise 14.01*, *Requesting Data Using XMLHttpRequest*, when we received data from the server with **`Fetch`**, we had to go through two steps. The first step involved passing the response of the **`HTTP`** request to the **`json()`** method, and the second step allowed us to receive the **`data`** object:

```
fetch('https://jsonplaceholder.typicode.com/posts/1')
  .then(function(res) {
    return res.json();
  }).then(function(data) {
    console.log(data);
  });
```

However, as we practiced in *Exercise 14.02*, *Requesting Data Using the Fetch API*, Axios automatically converts the response to **`JSON`** data, which is simpler to process:

```
axios.get('https://jsonplaceholder.typicode.com/posts/1')
  .then(function(res) {
console.log(res);
});
```

Thus, it can be concluded that **`Axios`** does response handling more effectively than the Fetch API.

BETTER ERROR HANDLING

When you request data from a server, it is important to catch any errors so that we know where we went wrong. To catch an error with the Fetch API, we use the **`catch()`** method. As an example, let's request data from an incorrect URL and add **`console.log`** both in the second **`then()`** method and the **`catch()`** method:

```
fetch('https://jsonplaceholder.typicode.com/this-is-wrong-url')
  .then(function(res) {
    return res.json();
  }).then(function(data) {
```

```
    console.log('Received data:', data);
  }).catch(function(err) {
    console.log('Error:', err);
  });
```

Now, can you guess what message will be printed? As the URL does not exist, the server will return a **`404`** error. You may have guessed that since this is an error, it must be caught and the error message will get displayed. Let's see what we get:

Figure 14.15: Catching error with the Fetch API

As you can see, even if the server returned a **404** error, we have received the data with an empty object.

Now, let's see how **`Axios`** handles this error:

```
axios.get('https://jsonplaceholder.typicode.com/this-is-wrong-url')
  .then(function(res) {
  console.log('Received data:', data);
  }).catch(function(err) {
     console.log('Error:', err);
  });
```

Again, guess if we will receive the **`404`**error message with **`Axios`** as shown in the following screenshot:

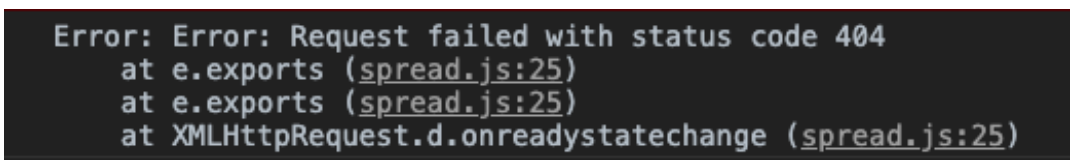

Figure 14.16: Catching an error with Axios

As you can see from the preceding figure, the error message with the status code is displayed. Along with that, the line numbers where the error occurred are also shown, which is exactly what we want.

Requesting data from an incorrect endpoint URL is a mistake that developers often make, and not returning the proper error status and message makes code harder to debug. Therefore, we recommend using **`Axios`** along with React to request data from a server.

We now have a better idea of how to request data from a server and we have tested some APIs. However, to test the APIs, we had to write some JavaScript and execute the JavaScript in the browser **`DevTools`** or **`CodePen`**, where we had to include third-party libraries such as **`Axios`**. This is a bit of a hassle when we only quickly want to verify the response sent from the server. In this section, we are going to look at a tool called Postman, an interactive and automatic standalone tool for testing APIs, and learn how to easily test and manage RESTful APIs.

TESTING APIS WITH POSTMAN

Postman has a simple user interface that allows us to easily test APIs by sending HTTP requests. For example, checking whether we are receiving the correct data from the endpoint with the **`GET`** method takes only three steps without writing any JavaScript code:

- Adding the endpoint
- Choosing the **`HTTP`** method
- Clicking the **`Send`** button

To install Postman, go to the download page, https://www.getpostman.com/downloads/, and follow the instructions.

> **NOTE**
>
> Postman is available on Mac, Windows, and Linux. At the time of writing this book, the stable version of Postman is 7.5.0.

Once installed, launch Postman and you will see the interface shown here:

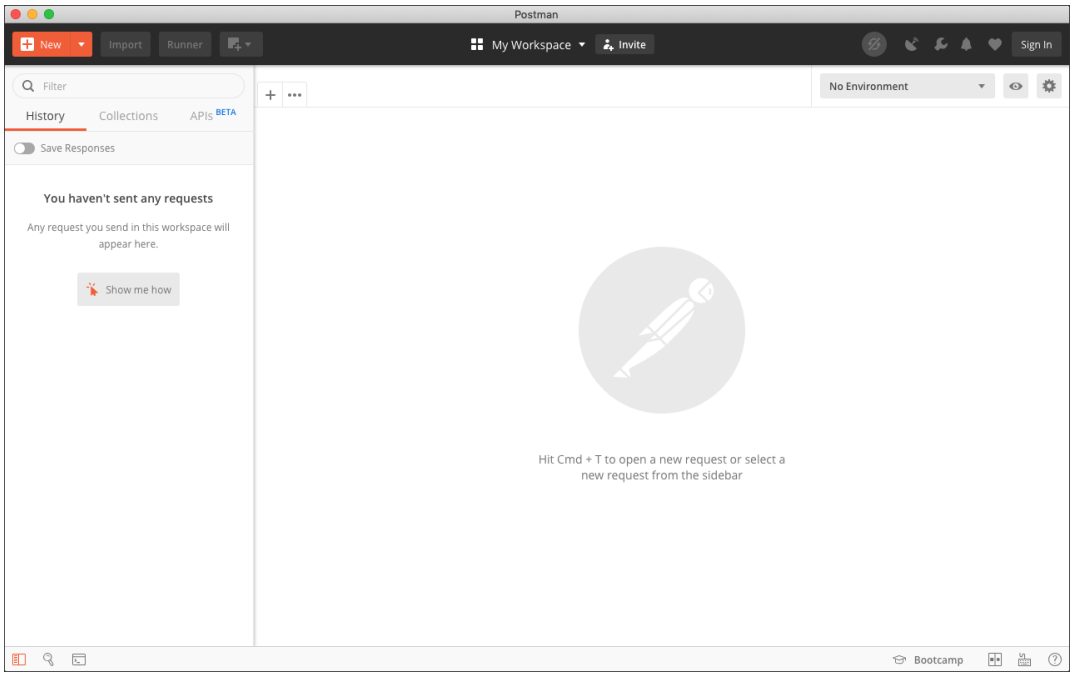

Figure 14.17: User interface of Postman

Now let's make the same requests we have done in the previous sections using Postman. In the following exercise, we are going to practice all of the HTTP methods we have learned previously in the **Five Common HTTP Methods** section, **`GET`**, **`POST`**, **`PUT`**, **`PATCH`**, and **`DELETE`**. For the APIs, we are going to follow the routes in **`JSONPlaceholder`**, http://jsonplaceholder.typicode.com/.

EXERCISE 14.04: GET AND POST REQUESTS WITH POSTMAN

In this exercise, firstly, we are going to make a request to receive data using the **GET** method, and then we'll use the POST request with Postman. Let's see how:

1. After you install **Postman**, click the **New** button in the top-left corner of the interface and click **Request** in the pop-up window.

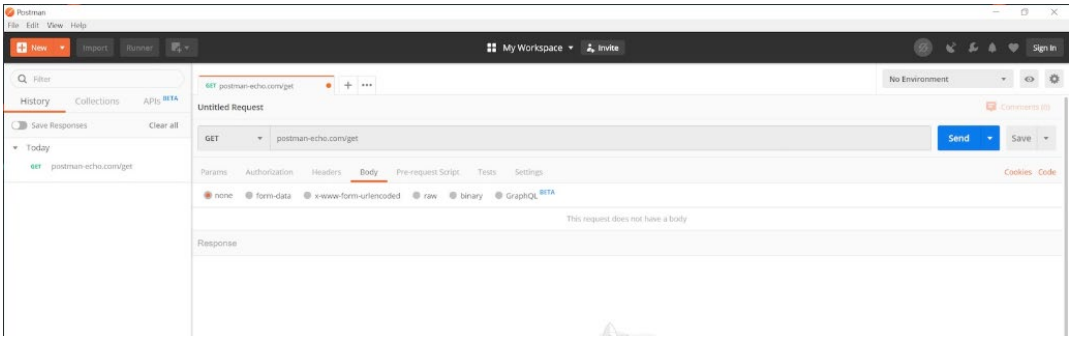

Figure 14.18: Postman UI

2. Give **GET method** as the **Request name**.

3. Choose a **Collection**. A collection in Postman is like a folder where you can organize your requests and easily share them with others. If you have no collections yet, click the **Create Collection** link and make one. Then click the Save button.

4. Add our first request. The **GET** method should already be selected, but if not, select **GET** in the drop-down menu.

5. In the input field with the placeholder saying **`Enter request URL`**, add http://jsonplaceholder.typicode.com/posts and click the **`Send`** button. You should successfully receive the **`200 OK`** status with the data from the server as shown here:

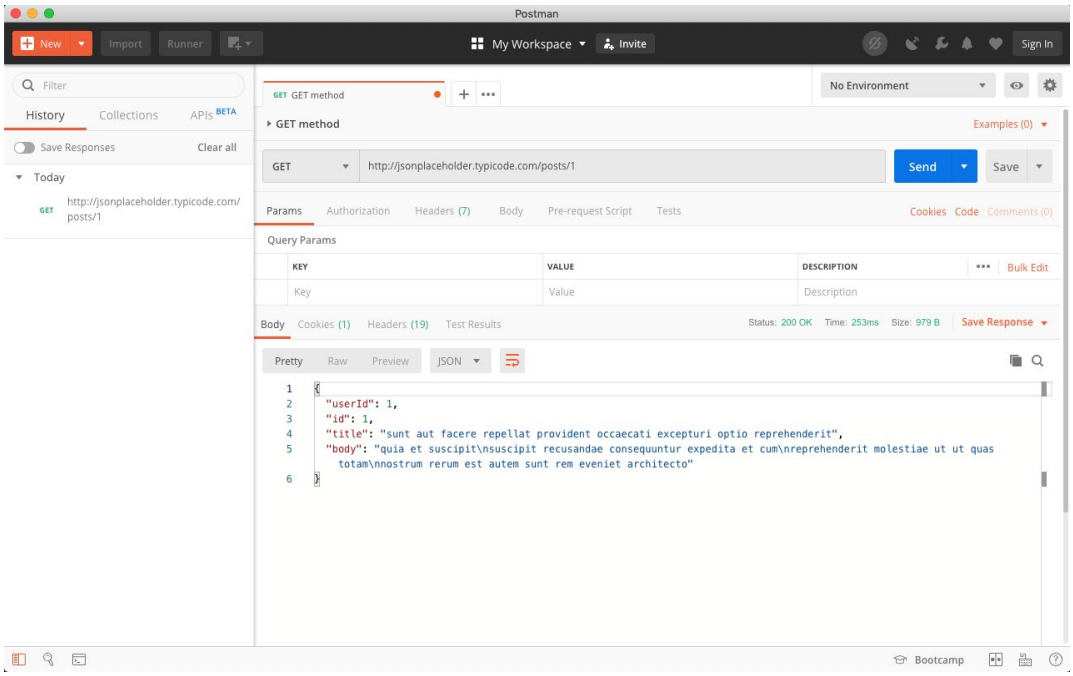

Figure 14.19: Received 200 OK status and data from the server

If you specify the wrong endpoint, it will return with a **`404 Not Found`** status and an empty object:

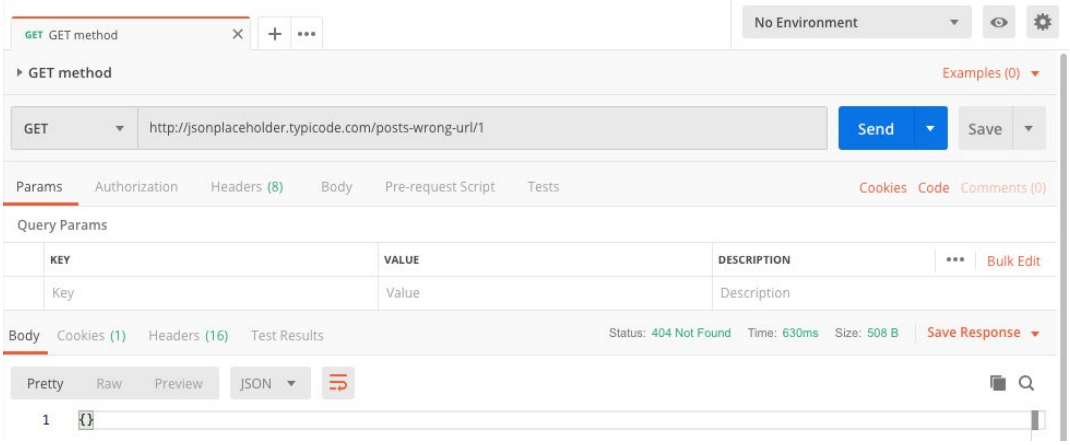

Figure 14.20: 404 Not Found status

We will now see the **`POST`** request with Postman.

6. Click the New button in the top-left corner again and click Request in the pop-up window.

7. Add **`POST method`** as a Request name and choose a collection.

   For the HTTP method, choose **`POST`** and add the endpoint, https://jsonplaceholder.typicode.com/posts. Let's say we want to add the title and body as **`user ID: 1`** for the new post. In that case, we need to include the data as shown in the following code in the body of the request. So, click the **`Body`** tab, choose **`raw`**, select **`JSON (application/json)`**, and add the following data in the **`textarea`** field:

```
{
  "title": "this is the title of the new post",
  "body": "this is the body of the new post",
  "userId": 1
}
```

The output is as follows:

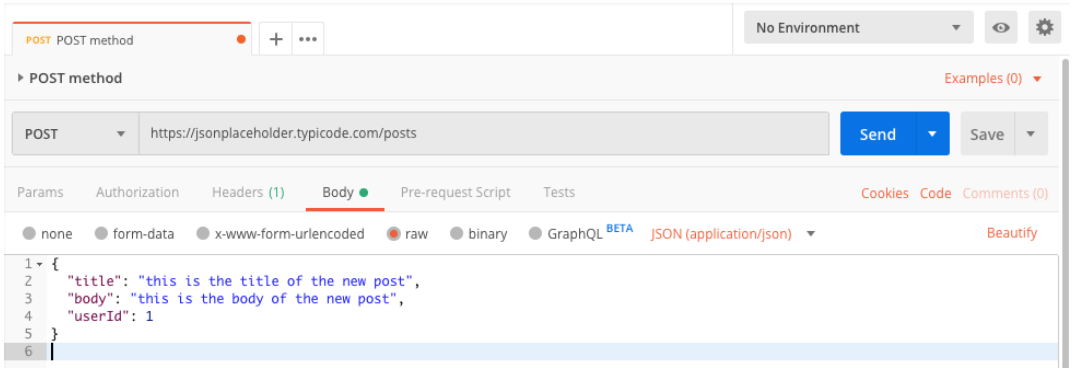

Figure 14.21: Adding the data

8. We also need to tell the server that the format of the data returned should be JSON.

   So, including the **`Content-Type`** with **`application/json`** by selecting JSON (**`application/json`**) from step 7 will automatically add the content-type to the header. You can see this in the **`Headers`** tab:

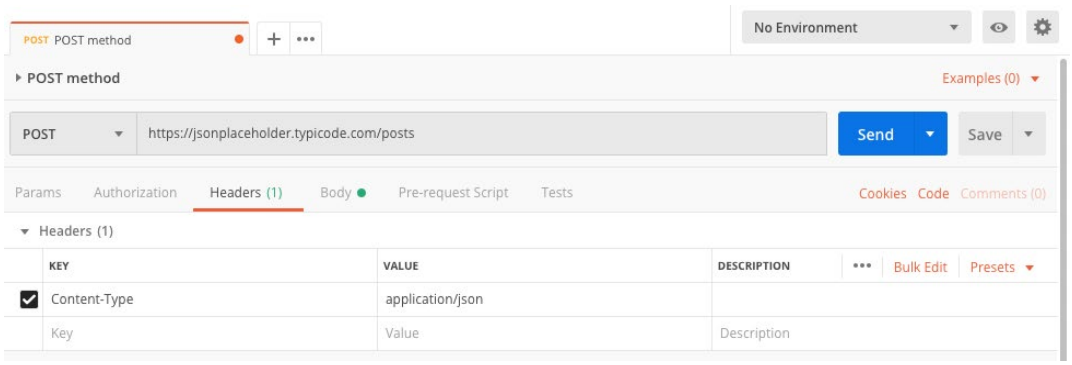

Figure 14.22: Content-Type added

9. Click the **Send** button, and you will receive a **201 Created** status with the newly created data, along with the ID.

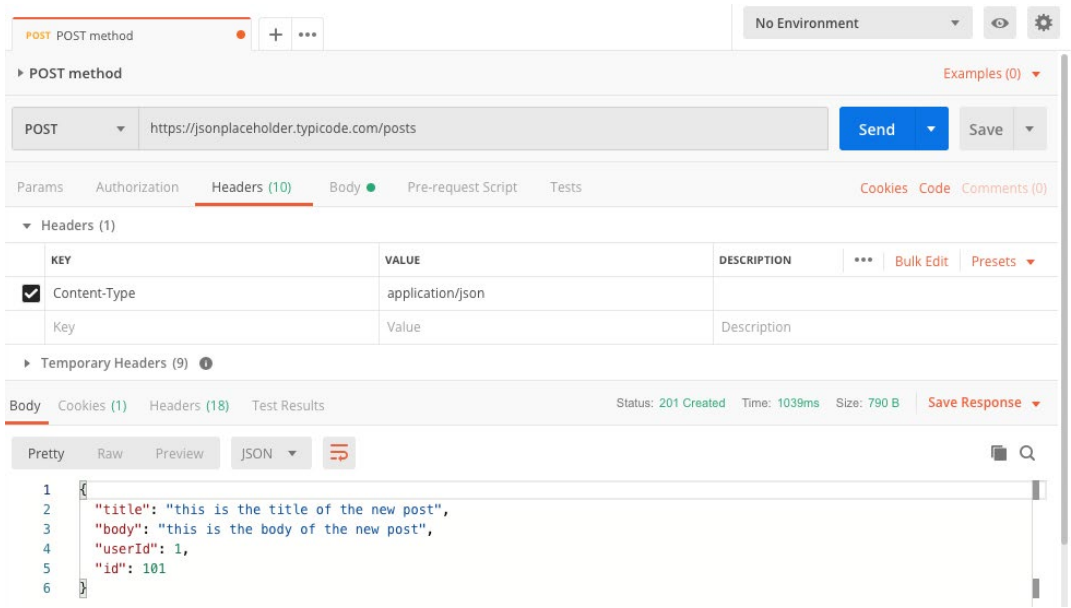

Figure 14.23: Received 201 Created status and data

We will now go ahead and test the other HTTP methods with Postman.

EXERCISE 14.05: PUT, PATCH, AND DELETE REQUESTS WITH POSTMAN

In this exercise, we will use the **PUT**, **PATCH**, and **DELETE** requests and test them with Postman:

1. Click the **New** button in the top-left corner of the interface and click **Request** in the pop-up window.
2. Select the **PUT** method. We are going to make changes to **post ID: 1**, so add the endpoint, https://jsonplaceholder.typicode.com/posts/1.
3. Click **body** and add the new data that we want to update.
4. Select raw and choose JSON (**application/json**), which will automatically add the content-type for us. In the **textarea**, add the following data:

```
{
  "title": "this is updated title"
}
```

5. Click the **Send** button, and you will receive a **200 OK** status with the updated data. However, you will notice that the returned data only contains two details, title and id, as shown in the following snippet. This is because, as mentioned in the *PUT versus PATCH* section, the **PUT** method will override the entire data matching with post id **1**. To partially update the data, we need to use **PATCH**.

```
{
  "title": "this is updated title",
  "id": 1
}
```

6. Create a new request and choose the **PATCH** method this time with the https://jsonplaceholder.typicode.com/posts/2 endpoint and add the same details as before.
7. Click the Send button, and you should receive a **200 OK** status with the entire data, but only the title updated.
8. Create a new request and select the **DELETE** method with the https://jsonplaceholder.typicode.com/posts/1 endpoint.
9. As we are not updating any details, we don't need to include any headers or body. Click the **Send** button.
10. You will receive a **200 OK** status with an empty object, which indicates that the data has successfully been deleted.

    Let's look at the following output:

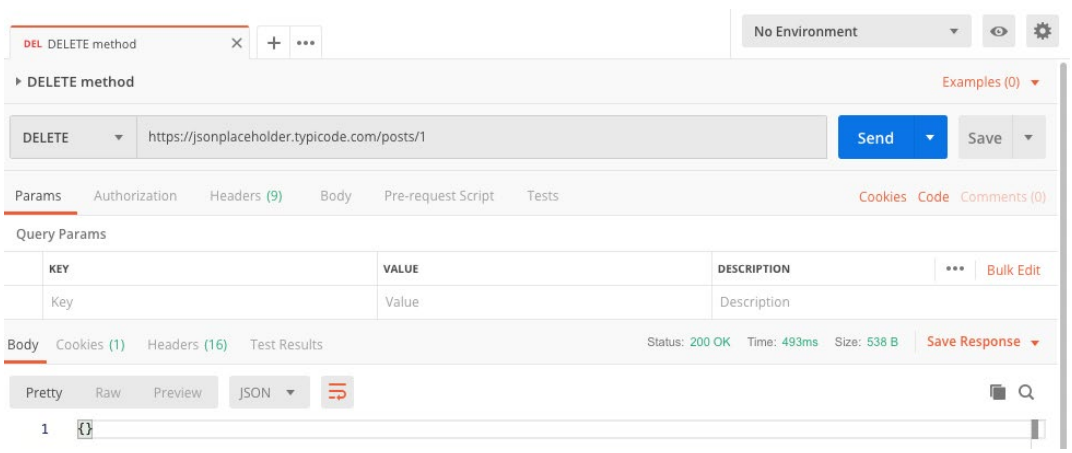

Figure 14.24: Received 200 OK when the data was successfully deleted

> **NOTE**
>
> The **`POST`**, **`PUT`**, **`PATCH`**, and **`DELETE`** requests did not actually update the data from the **`JSONPlaceholder`** server; however, **`JSONPlaceholder`** returned the correct status as if the data had been updated.

Now that we know the basics of fetching data and the various APIs used to do it in React, let's see how to create API requests in React.

MAKING API REQUESTS IN REACT

In the previous sections, we learned the different ways of requesting data. Also, we discussed the benefits of using **`Axios`** over other methods, such as **`XMLHttpRequest`** and Fetch API.

In this section, based on what we have learned so far, we will make API requests with React. For the exercises, we are going to use NASA Open APIs (https://api.nasa.gov/) to search NASA images. In this chapter, we are only going to focus on how to receive data back from the NASA server using React.

REACT BOILERPLATE AND AXIOS

To get started, let's install create-react-app first and then use **`Axios`** to make API requests. Remember that **it is not React's job to make API requests to the server**. React is for helping us to display content on the screen and it is **`Axios`**, a layer underneath, that makes requests and receives data back from the server.

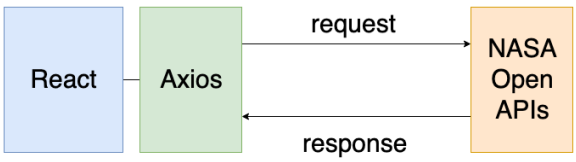

Figure 14.25: Requesting data with Axios in React

We will go through an exercise to integrate Axios with React boilerplate.

EXERCISE 14.06: INSTALLING REACT BOILERPLATE AND AXIOS

In this exercise, we will integrate **`Axios`** with the React boilerplate. We will install the NASA API and we will request data from this API using React. Let's see how:

1. Open the terminal and type in the command:

```
npx create-react-app search-nasa-images
```

   This should install all the necessary dependencies automatically for you.

2. Type in the **`cd search-nasa-images`** command to navigate to the **`search-nasa-images`** folder where we had just installed create-react-app.
3. Type in the **`yarn add axios`** command to install **`Axios`** onto the boilerplate.
4. Open the boilerplate in your preferred text editor.
5. Select all the files in the **`src`** folder and remove them all. We are going to create our own files.
6. Create a new file called **`index.js`** in the **`src`** folder and add the following code:

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.render(<App />, document.querySelector('#root'));
```

7. In the same directory, create another file called **`App.js`** and add the following code:

```
import React from 'react';
import './App.css';
const App = () => {
  return (
    <div className="o-container">
    <h1>Search NASA images</h1>
    </div>
  );
};
export default App;
```

8. Create another file called **`App.css`** in **`src`** folder and add the following CSS code:

```
.o-container {
  margin: 50px auto;
  max-width: 700px;
}

h1 {
  text-align: center;
}
```

9. Run the app by typing **`yarn start`** in the Terminal. In your browser, go to http://localhost:3000/ and you should see the heading **`Search NASA images`** on the screen:

Figure 14.26: The output of the boilerplate

We have now prepared the boilerplate and are ready to fetch data from NASA. Before we fetch data, let's take a look at what the endpoint looks like.

TESTING THE NASA API WITH POSTMAN

NASA provides a lot of APIs, and what we are going to use is in the section is the NASA Image and Video Library API. According to the documentation page, "*the images API contains 4 endpoints*", the four URLs where clients of a specific service can gain access. That means we are going to use the **`GET`** method and the base URL will be https://images-api.nasa.gov followed by the first endpoint from the table (as you'll be able to see in the following exercise), **`/search?q={q}`**. For example, to search for **`europe`**, the endpoint we are going to request from will be https://images-api.nasa.gov/search?q=europa.

Let's check whether the endpoint properly returns any data through the following exercise.

EXERCISE 14.07: TESTING THE ENDPOINT WITH POSTMAN

In this exercise, we will test the NASA API using the Postman tool. Let's see how we do that:

1. Open up Postman and create a new request called **`NASA image search`**.
2. Leave the **`GET`** method as it is and add the URL https://images-api.nasa.gov/search.

   Make sure we do not add the **`q`** parameter at the end of the URL.
3. In the Params tab, add **`q`** as the **`Key`** and **`europa`** as the **`Value`**.
4. Click the **`Send`** button, and it will return the response with **`200 OK`** status and the data.

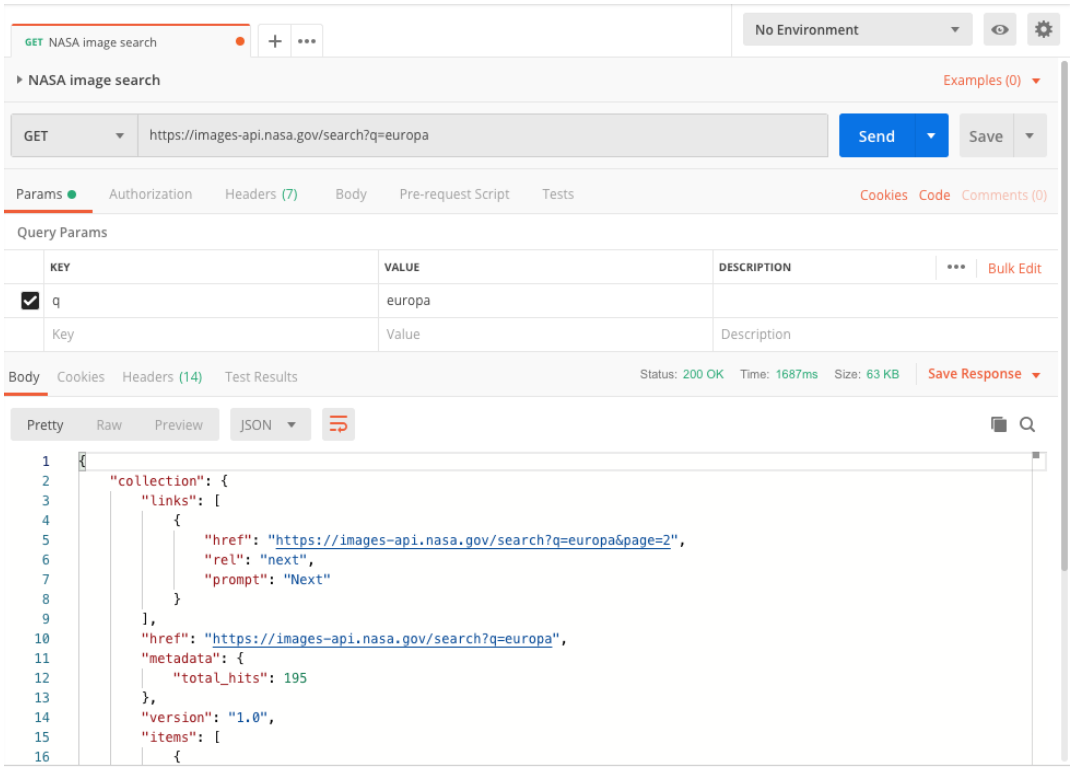

Figure 14.27: Requested and received data

5. If you scroll down the result, you will see the **`items`** array, and each array contains **`href`**, **`links`**, and **`data`**. In the **`links`**, you will see **`href`**, and clicking on the link will show you the image. That means in React, we are going to access the image by following **`collection.items[#].links.href`**.

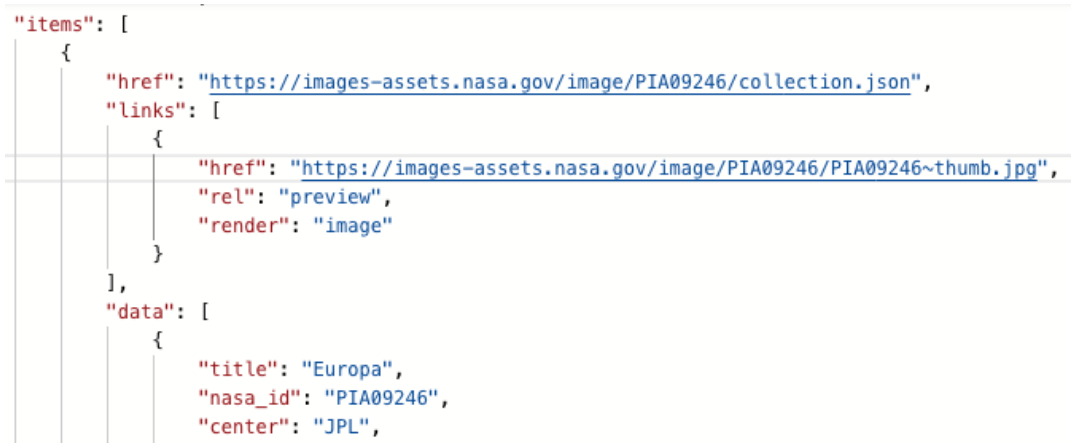

Figure 14.28: Finding the image

We have successfully tested and confirmed that the endpoint for searching images in the NASA images API app is working fine. It's time to add it to our React app.

FETCHING DATA WITH REACT

In this section, we are going to use **`Axios`** inside React and fetch data from the endpoint we have tested with Postman.

EXERCISE 14.08: CREATING A CONTROLLED COMPONENT TO FETCH DATA

In this exercise, we will first need to update a functional component to a class-based component as we will need to initialize the state of the component. Then we will create a controlled search form component. We will then create an **`Axios`** request in order to fetch images using the NASA API. Perform the following steps:

1. Update the current functional **`App`** (used in the previous exercise) component to the class-based component. The code should now look like this:

```
import React, { Component } from 'react';

import './App.css';

class App extends Component {
```

```
  render() {
    return (
  <div className="o-container">
      <h1>Search NASA images</h1>
  </div>
    );
  }
};

export default App;
```

2. Let's add a form with a class name, **`c-search`**, under **`<h1>`** in the **`render`** function. The **`form`** contains two children elements, **`<input>`** and **`<button>`**. With the **`form`** element, the code should look like this:

```
import React, { Component } from 'react';

import './App.css';

class App extends Component {
  render() {
    return (
  <div className="o-container">
      <h1>Search NASA images</h1>

      <form className="c-search">
      <input
        type="search"
        name="image-search"
        className="c-search__input" />

      <button>Search images</button>
      </form>
  </div>
    );
  }
};

export default App;
```

3. In **`App.css`**, add the following CSS code:

```
.c-search {
  display: flex;
  margin: 0 auto;
  width: 50%;
}

.c-search__input {
  display: block;
  flex-grow: 1;
  margin-right: 10px;
}
```

4. When you refresh the page on your browser, you should see the screen with the form underneath the heading as shown in the following screenshot:

Figure 14.29: Heading and form

5. To create the controlled component, we are going to create a state and update the state when we type text in the search input field. Initialize the term state in the **`constructor()`** method. Add an **`onChange`** event in the search field with a function that will update the term state with the value of the search field. Finally, add the value from the term state. The updated code should look like this:

`App.js`

```
6 class App extends Component {
7  constructor(props) {
8  super(props);

10 this.state = {
11    term: ''
12  };
13  }
…
30 return (
31    <div className="o-container">
32        <h1>Search NASA images</h1>

34        <form className="c-search">
35            <input
36           type="search"
37           name="image-search"
38           className="c-search__input"
39            value={this.state.term}
40            onChange={e => this.setState({ term: e.target.value })} />
```

The complete code can be found here: https://packt.live/2YXyLMK

6. Add the **`console.log(this.state.term)`** right above the **`return()`** method to see if we are getting the updated value from the search field when we type in the search field. Open the **`DevTools`** and make sure you are getting the updated term state in the Console. When you confirmed getting the updated term state, remove the **`console.log`**.

   We are now getting the search term and storing it in the term state. Next, we will get the value from the term state when we submit the form and make a request with the search term stored in the state.

7. It's time to make a request with the search term. To make a request, we are going to use **`Axios`**. Let's import **`Axios`**:

```
import axios from 'axios';
```

8. As we are going to request data when submitting the form, we are going to add an **`onSubmit`** event to the **`<form>`** element and reference to a function called **`onSearchSubmit`**. Make sure to bind this keyword to the function call:

```
<form className="c-search" onSubmit={this.onSearchSubmit.bind(this)}>
```

9. Create the **`onSearchSubmit`** function under the **`constructor()`** method and receive an event as an argument. With the event, we are going to call the **`preventDefault`**() method. The **`preventDefault`**() method will prevent the browser from refreshing the page as it normally would do when submitting the form. It is important to prevent the default action, otherwise our React app will lose all the data stored in the state and will restart from the initial state:

```
onSearchSubmit = (event) => {
  event.preventDefault();
};
```

10. Under the **`preventDefault()`** method, we are going to request data using Axios. As we are going to use the **`GET`** method, we will add the Axios **`get`** method. For the first argument, we will add the base URL for the endpoint, which is https://images-api.nasa.gov/search, and for the second argument, we will add params as follows:

```
axios.get('https://images-api.nasa.gov/search', {
  params: { q: this.state.term }
})
```

We are now going to receive the data in **`Promise`** with the **`then()`** method.

11. Inside the **`then()`** method, print the response:

```
    .then(res => {
      console.log(res);
    });
```

The entire code should look like this:

`App.js`

```
1  import React, { Component } from 'react';
2  import axios from 'axios';
3
4  import './App.css';
5
6  class App extends Component {
7   constructor(props) {
8    super(props);
9
10  this.state = {
11    term: ''
12  };
13  }
14
15  onSearchSubmit = (event) => {
16  event.preventDefault();
17
18  axios
19  .get('https://images-api.nasa.gov/search', {
20    params: { q: this.state.term }
21  })
22  .then(res => {
23    console.log(res);
24  });
25  };
```

`The complete code can be found here:` https://packt.live/2zB5DQQ

12. Add **`europa`** in the search field and click the **`Search images`** button. Open the **`DevTools`**, and in the Console panel, in a few moments, you will receive data from the server:

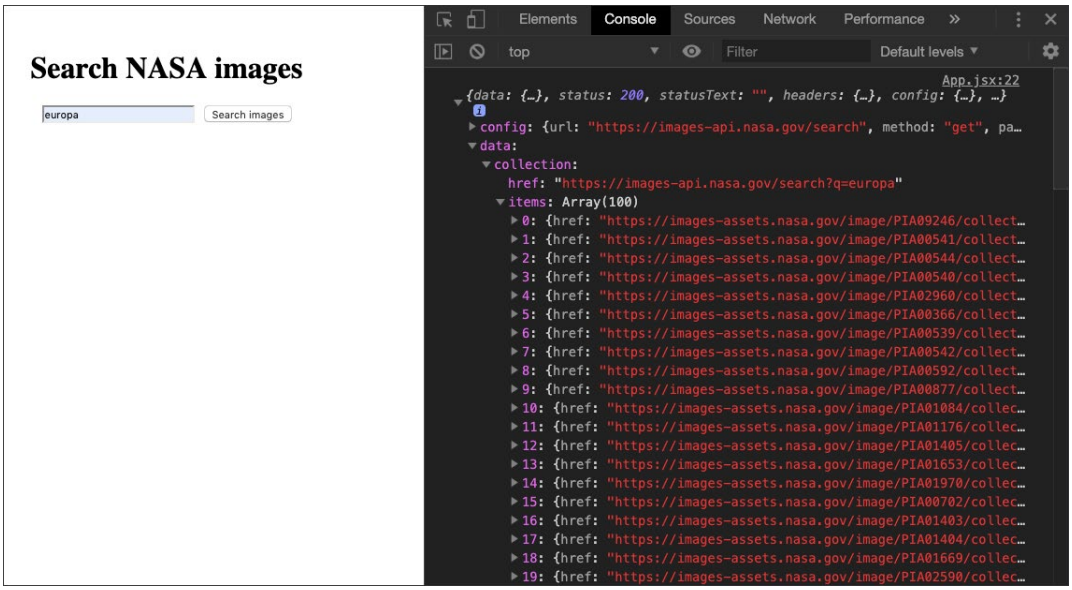

Figure 14.30 Receiving data from the server

As we can see, on the right-hand side of the screenshot the list of URLs of the images is displayed.

With all this knowledge, let's get started with the activity.

ACTIVITY 14.01: BUILDING AN APP TO REQUEST DATA FROM UNSPLASH

The aim of this activity is to build an app that makes API requests to Unsplash and get two kinds of data, one for a random photo URL and the other for statistics such as the total downloads, views, and likes of the photo.

Unsplash is a place for sharing high-definition photos under the Unsplash license, https://unsplash.com/license. Unsplash also provides free APIs allowing us to request data from their large, open collection of high-quality photos.

The optional steps are showing you how to display the random photo covering the entire page and its statistics on the screen. The complete app with the optional steps should look similar to this:

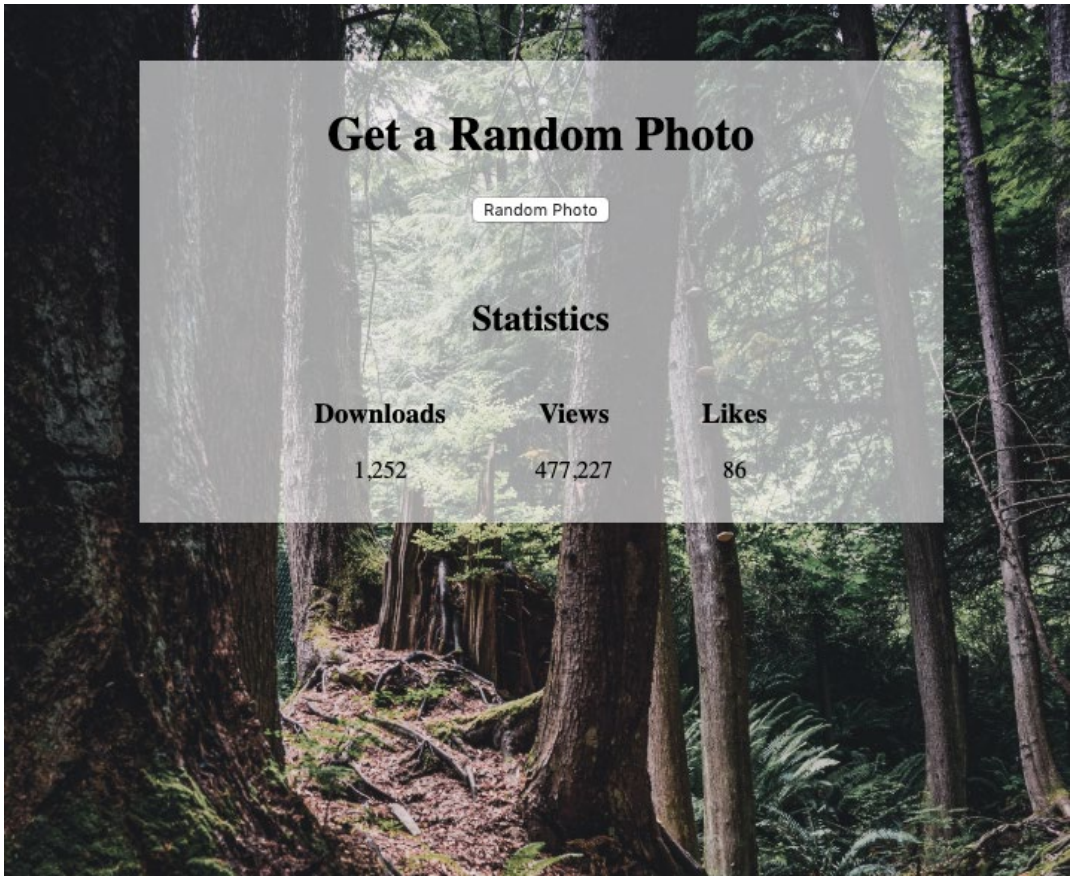

Figure 14.31: The output of the activity with the optional steps

The following steps will help you complete the activity:

1. To make API requests, you need to create a developer account. Register at https://unsplash.com/join and create a new app to get the **`Access Key`**.
2. To get the random photo, check out this documentation page to get the endpoint: https://unsplash.com/documentation#get-a-random-photo
3. To get the photo's statistics, check out this documentation page to see how you can create the endpoint: https://unsplash.com/documentation#get-a-photos-statistics

4. To authenticate requests, check out this documentation page to see what details you need to pass: https://unsplash.com/documentation#public-actions
5. Lastly, you can only make **50** requests per hour in demo mode. Please keep that in mind.

> **NOTE**
>
> The solution of this activity can be found on page 715

SUMMARY

This chapter covered how to fetch data by making API requests in three different ways, **XMLHttpRequest**, the **Fetch API**, and **Axios**. Throughout this chapter, we practiced making requests to **JSONPlaceholder** and learned several different **HTTP** methods. Also, we learned how to get the images from **NASA Open API**s in React. At the end of the chapter, the activity showed you how to make requests to Unsplash to display a random photo and statistics.

We compared three different ways, **XMLHttpRequest**, **Fetch API**, and **Axios**, to request data and discussed why we should use **Axios** over the other two ways.

We learned what a **RESTful API** is and what the five common **HTTP** methods are. Furthermore, we talked about the differences between the Accept and Content-Type header, followed by how to test APIs with **Postman**.

After that, we learned how to make API requests in React by making requests to NASA Open APIs and searched NASA images by submitting a search term.

With this essential understanding of how to create a search field and make a request to the server, in the activity, we practiced searching images from Unsplash. To request data, we also learned how to authenticate by sending the right details through headers. Finally, we received data with promises using the **then()** method. In the next chapter, we are going to discuss the differences between promises and **async**/**await** and how to use them in the appropriate places. We will take a deep dive into the Promise API and **async**/**await** methods.

15

PROMISE API AND ASYNC/AWAIT

OVERVIEW

This chapter takes a deep dive into the Promise API and the methods used in order to make a network request. You will be introduced to another method, **`async`**/**`await`**, and eventually you will be able to explain why async/await is better than the Promise API. By the end of this chapter, you will have a good understanding of essential techniques in React and the modern way to fetch data from the server.

INTRODUCTION

JavaScript is a synchronous language and React, being a JavaScript framework, is synchronous too. Being synchronous means only one line of code can be executed at a time and the next line of code cannot start until the current code has finished executing. This can cause an issue when we make network requests. When we make network requests in a synchronous fashion the code is executed sequentially from top to bottom. This means when one network request is executed, the remaining code has to wait until that network request is completed, which can dramatically reduce the user experience. The solution for that is to make asynchronous callbacks for network requests where the operations run in parallel. The Promise API provides a simpler way to manage asynchronous operations compared to the traditional asynchronous callbacks. The Promise API can also be easily integrated with React, which helps us to create asynchronous operations easily when making network requests in React applications. However, since JavaScript is natively synchronous, the modern way to fetch data is with the **`async`**/**`await`** functions, which help you write asynchronous code in a synchronous manner. We'll take a look at how to use **`async`**/**`await`** in this chapter, and we will also learn how to use async/await in loops, which is important to understand when fetching another set of data in parallel from the server at the same time.

We have seen the Promise API in bits and pieces in the previous chapter, now we will take a deep dive into it.

WHAT IS THE PROMISE API?

As mentioned earlier, the Promise API helps manage asynchronous operations when making network requests to the server. It is a JavaScript object that contains the future values of an async operation, either a resolved value or a reason if it was rejected. A promise has three states:

- **`Resolved`**: The action/operation has succeeded.
- **`Rejected`**: The action/operation is failed.
- **`Pending`**: The action/operation is pending; it's not fulfilled or rejected yet.

We have seen how to fetch data using the promise-based Fetch API in *Chapter 14*; *Fetching Data by Making API Requests*, let's review that again and see what values are getting returned.

Let's add the following code in the Chrome Console window:

```
fetch('https://jsonplaceholder.typicode.com/posts/1')
  .then(function(res) {
    return res.json();
  }).then(function(data) {
    console.log(data);
  });
```

The following screenshot shows the values returned:

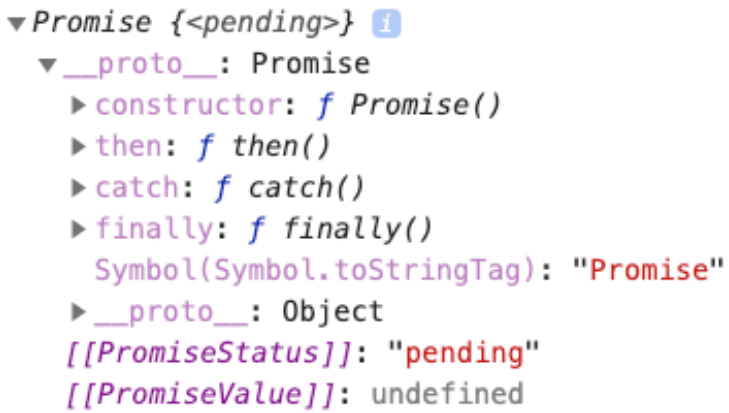

Figure 15.1: Returned promise

As you can see, the promise was returned with the **`PromiseStatus`** as resolved (fulfilled). The Promise API also comes with several methods, such as **`then()`**, **`catch()`** and **`finally()`**. We have already learned how to use **`then()`** and **`catch()`** methods in *Chapter 14*; *Fetching Data by Making API Requests*, but let's go over them again:

- **`then()`**: Returns another promise, which allows us to chain multiple calls, and we can receive a response such as data returned from a server.
- **`finally()`**: Also returns another promise, but the difference from the **`then()`** method is that **`finally()`** will still run regardless of the previous promise's outcome, whether it has been either fulfilled or rejected.
- **`catch()`**: Also returns another promise, but only handles the rejected cases.

To understand promises better, let's think about a hypothetical situation.

Let's imagine your partner is going to do grocery shopping while you are preparing a meal. You ask your partner to buy some eggs, cheese, and spaghetti to make carbonara and also to pick up clothes from a dry cleaner. Your partner **promises** that they will get the food and also pick up the clothes and bring them home. After a few hours, your partner comes back home with all the food **successfully** but with no clothes with the **reason** that the dry cleaner was closed.

If we translate this scenario into a JavaScript promise, this time with **`Axios`**, we could write the code like this:

```
axios.all([
    axios.get('grocery-shopping-mall'),
    axios.get('dry-cleaner')
  ])
  .then(axios.spread((groceryRes, dryCleanerRes) => {
    // bought all food
  })
  .catch((groceryErr, dryCleanerErr) => {
    // dry cleaner is closed
  })
  .finally(() => {
    // shopping completed
  });
```

As **`Axios`** is a promise-based HTTP client, it provides all three methods, **`then()`**, **`catch()`** and **`finally()`**, as specified on the documentation page, and here is an example derived from the documentation page:

```
axios.get('/user?ID=12345')
  .then(function (response) {
// handle success
console.log(response);
  })
  .catch(function (error) {
// handle error
console.log(error);
  })
  .finally(function () {
// always executed
  });
```

As we can see, the **`then()`** and **`catch()`** functions are designed to catch the errors thrown during network requests, while **`finally()`** gets executed each time regardless of whether a network error is thrown. Let's look into the following exercise to see how to fetch data through promises.

Throughout this chapter, we are going to use the **`Openweather`** API https://packt.live/2WU2L9z for the exercises. The **`Openweather`** API provides weather-related data collection. To register, sign up at https://packt.live/3bx042Q. Once you are registered, you should be able to get the API key on the API keys page.

EXERCISE 15.01: FETCHING DATA THROUGH PROMISES

In this exercise, we are going to practice how to make use of the **`then()`**, **`catch`**, and **`finally()`** methods with **`Axios`** in React. We will use the **`useState`** hook to create a controlled form component. At the end of this exercise, you should have the outcome shown in the following screenshot and be able to type a city name and display its current temperature.

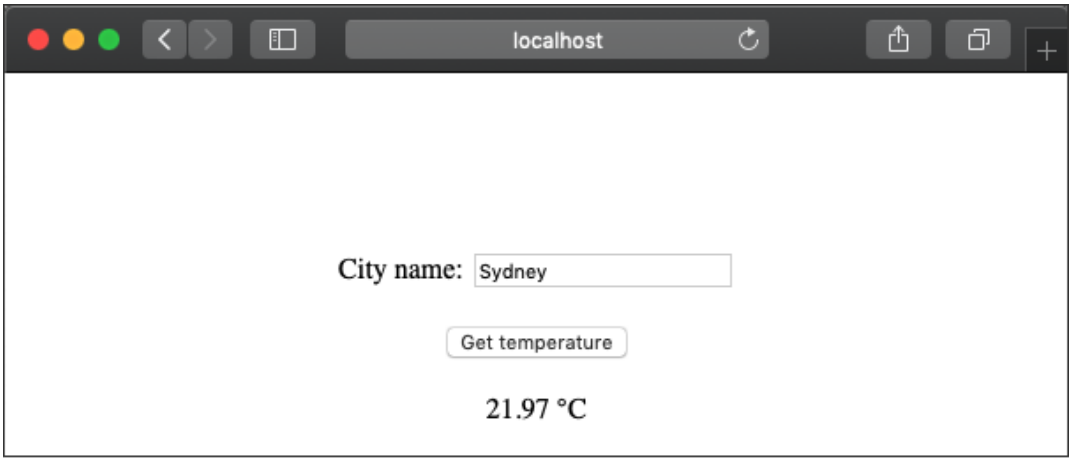

Figure 15.2: The outcome of getting temperature

1. To get started, first install **`create-react-app`** and then follow these steps:
2. Select all the files in the **`src`** folder and remove them all. We are going to create our own files.

3. Create a new file called **`index.js`** in the **`src`** folder and add the following code:

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.render(<App />, document.querySelector('#root'));
```

4. First, let's create a CSS file and add styles. In the **`src`** folder, create a file called **`App.css`** and add the following code:

```
body {
  margin: 0;
}
.page {
  align-items: center;
  display: flex;
  height: 100vh;
  justify-content: center;
  width: 100vw;
}
.box {
  display: flex;
  flex-direction: column;
}
.weather-button {
  display: block;
  margin: 20px auto 0;
}
.temp {
  padding: 20px;
  text-align: center;
}
```

5. In the **`src`** folder, create another file called **`App.js`** and import React and the CSS file we created earlier. Also add some boilerplate code:

```
import React from 'react';
import './App.css';
const App = () => {
  return <div>App page</div>;
};
export default App;
```

6. Let's add JSX code to display the form and the initial temperature, which will be **`0`**. First, add the page wrapper with the class name page. Inside the page wrapper, add another wrapper called **`box`**. The **`box`** wrapper will align the form and the output of the temperature in the middle horizontally and vertically. Inside the box wrapper, add a form (with the class name **`weather-form`**) with an input field and a button (with the class name **`weather-button`**). Below the form, add a **`div`** element with the class name **`temp`**:

```
import React from 'react';
import './App.css';
const App = () => {
  return (
  <div className="page">
   <div className="box">
    <form className="weather-form">
      <label htmlFor="city">City name: </label>
          <input type="text" id="city" placeholder="Type a city" /></
label>
      <button className="weather-button">Get temperature</button>
    </form>
    <div className="temp">
      0 &#8451;
    </div>
  </div>
</div>
  );
};
export default App;
```

The output will be as follows:

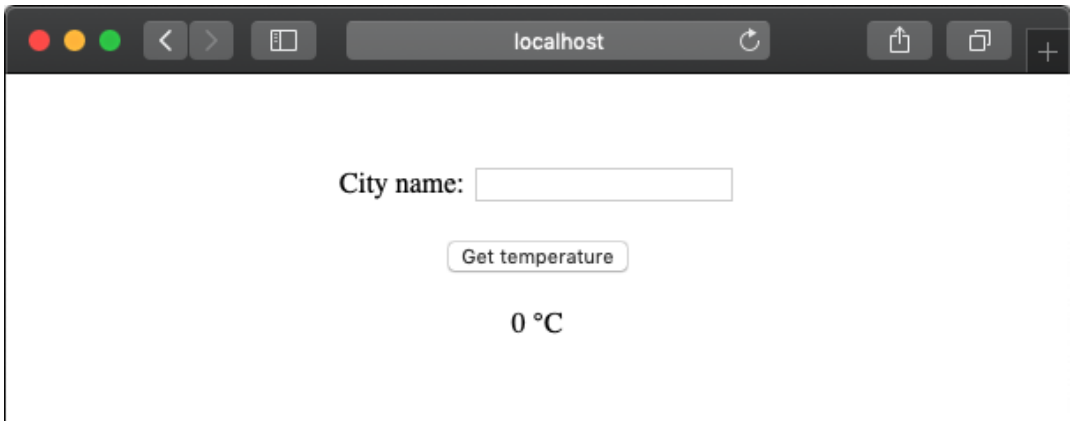

Figure 15.3: The output of the initial temperature form

7. Before we fetch any data, let's create a controlled form component, as discussed in *Chapter 8*, *Introduction to Formik*, to get the city name from the **`input`** field. By using the controlled form component, our React components will control the input field and store the value from the input field to a state.
8. To do that, use the **`useState`** hook. Let's import the hook first and then declare a new state variable called city.
9. Update the **`city`** value when we make changes in the input field, and we are going to use the updated city value for the input field value. This step will allow you to update the city value while typing text in the **`input`** field so that we can use it when fetching data:

```
import React, { useState } from 'react';
import './App.css';
const App = () => {
  const [city, setCity] = useState('');
  return (
  <div className="page">
  <div className="box">
    <form className="weather-form">
      <label htmlFor="city">City name: </label>
           <input type="text" id="city" placeholder="Type a city"
      value={city} onChange={e => setCity(e.target.value)} />
      <button className="weather-button">Get temperature</button>
    </form>
    <div className="temp">
```

```
          0 &#8451;
        </div>
      </div>
      </div>
      );
  };
  export default App;
```

It's time to fetch data.

10. To fetch data, install **`Axios`** by running the yarn add **`axios`** command from your terminal, and to use **`Axios`**, let's import the library:

```
import axios from 'axios';
```

11. Add the **`onSubmit`** attribute to the form element and add a reference to a function called **`submitForm`**. We are going to fetch data when submitting the form:

```
<form onSubmit={submitForm} className="weather-form">
```

12. Create new function called **`submitForm`** above the return method and receive the event. Firstly, add **`e.preventDefault()`**;. By default, when you click on the submit button in a form, the browser will be reloaded. When the browser is reloaded, our React app will be restarted, which means all our previously stored values in a state will be lost. To prevent this from happening, we are going to add **`e.preventDefault();`**:

```
const submitForm = e => {
  e.preventDefault();
};
```

13. Now let's fetch data. To get the current temperature of the city, we are going to use the API from https://openweathermap.org/current. So, the API should look like this:

```
https://api.openweathermap.org/data/2.5/weather?
  q=city-name
  &appid=your-api-key
  &units=metric
```

We are going to send three values parameter values:

- **q**: The **city** name that we will get from the city state updated by **setCity**
- **appid**: Your API key, which can be found on the API keys page.
- **units=metric**: Receiving values in the **Celsius** scale.

Now we can get the city name so the endpoint should look like this:

https://api.openweathermap.org/data/2.5/weather?q=${city}&appid=your-api-key&units=metric

14. Let's fetch the data using Axios. Use the **GET** method, and let's see what data we are getting back:

```
     const submitForm = e => {
  e.preventDefault();


  const url =
`https://api.openweathermap.org/data/2.5/weather?q=${city}&appid=your-
api-
key&units=metric`;

  axios.get(url)
.finally(() => {
    console.log('loading');
})
.then(res => {
    console.log(res);
})
.catch(err => {
    console.log(err);
});
};
```

The output is as follows:

Figure 15.4: Data received

As the **`finally()`** method is always executed, the loading message was first logged followed by the data received from the server. We can retrieve the temperature from **`data.main.temp`**.

15. Declare a new state called temp with the default value **0**, and update it in **`then()`** method. Finally, let's replace **0** to **`{temp}`** to display the temperature:

App.js

```
5  const App = () => {
6  const [city, setCity] = useState('');
7  const [temp, setTemp] = useState(0);
8
9  const submitForm = e => {
10  e.preventDefault();
11
12  const url =
`https://api.openweathermap.org/data/2.5/weather?q=${city}&appid=your-api-
key&units=metric`;

14    axios
15  .get(url)
16  .finally(() => {
17      console.log('loading');
18  })
19    .then(res => {
20      console.log(res);
21      setTemp(res.data.main.temp);
```

The complete code can be found here: https://packt.live/2zAbxl5

16. Let's search **`Seoul`** to get the current temperature. The output will be as follows:

Figure 15.5: Getting the temperature in Seoul

And if we search for a city name that doesn't exist (for example, **dadasdasd**), we will catch an error and the error should look like this:

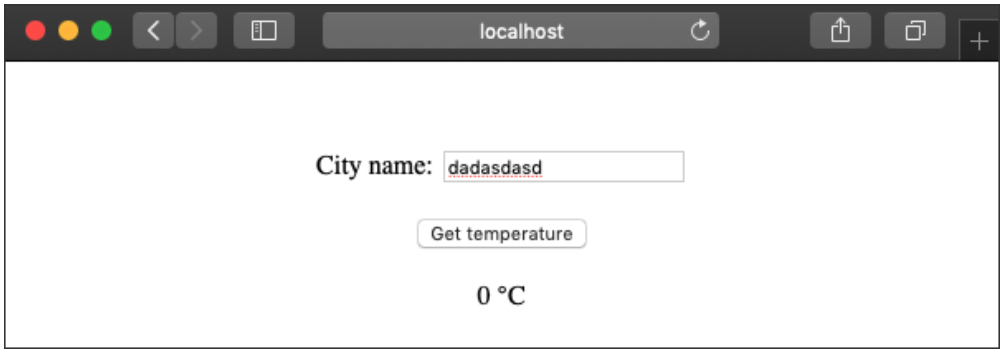

Figure 15.6: Searching for a city that doesn't exist

Let's look at the error console:

```
loading                                                    App.js:17
Error: Request failed with status code 404                 App.js:24
    at createError (createError.js:17)
    at settle (settle.js:19)
    at XMLHttpRequest.handleLoad (xhr.js:60)
```

Figure 15.7: Getting an error message

As we can see in the preceding screenshot, the **GET** method returns a **404**-error message since the city name doesn't exist.

Let's now look at the **async/await** methods, which will help us to structure asynchronous operations in a synchronous way.

WHAT IS ASYNC/AWAIT?

In the previous section, we learned what a promise is and went through an exercise to see how to use it when fetching data from a server. Promises allow us to easily handle asynchronous operations. Since ECMAScript 2017, **async/await** has been added, and it provides a new way to write asynchronous code. However, **async/await** is not a completely new feature; rather it is a syntax sugar on top of promises, and it makes asynchronous code easier to read and write.

> **NOTE**
>
> **async/await** is not supported in Internet Explorer and older browsers, so please use it with caution.

As the name **async/await** suggests, it consists of two keywords, async and await. Let's talk about the **async** function first.

ASYNC

The **async** function helps us to write promise-based code in a synchronous fashion but without blocking the execution thread. The rest of the code runs in parallel along with its execution.

The **async** keyword is added before the function, and that means the function will always return a promise. For example, let's create an async function called **currentWeather()**:

```
async function currentWeather() {
  return 'sunny';
}
```

It returns a resolved promise with **sunny**.

AWAIT

await is only used inside **async** functions, and it waits for a promise to **resolve** or **reject**. It is to be noted that it makes the **async** function block and not the entire program execution. **await** causes **async** execution to pause until a promise is settled, either **fulfilled** or **rejected**. The **async** function resumes execution after fulfillment:

```
async function currentWeather() {
  const res = await fetch(url);
  return res.json();
}
```

In the preceding example, the function pauses its execution while fetching the data, and resumes when the promise settles and returns its response.

We have discussed what the async and await keywords are. Now let's discuss how we can handle three promise methods, **then()**, **error()**, and **finally()**, with **async/await**. To quickly recapitulate from the previous chapter:

- **then()**: Returns a promise with callback functions in success and failure cases
- **catch()**: Returns a promise and deals with rejected cases
- **finally()**: Returns a promise and runs with successful and rejected cases

THEN()

In the code example in the **`await`** section, we didn't have to write a **`then()`** method. Instead, we created an **`async`** function and handled the response inside the function, which makes the code easier to read and write.

ERROR()

To catch an error with a promise with both the Fetch API and Axios, we used the **`catch()`** method in Exercise 15.01. With async/await, we can now handle errors synchronously with **`try`**/**`catch`**:

```
async function currentWeather() {
  try {
  const res = await fetch(url);
  return res.json();
  } catch(err) {
  console.log(err);
  }
}
```

This example will catch errors with fetching the URL and returning the response. If we don't catch the error inside the **`async`** function, we can catch errors from the function itself, as in the following code:

```
async function currentWeather() {
  const res = await fetch(url);
  return res.json();
}
currentWeather().catch(console.error(err));
```

FINALLY()

With the Promise API, as we discussed earlier, the **`finally()`** method will still run regardless of the previous outcome. To use the **`finally()`** method with async/await, we can use the **`try/catch/finally`** pattern. We talked about how to use **`try/catch`** in the preceding section. To use the **`finally()`** method, attach it after the **`catch()`** method:

```
async function currentWeather() {
  try {
  const res = await fetch(url);
   return res.json();
  } catch(err) {
  console.log(err);
  } finally() {
    console.log('All done!');
  }
}
```

Now that we have seen how the promise-based methods such as **`then`**, **`catch`**, and **`finally`** are handled with **`async/await`**, let's dive right into understanding how network errors can be better handled with **`async/await`**.

BETTER ERROR HANDLING WITH ASYNC/AWAIT

`async/await` makes code cleaner, easier to read, and easier to write than promises. In this section, we are going to discuss one more benefit that **`async/await`** provides.

`async/await` allows us to handle both synchronous and asynchronous errors with **`try/catch`**. Let's look at an example where the Promise API methods **`then`** and **`catch`** have been used:

```
const fetchData = () => {
  try {
fetch('https://jsonplaceholder.typicode.com/posts/1')
    .then(function(res) {
      return JSON.parse(res.json());
    }).then(function(data) {
      console.log(data);
    });
} catch(err) {
    console.log(err);
```

```
}
};

fetchData();
```

In this example, **`try/catch`** won't catch an error even if **`JSON.parse`** fails. We can clearly see in the following screenshot that the **`console`**.log message in the **`catch()`** method doesn't get displayed:

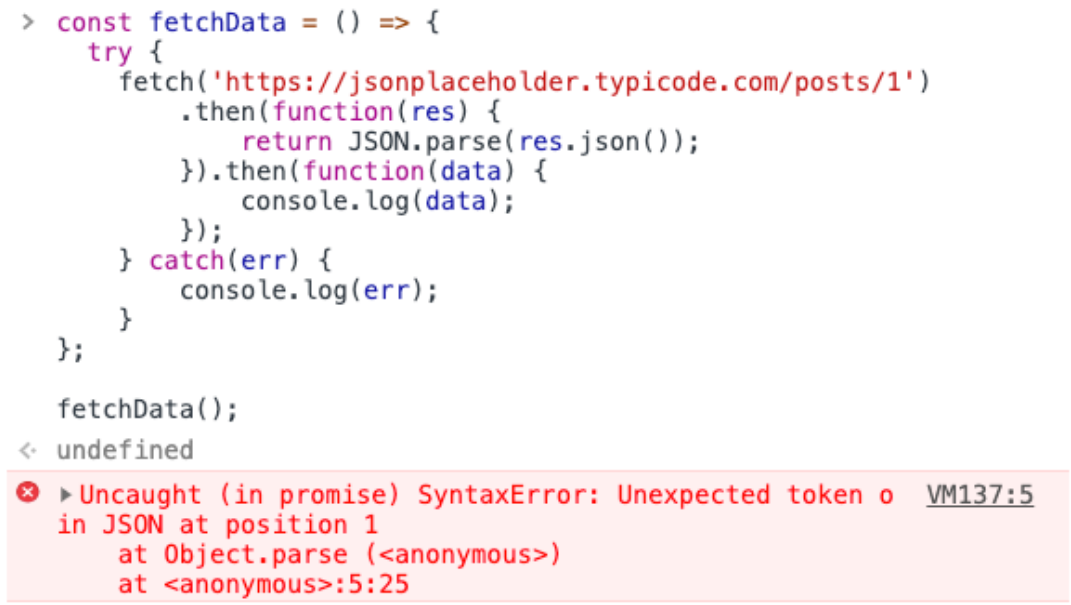

Figure 15.8: Not catching an error

However, with **`async/await`**, we can catch the error with **`try/catch`**, as you can see in the following example:

```
const fetchData = async () => {
  try {
    const res = await fetch('https://jsonplaceholder.typicode.com/
posts/1');
    const data = JSON.parse(res.json());
    console.log(data)
  } catch(err) {
  console.log('Error Message: ', err);
  }
}

fetchData();
```

The error message in the **`catch()`** method is displayed:

```
> const fetchData = async () => {
    try {
      const res = await
fetch('https://jsonplaceholder.typicode.com/posts/1');
      const data = JSON.parse(res.json());
      console.log(data)
    } catch(err) {
      console.log('Error Message: ', err);
    }
}

fetchData();
<- ▶ Promise {<pending>}
Error Message:  SyntaxError: Unexpected token o in JSON    VM193:7
at position 1
    at Object.parse (<anonymous>)
    at fetchData (<anonymous>:4:23)
```

Figure 15.9: Catching the error

We have discussed how to use **`async/await`** and the benefits of it. Now let's go through an exercise to convert the code used in Exercise 15.01 and use **`async/await`** functions instead of promise-based ones.

EXERCISE 15.02: CONVERTING SUBMITFORM() TO ASYNC/AWAIT

In this exercise, we are going to modify the **`submitForm()`** function used in *Exercise 15.01*, *Fetching Data through Promises* and use **`async/await`** in order to initiate a network request. We will see how the network errors are handled better than the approach shown in the previous exercise. Let's see how:

1. Firstly, we are going to add the **`async`** keyword in front of e and wrap **`e`** with parentheses, **`()`**:

```
const submitForm = async (e) => {
```

2. Add the **`await`** keyword in front of the **`axios.get()`** method and store the returned value in a **`const`** variable called **`res`**. Remove all three methods, including **`finally()`**, **`then()`**, and **`catch()`**, and instead update the temp state right after assigning the **`res`** value:

```
const res = await axios.get(url);
setTemp(res.data.main.temp);
```

3. Let's search for **`London`**, and the temperature should be displayed:

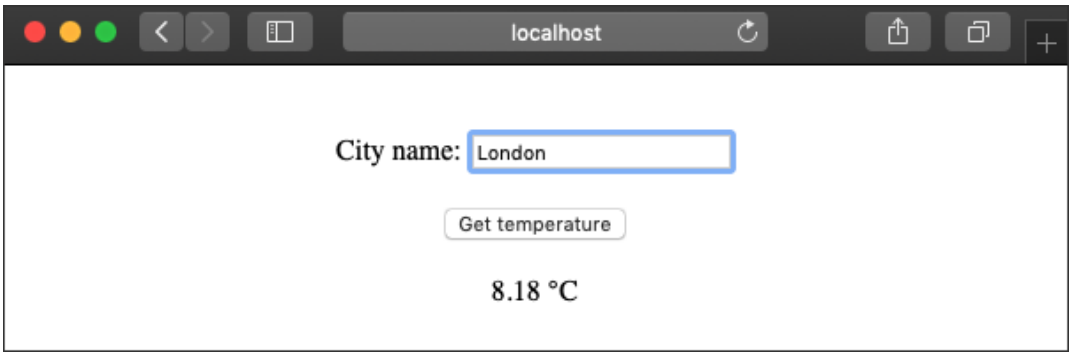

Figure 15.10: Displaying London's temperature

4. To catch any errors, let's add the **`try/catch`** pattern:

```
try {
  const res = await axios.get(url);
  setTemp(res.data.main.temp);
} catch (error) {
  console.error(error);
}
```

5. Now let's search for a city name that doesn't exist (for example, **`xyz`**) and in your **`DevTools`**, you should see that the error has been caught:

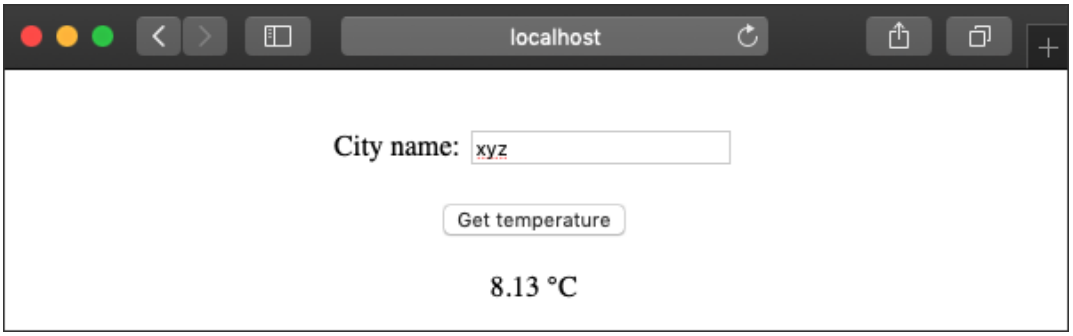

Figure 15.11: Searching for a city name that doesn't exist

The console output is as follows:

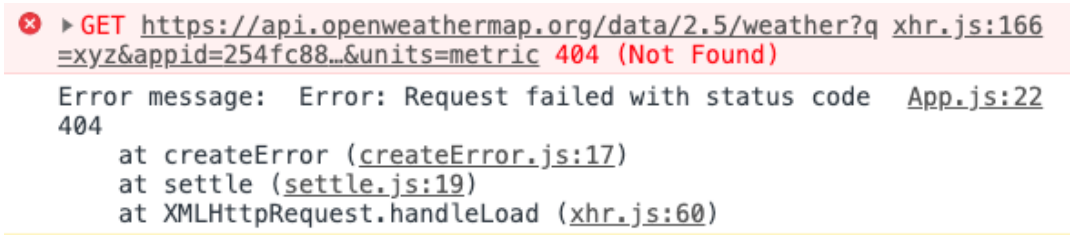

Figure 15.12: Showing the error message

6. Finally, to display the loading message before displaying the temperature, we could use the **`try/catch/finally`** pattern. However, as the finally method is only called after the **`catch()`** method, we need to declare a flag before fetching data using **`Axios`** and once it's done, update the flag value inside **`finally`**. First, let's define a state called loading and declare a flag by updating the loading state value to **`true`**. And in the **`finally()`** method, change the loading state back to **`false`**:

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

const submitForm = async (e) => {
  e.preventDefault();

  setLoading(true);
  console.log(loading); // true

  const url =
`https://api.openweathermap.org/data/2.5/weather?q=${city}&appid=your-
api-key&units=metric`;

  try {
    const res = await axios.get(url);
    setTemp(res.data.main.temp);
  } catch (error) {
    console.log('Error message: ', error);
  } finally {
    setLoading(false);
  }

  console.log(loading); // false
```

```
    };

    return(
      <form onSubmit={submitForm} className="weather-form">
        {loading && <p>Loading...</p>}
      ...
    );
```

The complete code converted to **`async/await`** should look like this:

`App.js`

```
1  import React, { useState } from 'react';
2  import axios from 'axios';
3  import './App.css';
4
5  const App = () => {
6  const [city, setCity] = useState('');
7  const [temp, setTemp] = useState(0);
8  const [loading, setLoading] = useState(false);
9
10  const submitForm = async (e) => {
11  e.preventDefault();
12
13  setLoading(true);
```

`The complete code can be found at:` https://packt.live/3dOxtYF

The output is as follows:

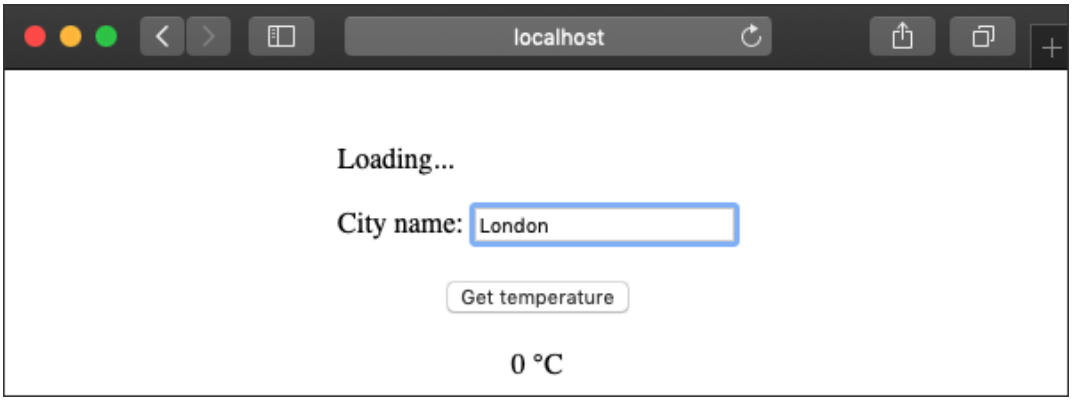

Figure 15.13: Displaying the Loading... message

Let's type in **`London`** to see its current temperature:

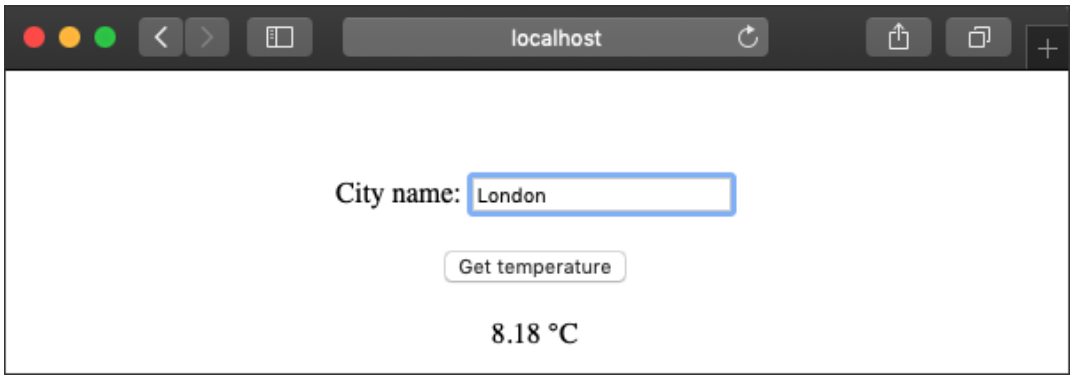

Figure 15.14: Displaying London's temperature

As we can see, **`async/await`** enables us to write asynchronous operations in a synchronous way simply.

Now, let's see how to use **`async/await`** within loops. It could be used in scenarios when we want to fetch data parallelly.

ASYNC/AWAIT WITHIN LOOPS

You will often be in a situation where you will need to fetch data using loops. For example, when you are fetching dataset from a database of the football teams in England and you're trying to get the player photos from the database along with the player names, you may need to loop the football team data and get each player's ID. And then you will be able to retrieve the player's photo and name using that ID.

We are going to talk about three pretty basic loops for **`async/await`**:

- **`for...of`**: For looping in sequence.
- **`forEach`**: We will talk about why we should not use this method.
- **`map`**: For looping in parallel.

To explain these, we are going to use a code example that will return each value from an array every second. Before looping, we will print **`Started`** in the console log and after looping, we will print **`Finished`** in the console log and see if the **`console.log`** message appears in the correct order.

For the initial code, we will create the timeout and sleep functions to exceed the **1**-second timeout:

```
const timeout = ms => {
  return new Promise(resolve => setTimeout(resolve, ms));
};
const sleep = number => {
  return timeout(1000).then(() => number);
};
const numbers = [0, 1, 2, 3];
const loop = async () => {
  console.log('Started');

  // looping goes here
  console.log('Finished');
  return;
};
// invoking the loop function
loop();
```

If we run the following code, we will get Started and then Finished:

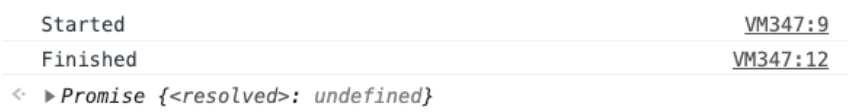

Figure 15.15: Order of looping

First, let's talk about the use case of **`for...of`**.

FOR...OF

The **`for...of`** statement iterates objects such as strings and arrays. When we add the following code between **`console.log('Started');`** and **`console.log('Finished');`**, we will get **`Started`** followed by **`0`**, **`1`**, **`2`**, and **`3`** every **`1`** second, and then **`Finished`**:

```
for (let number of numbers) {
  const output = await sleep(number);
```

```
  console.log(output);
}
```

Figure 15.16: Output of for...of function

Because **`sleep()`** returns a promise, we can wait for a promise to resolve and capture in the console. Therefore, **`for...of`** is great if you need to get data in sequence.

FOREACH()

The **`forEach`** method is great if you need to call a function once and iterate each element in an array. In the following code example, you may expect the same result as **`for...of`**:

```
numbers.forEach(async number => {
  const output = await sleep(number);
  console.log(output)
});
```

However, the results you will be getting might be different from what you might have expected:

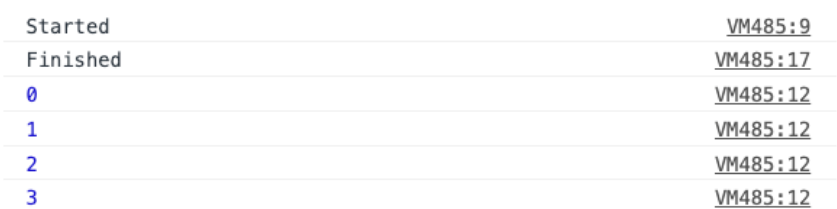

Figure 15.17: Output of forEach function

This is because **`forEach`** is not aware of promises, so async and await do not work with **`forEach`**. If you wanted to return an array of promises, use the **`map()`** method instead of **`forEach`**.

MAP()

Similar to **`forEach`**, the **`map()`** method also calls a function once on every element in the array; however, it creates a new array with the returned values.

To receive the data in parallel, you can use **`map()`** because **`map()`** always returns promises if they are used with **`await`** in an array. You can use **`Promise.all()`** to resolve all the **`iterable`** promises:

```
const numberPromise = await numbers.map(async number => {
  const output = await sleep(number);
  return output;
});
const numbersOutput = await Promise.all(numberPromise);
console.log(numbersOutput);
```

The output should be as follows:

```
Started                                    VM526:9
▶ (4) [0, 1, 2, 3]                         VM526:15
Finished                                   VM526:20
```

Figure 15.18: Output of map function

As we can see in the preceding code, **`numberPromise`** will contain **`iterable`** promises and we can use **`Promise.all()`** to get the output of **`numberPromise`**.

We now have a good understanding of promises and how to use async/await to fetch data. Also, we have learned how to catch errors using the **`try/catch`** pattern. Moreover, we have discussed three different looping methods and how **`async/await`** works with each loop method. We found that it is great to use the **`for...of`** method to get data in sequence, and the **`map()`** method to get data in parallel. With what we have learned so far, let's dive into the activity.

ACTIVITY 15.01: CREATING A MOVIE APP

The aim of this activity is to build an app that displays the five most popular movies when you click the search button. The list of displayed movies will include the poster image, the title, and photos of three members of the cast. To get the details of the movies, we are going to fetch data from The Movie Database API, and we are going to make use of **`async/await`** when fetching data.

The Movie Database provides a huge amount of movie data with various different API endpoints. In this activity, we are especially going to use two API endpoints to get the five most popular movies at the current time and three members of the cast for each movie.

The complete app should look as shown in the following screenshot:

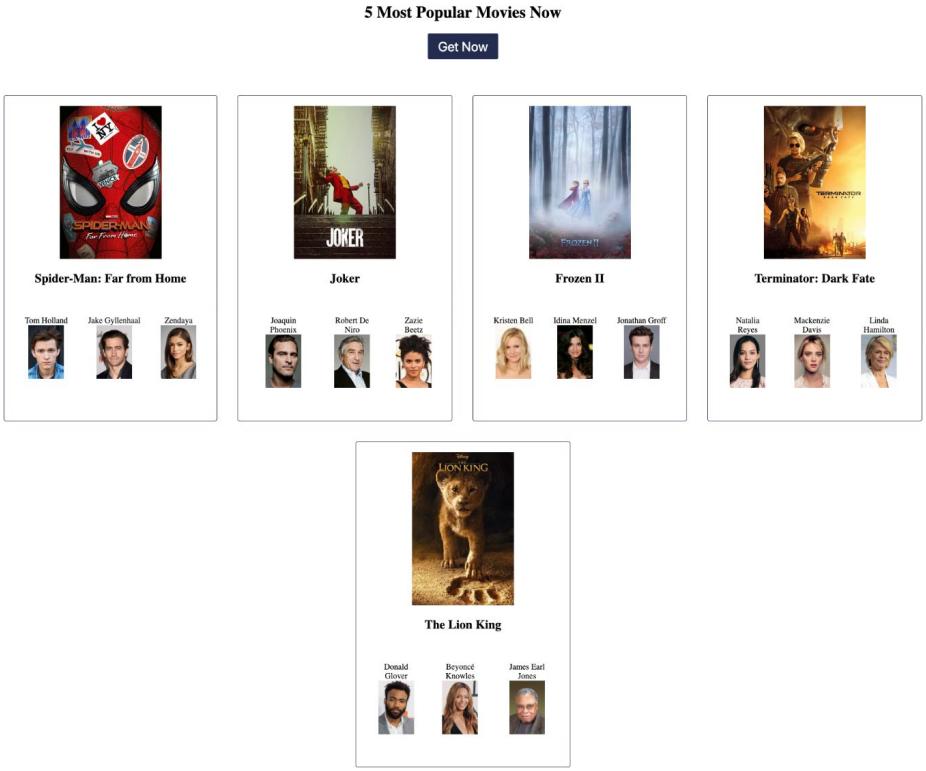

Figure 15.19: Completed movie app

The following steps will help you complete the activity:

1. To use The Movie Database API:
2. First, you need to register in order to get an API key. To create your account, please go to the registration page, https://www.themoviedb.org/account/signup.
3. Once you are registered, head to the API documentation page, https://developers.themoviedb.org/3/getting-started/introduction, where you can find all the necessary details of how to get the movie data.
4. For this activity, you need to take a closer look at the Movie Discover page, https://developers.themoviedb.org/3/discover/movie-discover, and the Get Credits page, https://developers.themoviedb.org/3/credits/get-credit-details , in the Movies section. To fetch the cast members, you will need to get the popular movie data first and then use the movie id to get the cast member data by looping over the movie data. To fetch the data, use **`async`**/**`await`** with Axios.

5. To test the API endpoint, you could use Postman, as we discussed in the previous chapter, or The Movie Database's documentation page allows you to directly test the endpoint. To test the endpoint, click on the **`Try it out`** tab, add the necessary details, and send a request for testing:

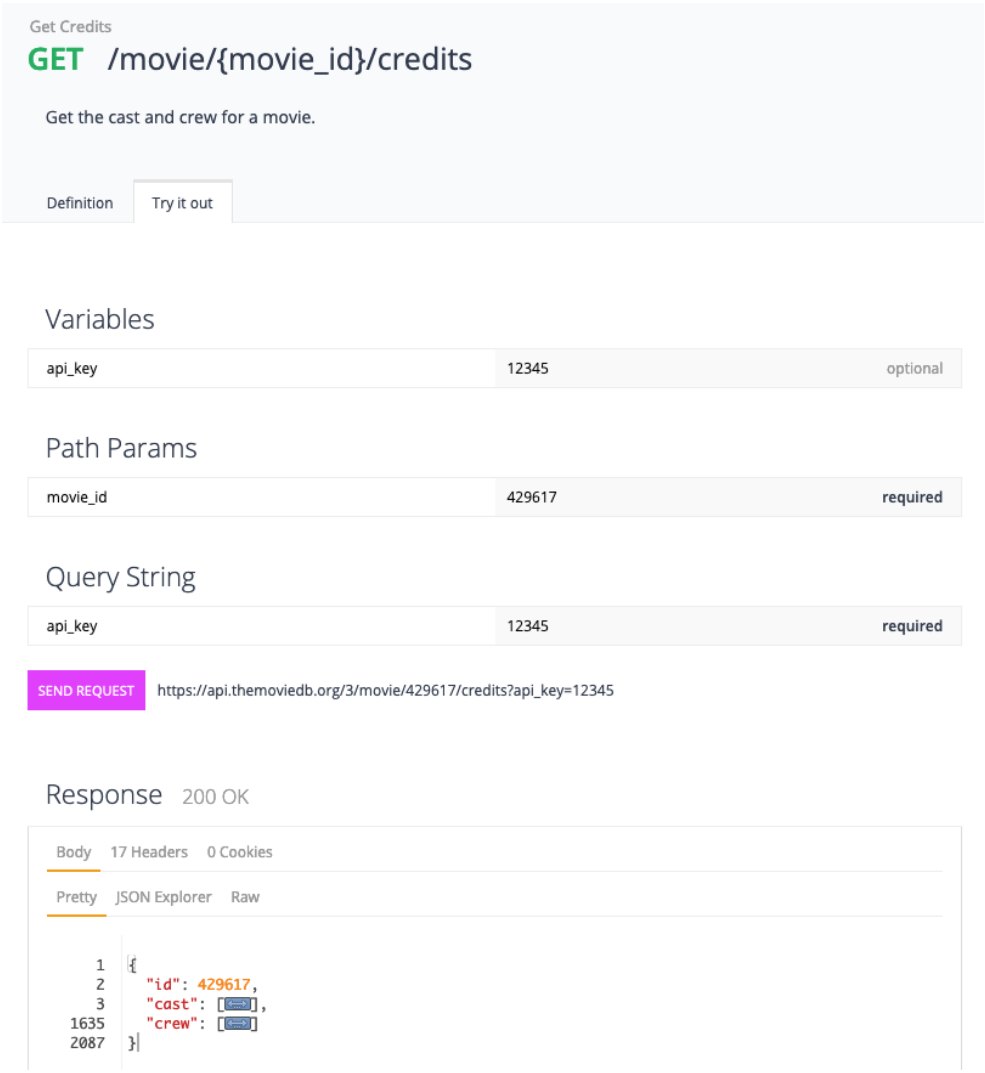

Figure 15.20: The Movie Database documentation to test the endpoint

For the styles, please use the following code:

App.css

```
.page {
padding-top: 50px;
text-align: center;
}

.button {
background-color: #232e50;
border-radius: 4px;
color: #fff;
 font-size: 26px;
 margin-bottom: 50px;
 padding: 10px 20px;
```

The complete code can be found here: https://packt.live/2WS4Ena

> **NOTE**
>
> The solution of this activity can be found on page 724.

SUMMARY

Throughout this chapter, we practiced how to use **async/await** when fetching data from a server. We also covered how to catch errors by using the **try/catch** pattern followed by how to use **async/await** inside loops. Firstly, we learned what a promise is and how to use it with three methods, **then()**, **finally()**, and **catch()**. Also, we discovered how to use **Axios** to use these three methods. After that, in the exercise, we practiced how to fetch data from **OpenWeather** API using **Axios** with **then()**, **finally()**, and **catch()**. After that, we dived into **async/await** by discussing what the **async** and **await** keywords are for. And then we talked about how to use **then()**, **error()**, and **finally()** methods with **async/await**, followed by an example of a benefit of using **async/await** over Promise while handling errors. In addition, we discussed how to use **async/await** within loops and saw an example of when to use **for...of** and **map()**, and also why the **forEach()** method is not recommended.

Having a good understanding of how to fetch data from servers is essential because most modern React-based apps require getting data from servers and dynamically displaying content on the screen. Throughout the previous and current chapters, we have learned most of the important aspects of fetching data and displaying it on the screen. In the next chapter, we will discuss some common errors you will often face when you fetch data in React.

16

FETCHING DATA ON INITIAL RENDER AND REFACTORING WITH HOOKS

OVERVIEW

This chapter will introduce you to the techniques of fetching data on initial rendering and fixing issues when a component falls into an infinite loop. You will be able to create class-based and functional component custom hooks and refactor the code further for simplicity and reusability. This chapter will give you a complete overview of how to fetch data on initial rendering and fix the common issue you will face with the infinite loop.

INTRODUCTION

In the previous chapter, we learned how to fetch data from servers using async/await and catch errors with the try/catch pattern. Also, we have further practiced how to use the async/await methods inside loops.

Now, let's consider a scenario. If you visit any of the popular social media sites, such as Instagram or Twitter, you get photos and content upon initial rendering of the web page and user interaction, such as clicking on a button or submitting a form, isn't necessary. In such a business use case, it is very common for an app to display content on initial rendering without any user interaction. In this chapter, we will look at the best way to fetch data upon initial rendering of a class-based component and how to achieve the same with a functional component.

Furthermore, for such scenarios, while fetching data from the server, we may encounter an issue with the component falling into an infinite loop due to re-rendering of the component. An example of this could be the state of the component getting updated without any safeguard after fetching data. We will learn how we can solve these issues in both class-based and functional components. We will discuss the custom hook and how we can further refactor the code so that we can reuse the hook to fetch data in such cases. Creating such types of hooks will help us avoid writing repetitive code and make the code reusable in such a way that other developers can easily understand our code.

Let's start with how to fetch data upon initial rendering in the correct manner so that you can avoid the infinite loop.

FETCHING DATA UPON INITIAL RENDERING

You may want to display data upon initial rendering of a component; for example, when you visit YouTube, you will see some trending videos on the initial page load. Now, if we want to design a page or a component of this kind, we could use the life cycle methods that we discussed in *Chapter 4*, *React Lifecycle Methods*. We need to decide which life cycle method to use to fetch data and store it in a state.

According to the life cycle diagram (https://packt.live/2zCiT7P), the component goes through a life cycle in the following order.

The **`constructor()-> render()->componentDidMount.`** **`componentDidMount()`** method is used for mounting steps and is the best life cycle method for fetching and manipulating data received from the server. After fetching data in the method, we update the state of the component with the data fetched within the **`componentDidMount()`** method in order to populate the data in the content.

From the life cycle methods' diagram (https://packt.live/2zCiT7P), it is evident that when the **`componentDidMount()`** life cycle method is running, the **`render()`** method has already been run once, which means if we fetch data in the **`componentDidMount()`** method, the data won't be loaded before the initial rendering. We can avoid assigning any undefined state in such a case, which is a best practice in any case.

While fetching data from a server, we are going to use **`axios`** with **`async/await`**, as we discussed in the previous chapter. Before we start fetching data in the **`componentDidMount()`** method, we need to **`import`** React and **`axios`**:

```
import React, { Component } from 'react';
import axios from 'axios';
```

We are going to create a class-based component and, in the **`constructor ()`**, we will define an **`items`** variable that will contain the data fetched from a server. We will initialize the items state with an empty array and then export the **`App`** component. It should look like this:

```
class App extends Component {
  constructor(props) {
  super(props);
  this.state = {
  items: []
  };
  }
}
export default App;
```

Once the data is received from the server, we can store it in the state with **`setState()`**:

```
class App extends Component {
  constructor(props) {
  super(props);
  this.state = {
```

```
  items: []
  };
  }
  async componentDidMount() {
  const res = await axios.get(
  'https://mydomain.com/api'
  );

  this.setState({
  items: res.data.items
  });
  }
}
export default App;
```

As we can see from the code, we can fetch data using the **`componentDidMount()`** method on initial rendering of the component. We have used **`async/await`** by adding the **`async`** keyword in front of the **`componentDidMount()`** method and the await keyword in front of the **`axios`** method. The async function, which contains the await expression, provides us with a simpler means of using promises synchronously.

To re-render the data received from the server, we could manipulate the state of the **`items`** array inside the **`render()`** method. If the data is a list of items, we could use the **`map()`** method to loop the array and display the items as shown in the following code:

```
class App extends Component {
  constructor(props) {
  super(props);
  this.state = {
  items: []
  };
  }
  async componentDidMount() {
  const res = await axios.get(
  'https://mydomain.com/api'
  );
  this.setState({
  items: res.data.items
  });
  }
```

```
  render() {
  return(
  <ul>
    {
    this.state.items.map((item, index) => (
    <li key={index}>{item.title}</li>
    ))
    }
  </ul>
  );
  }
}
export default App;
```

> **NOTE**
>
> The preceding code is just an example with no valid API endpoint, so there will be no output.

One thing to note here is that the **`render()`** method will be called twice in the code example. When the preceding code is loaded, the **`render()`** method will be called first, followed by **`componentDidMount()`**. When the state is updated in **`componentDidMount()`**, it will call the **`render()`** method again and this will result in the **`render()`** method being called twice. However, as the extra rendering happens before the browser gets updated, users will not notice the intermediate state.

> **NOTE**
>
> To initially assign values to the state, please **DO NOT** do it in the **`componentDidMount`**() method, because updating the state in the **`componentDidMount()`** method will call **`render()`** twice and it may cause performance issues in the component. The best place to initially set the values of state variables is inside **`constructor()`**.

With this knowledge, let's take a look at the following exercise.

EXERCISE 16.01: FETCHING POPULAR GOOGLE FONTS ON INITIAL RENDERING

In this exercise, we will display the top ten Google fonts upon initial rendering of a website. Each font should contain a link to the font's page on Google Fonts. You can access the Developer API at https://packt.live/2T0MLS0, and the key to access this at https://packt.live/2X9hf5N. To get the API key, click the **`Get a key`** button and follow the steps instructed.

At the end of this exercise, we will display 10 popular Google fonts and each list will have a link to each font's site on the Google Fonts website:

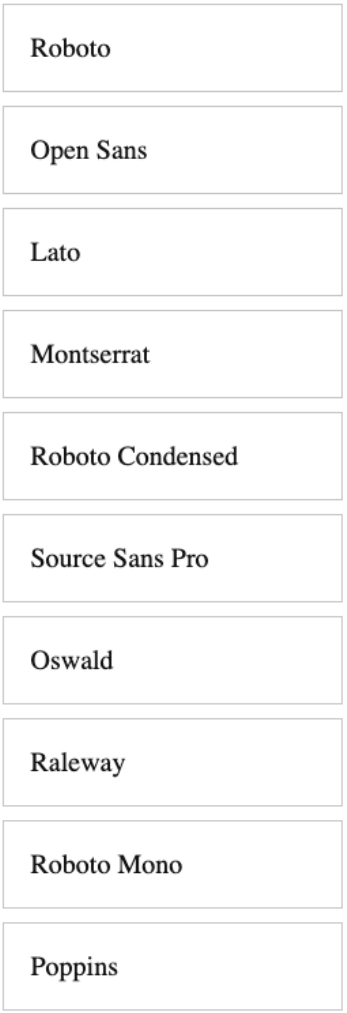

Figure 16.1: The outcome of the 10 most popular Google fonts

1. Start by creating a new React app:

```
$ npx create-react-app font
```

2. Remove all the files present in the **`src`** folder and create **`App.css`**. Then, add the CSS styles as follows:

```
.container {
  margin: 50px auto;
  width: 250px;
}

.card__item {
  border: 1px solid #ccc;
  color: #111;
  display: block;
  font-size: 20px;
  margin: 10px 0;
  padding: 20px;
  text-decoration: none;
  transition: background-color 0.3s ease-in-out;
}

.card__item:hover {
  background-color: #eee;
}
```

3. In the **`src`** folder, create **`index.js`**. Import **`React`**, **`ReactDOM`**, and **`App`**. Also, render the **`<App>`** component in the DOM:

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
ReactDOM.render(<App />, document.querySelector('#root'));
```

4. In the **`src`** folder, create another file called **`App.js`** and import the React and CSS files we created earlier. Also, add boilerplate code and import the **`FontList`** component, which we will create in the next step:

```
import React from 'react';
import FontList from './FontList';
import './App.css';
const App = () => {
```

```
  return (
  <div className="container">
  <FontList />
  </div>
  );
};
export default App;
```

5. We are going to use **`axios`**, so install **`axios`** by typing the following command in the root directory in your terminal:

```
yarn add axios
```

6. Create another file, called **`FontList.js`**, and create a class-based component called **`FontList`**. Add boilerplate code too:

```
     import React, { Component } from 'react';
import axios from 'axios';
class FontList extends Component {
  render() {
  return (
  <div className="card">
    FontList component
  </div>
  );
  }
}
export default FontList;
```

7. Add the API key that we have prepared:

```
import React, { Component } from 'react';
import axios from 'axios';
const API_KEY = '12345';
class FontList extends Component {
  render() {
  return (
  <div className="card">
    FontList component
  </div>
```

```
    );
      }
    }
    export default FontList;
```

8. To fetch the data on initial rendering, use the **`componentDidMount()`** life cycle method. As we are going to use **`async`**/**`await`**, let's add the **`async`** keyword in front of **`componentDidMount()`** and add the method right above **`render()`**:

```
    async componentDidMount() {
      // 1. Fetch data with axios

      // 2. Update state
    }
```

9. Fetch data using **`axios`**. Add the **`await`** keyword in front of **`axios`**, and then add the endpoint. The endpoint for the popular Google fonts should start with https://www.googleapis.com/webfonts/v1/webfonts and add two parameters:

   `- key=${API_KEY}`

   `- sort=popularity`

```
    async componentDidMount() {
      // 1. Fetch data with axios
      const res = await
      axios.get(`https://www.googleapis.com/webfonts/v1/
    webfonts?key=${API_KEY}&so
      rt=popularity`);

      // 2. Update state
    }
```

10. Check the response using **`console.log`**:

```
    console.log(res)
```

You should get results along the lines of the following screenshot. The data we want is in **`data > items`** as an array. Hence, we can get the only data we want from **`res.data.items`**:

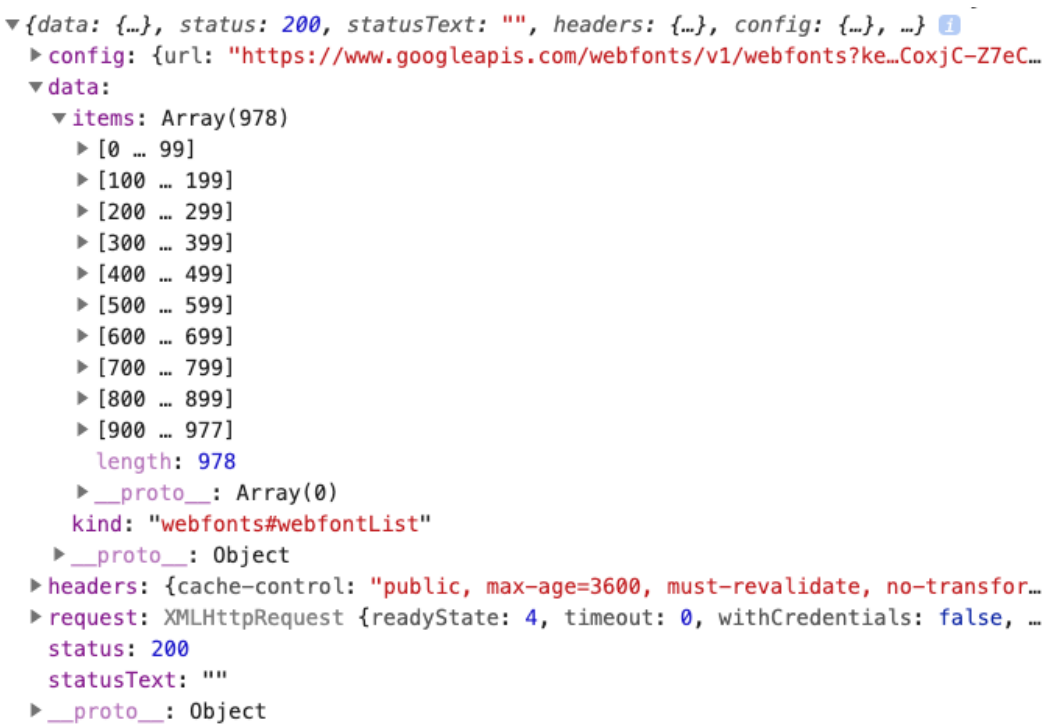

Figure 16.2: The data received from the Google Fonts API

11. Before storing the data in a state, initialize the state in **`constructor()`**:

```
constructor(props) {
  super(props);
  this.state = {
  fonts: []
  };
}
```

12. Update the state with **`res.data.items`**. As we only want to display the 10 most popular Google fonts, we will use the **`slice()`** method to fetch the first 10 items from the **`res.data.items`** array:

```
async componentDidMount() {
  // 1. Fetch data with axios
  const res = await
  axios.get(`https://www.googleapis.com/webfonts/v1/
webfonts?key=${API_KEY}
          &sort=popularity`);

  // 2. Update state
  this.setState({
  fonts: res.data.items.slice(0, 10)
  });
}
```

To display the data inside the **`render()`** method, we will loop the **`fonts`** state using the **`map()`** method. We are going to receive two values, each **`font`** and **`index`**. We are going to get the family name from **`font`** and we are going to use **`index`** as a key.

13. Call the **`map`** method with two parameters, **`font`** and **`index`**:

```
render() {
  return (
  <div className="card">
  {this.state.fonts.map((font, index) => (
```

Also, to build the URL to link to the font page on the Google Fonts site, we will start with https://fonts.google.com/specimen/, followed by the font family name, but we need to replace the space with **+**.

14. Replace the space, `' '`, with **+**:

```
render() {
  return (
  <div className="card">
  {this.state.fonts.map((font, index) => (
    <a
    href={`https://fonts.google.com/specimen/${font.family.replace(
    ' ',
    '+'
    )}`}
```

```
        className="card__item"
        key={index}
        >
        {font.family}
        </a>
      ))}
      </div>
      );
    }
```

Once you have followed these steps, you should get the 10 most popular Google fonts, as indicated in the following screenshot. Also check whether the link takes you to its corresponding Google Fonts page:

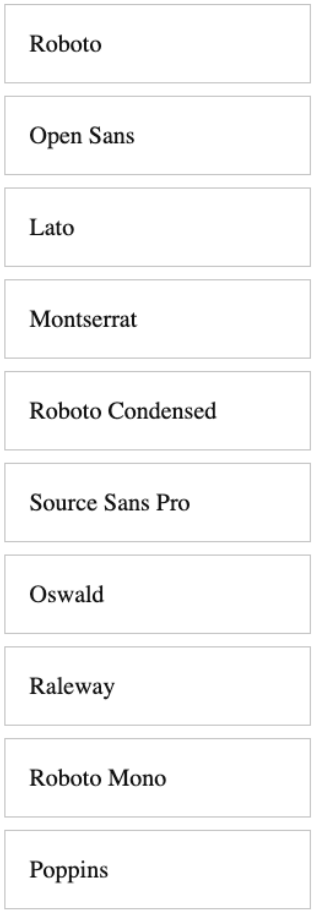

Figure 16.3: The outcome of the 10 most popular Google fonts

We have seen how we can retrieve the 10 most popular Google fonts and display them while the component loads for the first time. Now, let's understand how to fetch data while a component gets updated.

FETCHING DATA ON UPDATE

We now have a good idea of which life cycle method we need to use while fetching data upon initial rendering. In this section, we are going to learn which life cycle method we need to use while fetching data while a component gets updated.

Suppose a child component is fetching data from the server with the values sent from the parent component. Now, if we update the state of the parent component and use the **`componentDidMount()`** method in the child component to fetch data from the server simultaneously, we won't be able to do so even if the child components are re-rendered. This is because the **`componentDidMount()`** method is only called once in the life cycle of any component (discussed in *Chapter 4*, *React Lifecycle Methods in React*).

The following diagram shows an example of a parent and child component:

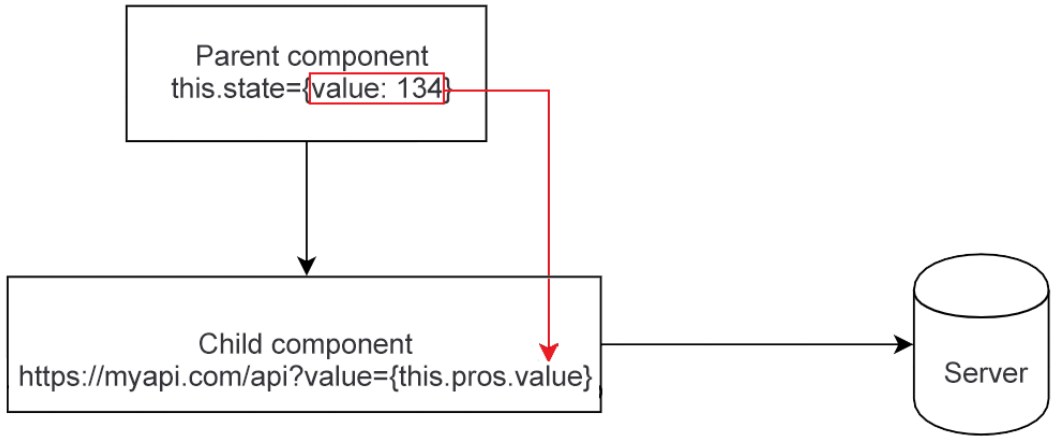

Figure 16.4: Updating the state from the parent component

The parent component has a state value of **`134`** and we send this value, **`134`**, to the child component. We are using **`134`** as part of the API endpoint. Let's say that we want to fetch data using a different value this time, **`987`**. We update the state in the parent component, which will re-render the child component. We expect that we will receive the new value, **`987`**, in the child component and reconstruct the API endpoint, which allows us to fetch another set of data. However, this fetching of the data will ideally never happen as the **`componentDidMount()`** method is called only once during the life cycle of a component, and therefore will not be called again when the updating occurs.

In this case, the best place to fetch data is in the **`componentDidUpdate()`** method. Hence, updating the state in the parent component will re-render the child component and the **`componentDidUpdate()`** method in the child component will be invoked. Let's look at the following code in order to understand how we will implement the method:

```
async componentDidUpdate() {
  const res = await axios.get(
  'https://mydomain.com/api'
  );
  this.setState({
  items: res.data.items
  });
}
```

As the **`componentDidUpdate()`** method cannot be used for the initial render, if we still wanted to fetch data upon initial rendering, we also need to keep fetching data in the **`componentDidMount()`** method:

```
async componentDidMount() {
  const res = await axios.get(
  'https://mydomain.com/api'
  );
  this.setState({
  items: res.data.items
  });
}
```

```
async componentDidUpdate() {
  const res = await axios.get(
  'https://mydomain.com/api'
  );
  this.setState({
  items: res.data.items
  });
}
```

As the code for fetching the data inside **`componentDidMount()`** and **`componentDidUpdate()`** is the same, we can create a new function called **`getData()`** and reuse the code.

> **NOTE**
>
> DO NOT attempt to run this code yet as it will cause an infinite loop.

```
async getData() {
  const res = await axios.get(
  'https://mydomain.com/api'
  );
  this.setState({
  items: res.data.items
  });
}
componentDidMount() {
  this.getData();
}
componentDidUpdate() {
  this.getData();
}
```

In the next section, we will discuss the infinite loop problem that we will encounter when a component updates.

INFINITE LOOP

We now have a good idea of which method to use between **`componentDidMount()`** and **`componentDidUpdate()`** while fetching data. However, it is very important to note that updating the state inside the **`componentDidUpdate()`** method will cause an infinite loop. Let's take a look at the following diagram:

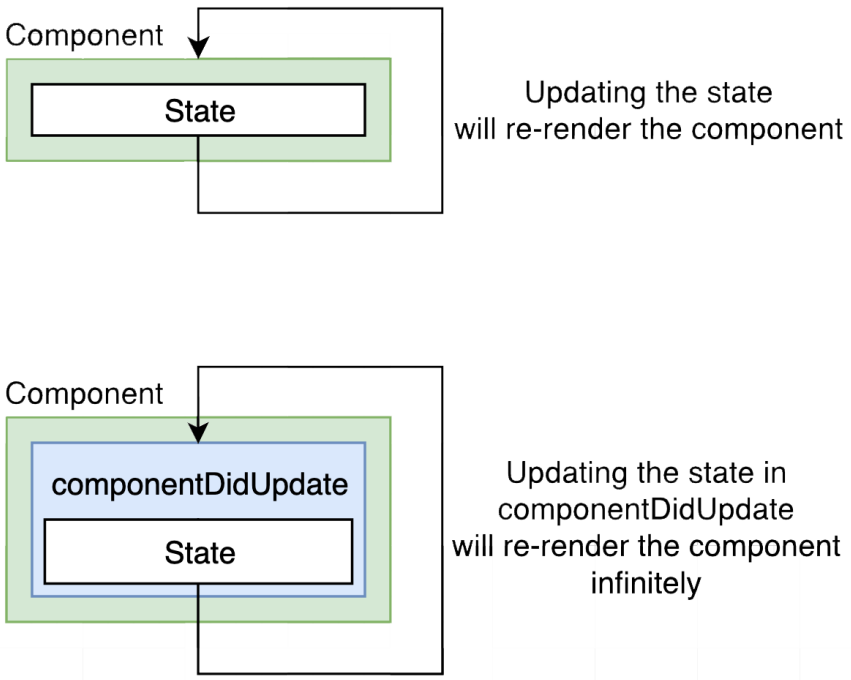

Figure 16.5: Infinite loop

As we can see from the preceding diagram, when the state in the component gets updated, the **`componentDidUpdate`** method will be called again and, therefore, the component will be rendered again. Now, when you update the state in the **`componentDidUpdate()`** method, it will **infinitely re-render** the component.

To avoid this situation, we need a safeguard. The **`componentDidUpdate()`** method returns three values, **`prevProps`**, **`prevState`**, and **`snapshot`** (discussed in *Chapter 4*, *React Lifecycle Methods in React*). We can use the **`prevProps`** value and create a safeguard. We can compare the previous prop value in a child component with the current prop sent from the parent component and, if they are not the same, we can update the state. When we update the state, the component will be re-rendered and this time, since the current prop becomes the previous prop, and their values are the same, we can skip updating the state, which prevents the infinite loop. The following code shows how we compare the previous prop value with the current prop value in the **`componentDidUpdate()`** method:

```
componentDidUpdate(prevProps) {
  if (prevProps.sort !== this.props.sort) {
  this.getData();
  }
}
```

Therefore, the problem of a component falling into an infinite loop can be avoided with this simple trick. Let's see the implementation in more detail in the following exercise.

EXERCISE 16.02: FETCHING TRENDING GOOGLE FONTS

Continuing from *Exercise 16.01*, *Fetching Popular Google Fonts on Initial Rendering*, we are going to add two buttons above the font list, one to get the popular Google fonts, and the other for the trending fonts. We are going to add a state to the **`App`** component and, when clicking on the button, we will update the state to either **`popularity`** or **`trending`**. The state will be sent to the **`FontList`** component as a prop and used as a **`sort`** parameter when fetching data. At the end of this initial rendering with **`componentDidMount`**:

1. In **`App.css`**, add the button styles:

```
    .card__button:first-of-type {
      margin: 10px;
    }
```

2. First, update the **App** component to the class-based component and add an initial **sort** state as popularity. Then, send the **sort** state as a prop to the **FontList** component:

```
import React, { Component } from 'react';
import FontList from './FontList';
import './App.css';
class App extends Component {
  constructor(props) {
  super(props);

  this.state = {
  sort: 'popularity'
  };
  }
  render() {
  return (
  <div className="container">
    <FontList sort={this.state.sort} />
  </div>
  );
  }
}
export default App;
```

3. Add two buttons—one for **popularity** and the other for **trending**. Each button will have a class name of **card__button** and also an **onClick** event, which will call a new function called **handleClick** when each **sort** value is sent:

```
render() {
  return (
  <div className="container">
   <div>
    <button
      className="card__button"
      onClick={() => {
      this.handleClick('popularity');
      }}
    >
    Popularity
```

```
      </button>
      <button
      className="card__button"
      onClick={() => {
      this.handleClick('trending');
      }}
      >
      Trending
      </button>
    </div>
    <FontList sort={this.state.sort} />
    </div>
    );
  }
```

4. Create the **`handleClick`** function and update the **`sort`** state with the value received from each button:

```
handleClick = sort => {
  this.setState({ sort });
};
```

5. From *Exercise 16.01*, *Fetching Popular Google Fonts on Initial Rendering*, add **`componentDidUpdate()`** and add the same code from the **`componentDidMount()`** method. However, make sure to have a safeguard with **`prevProps`**. Also, update the **`sort`** parameter value to the **`sort`** value received as the prop from the **`App`** component:

```
class FontList extends Component {
  constructor(props) {
  super(props);

  this.state = {
  fonts: []
  };
  }
  async componentDidMount() {
  // 1. Fetch data with axios
  const res = await
  axios.get(`https://www.googleapis.com/webfonts/v1/
webfonts?key=${API_KEY}&so
          rt=${this.props.sort}`);
  // 2. Update state
```

```
    this.setState({
    fonts: res.data.items.slice(0, 10)
    });
    }
    async componentDidUpdate(prevProps) {
    if (prevProps.sort !== this.props.sort) {
    const res = await axios.get(
    `https://www.googleapis.com/webfonts/v1/webfonts?key=${API_
KEY}&sort=${th
     is.props.sort}`);
    this.setState({
      fonts: res.data.items.slice(0, 10)
    });
    }
    }
```

6. Update the **`render`** function as we did for the previous exercise:

```
    render() {
    return (
    <div className="card">
      {this.state.fonts.map((font, index) => (
      <a
       href={`https://fonts.google.com/specimen/${font.family.replace(
        ' ',
        '+'
      )}`}
      className="card__item"
      key={index}
      >
      {font.family}
      </a>
      ))}
```

```
    </div>
    );
    }
}
export default FontList;
```

7. Refactor both the **`componentDidMount()`** and **`componentDidUpdate()`** methods so that we can reuse the duplicate code:

```
async getFonts() {
  // 1. Fetch data with axios
  const res = await
  axios.get(`https://www.googleapis.com/webfonts/v1/
webfonts?key=${API_KEY}&so
          rt=${this.props.sort}`
  );

  // 2. Update state
  this.setState({
  fonts: res.data.items.slice(0, 10)
  });
}

componentDidMount() {
  this.getFonts();
}

componentDidUpdate(prevProps) {
  if (prevProps.sort !== this.props.sort) {
  this.getFonts();
  }
}
```

8. Click on the buttons and we should see that the list of fonts gets updated between **`trending`** and **`popularity`**:

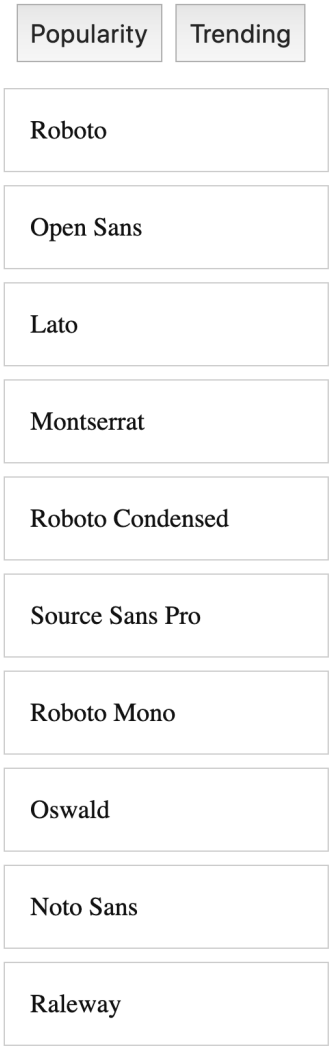

Figure 16.6: The output of sorting buttons

Hitherto, we've seen how to fetch data for class components using the life cycle methods. Now, we're going to learn how to fetch data for functional components. Since, by now, we already have a good understanding of React Hooks, let's dive right into using React Hooks to fetch data.

REACT HOOKS TO FETCH DATA

We have learned about the life cycle methods we need to use for fetching data and how to fix the infinite loop when updating the state in the **`componentDidUpdate()`** method.

In early 2019, React 16.8 was released with a stable release of React Hooks. As we discussed in the *Chapter 15*, *Promise API and async/await*.

React Hooks allows us to use state and other features in the function components without writing a class. In this section, we are going to learn how to use React Hooks to refactor what we have done in the previous sections, such as fetching data or updating the state by clicking on either the **`Popularity`** or **`Trending`** buttons.

In the class-based components, we have used the **`componentDidMount()`** method to initially fetch data, and the **`componentDidUpdate()`** method to re-fetch data upon re-rendering. To avoid the infinite loop, we used **`prevProps`** to compare whether the previous prop is the same as the current prop.

With **`useEffect`**, we can do both initial requests and follow-up requests in the same place. To recap **`useEffect`**, we can send a second argument to **`useEffect`** and, if the second argument has changed (meaning it has different values than the previous rendering), the function in **`useEffect`** will be called, whereas if the second argument is the same, the function inside **`useEffect`** won't be called:

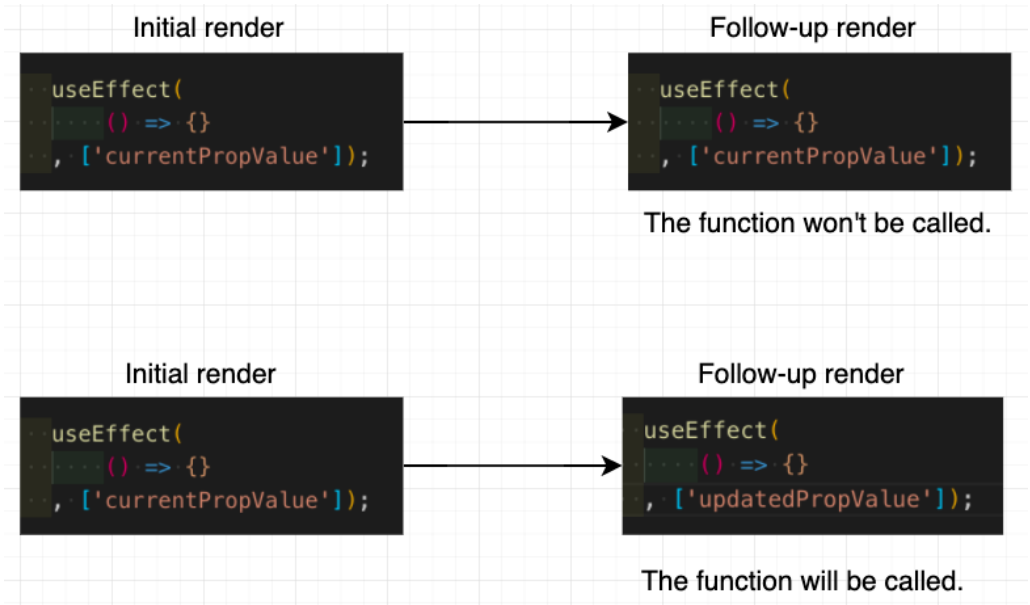

Figure 16.7: Setting the second argument

From the diagram, when we send the same value as the second argument to **`useEffect`** and **`currentPropValue`**, the function inside **`useEffect`** won't be called. However, when we send a different value, **`updatedPropValue`**, the function inside the **`useEffect`** will be called and we can execute the code in the function on the follow-up rendering.

Now, let's see how we can fetch data using the **`useEffect`** hook.

To fetch data in **`useEffect()`**, add the **`axios`** call inside **`useEffect`**, followed by updating the state using **`useState()`**:

```
useEffect(async () => {
  const res = await axios.get(
  'https://mydomain.com/api'
  );

  setRes(res);
}, [propValue]);
```

However, if you run the preceding code, you will receive an error message because we are not allowed to use **`useEffect`** if we are passing an async function, or if the function returns a promise:

```
  Line 9:13:  Effect callbacks are synchronous to prevent race conditions. Put
the async function inside:

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

Learn more about data fetching with Hooks: https://fb.me/react-hooks-data-fetchi
ng  react-hooks/exhaustive-deps
```

Figure 16.8: An error message

To fix this error message and call the **`async`** function, we need to create a second function and call it inside **`useEffect`**:

```
useEffect(() => {
  const getRes = async () => {
  const res = await axios.get(
  'https://mydomain.com/api'
```

```
  );

  setRes(res);
  }

  getRes();
}, [propValue]);
```

In this way, we can fetch data for the follow-up rendering as well as the initial rendering and avoid the infinite loop because receiving a different **`propValue`** will not execute the code inside the **`useEffect`** method and will safeguard the infinite re-rendering.

EXERCISE 16.03: REFACTORING THE FONTLIST COMPONENT

In this exercise, we are going to refactor the **`FontList`** class-based component we completed in Exercise 16.02 to the function component using React Hooks. During this exercise, you will learn how to use **`useState`** to handle the state and **`useEffect`** to fetch data in the functional component. By the end of this exercise, you should be able to know how to properly use **`useEffect`** to fetch data by avoiding the infinite loop issue.

Let's look at the completed **`FontList`** class component:

`FontList.js`

```
1  import React, { useState, useEffect } from 'react';
2  import axios from 'axios';
3
4  const API_KEY = '12345';
5
6  const FontList = props => {
7  const [fonts, setFonts] = useState([]);
8
9  useEffect(() => {
10  const getFonts = async () => {
11  // 1. Fetch data with axios
12  const res = await axios.get(
13    `https://www.googleapis.com/webfonts/v1/webfonts?key=${API_KEY}&sort=${prop
      s.sort}`
14  );
15
16  // 2. Update state
```

`The complete code can be found here:` https://packt.live/3dGTDvR

9. First, we are going to change the class to function and remove **`Component`** from the **`import`** statement:

```
import React from 'react';
import axios from 'axios';

const API_KEY = '12345';

const FontList = props => {
…
```

10. Remove the constructor. We will initialize the **`fonts`** state using the **`useState`** hook. To use the **`useState`** hook, we need to import it:

```
import React, { useState } from 'react';
import axios from 'axios';

const API_KEY = '12345';

const FontList = props => {
  const [fonts, setFonts] = useState([]);
…
```

11. Remove the **`render()`** method and return the JSX code directly:

```
import React, { useState } from 'react';
import axios from 'axios';

const API_KEY = '12345';

const FontList = props => {
  const [fonts, setFonts] = useState([]);
  return (
  <div className="card">
  {fonts.map((font, index) => (
    <a
    href={`https://fonts.google.com/specimen/${font.family.replace(
    ' ',
    '+'
    )}`}
```

```
      className="card__item"
      key={index}
      >
      {font.family}
      </a>
    ))}
    </div>
    );
}

export default FontList;
```

12. Between **`useState()`** and **`return()`**, add **`useEffect()`** to fetch data and add **`props.sort`** as the second argument. The value of **`props.sort`** will be either **`popularity`** or **`trending`**, and this value will be updated by the parent component, **`App`**. So, unless the **`App`** component changes, the function inside **`useEffect`** will only be called once:

```
useEffect(() => {

}, [props.sort]);
```

13. Let's add **`getFonts()`** inside **`useEffect()`** and call it:

```
useEffect(() => {
  const getFonts = async () => {
  // 1. Fetch data with axios
  const res = await axios.get(
  `https://www.googleapis.com/webfonts/v1/webfonts?key=${API_
KEY}&sort=${pro
   ps.sort}`
  );

  // 2. Update state
  setFonts(res.data.items.slice(0, 10));
  };

  getFonts();
}, [props.sort]);
```

Once you have completed the refactoring, you should still see the two buttons with **`Popularity`** and **`Trending`** and the font lists received upon initial rendering:

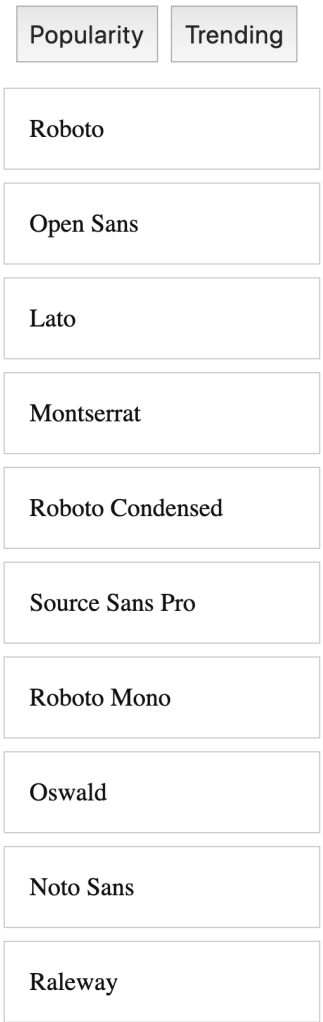

Figure 16.9: Output from refactoring

We now have a good understanding of how we can fetch data using the **`useEffect`** hook on the follow-up rendering and can then avoid the infinite loop issue. For now, the hooks work really well, making our code simpler and easier to understand. However, with the custom hook, we can make our code more reusable so that it can be shared with many other components. This will help make our code much easier to work on.

MORE REFACTORING WITH CUSTOM HOOK

In the previous sections, we learned how to use hooks to fetch data in a function component. We also discussed how to avoid the infinite loop by specifying the second argument in **`useEffect`**.

In this section, we will learn how to further refactor the previous code by using a custom hook. While developing React applications, it is always a good idea to reuse code in one place and share it across different components.

As we discussed in the previous chapter, and also specified in the React documentation (https://packt.live/3cwLUjN), *Building your own hooks lets you extract component logic into reusable functions,* which means hooks allow us to easily share and reuse code.

Let's recap how to create the custom hook. To create a custom hook, we first create a new file. The filename convention is to start with use, followed by the hook name—for example, **`useResponse.js`**.

Inside the hook file, we need to import necessary hooks, such as **`useState`** or **`useEffect`**, and create a function, such as **`useResponse`**. To fetch data in the custom hooks, we do almost the same as in the previous section, but we will return the response we fetched this time. And, finally, we need to export the hook so that other components can use it:

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

const useResponse = (propValue) => {
  const [res, setRes] = useState([]);

  useEffect(() => {
  const getRes = async () => {
  const res = await axios.get(
    ‹https://mydomain.com/api›
  );

  setRes(res);
  }

  getRes();
  }, [propValue]);
```

```
  return res;
};

export default useResponse;
```

For other components to use this hook in order to retrieve the data from the server, we import the hook first, and then call the hook and assign it to a variable:

```
import useResponse from './useResponse';
const response = useResponse(propValue);
```

EXERCISE 16.04: REFACTORING A FONTLIST COMPONENT WITH A CUSTOM HOOK

In this exercise, we are going to create a custom hook called **`useFonts`** and move the **`useEffect`** part to the **`useFonts`** hook. By doing so, we can reuse the **`useFonts`** custom hook in other components and improve code readability. At the end of this exercise, you will learn how to create a custom hook and use it in any other component.

Let's recap what we did in *Exercise 16.03, Refactoring the FontList Component*

`FontList.js`

```
1  import React, { useState, useEffect } from 'react';
2  import axios from 'axios';
3  const API_KEY = '12345';
4  const FontList = props => {
5  const [fonts, setFonts] = useState([]);
6  useEffect(() => {
7  const getFonts = async () => {
8  // 1. Fetch data with axios
9  const res = await axios.get(
10     `https://www.googleapis.com/webfonts/v1/webfonts?key=${API_KEY}&sort=${prop
       s.sort}`
  );

16  // 2. Update state
```

`The complete code can be found here:` https://packt.live/3dGTDvR

1. First, create a new file called **`useFonts.js`** in the same directory as **`FontList.js`**.

2. Import **useState** and **useEffect**.

   As we are only going to create a function and return a value and will not be making use of React objects such as **React.createElement** or **React.Component**, we do not need to import React from **'react'**. As we are going to use **axios**, we need to import **axios**:

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

3. Create a function called **useFonts**, and then move **API_KEY** and **useEffect** to **useFonts**. Then, export **useFonts**:

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

const API_KEY = '12345';

const useFonts = () => {
  useEffect(() => {
  const getFonts = async () => {
  // 1. Fetch data with axios
  const res = await axios.get(
    `https://www.googleapis.com/webfonts/v1/webfonts?key=${API_
KEY}&sort=${p
    rops.sort}`
  );

  // 2. Update state
  setFonts(res.data.items.slice(0, 10));
  };

  getFonts();

  }, [props.sort]);
}

export default useFonts;
```

   We will receive **props.sort** as **sort**.

4. Add **`sort`** as a function parameter and replace **`props.sort`** with **`sort`**. Define **`useState`** to store the fetched data in the state:

```
const useFonts = sort => {
  const [fonts, setFonts] = useState([]);

  useEffect(() => {
  const getFonts = async () => {
  // 1. Fetch data with axios
  const res = await axios.get(
    `https://www.googleapis.com/webfonts/v1/webfonts?key=${API_
KEY}&sort=${so
    rt}`
  );

  // 2. Update state
  setFonts(res.data.items.slice(0, 10));
  };

  getFonts();

  }, [sort]);
}

...
```

5. Return the **`fonts`** state:

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

const API_KEY = '12345';

const useFonts = sort => {
  const [fonts, setFonts] = useState([]);

  useEffect(() => {
  const getFonts = async () => {
  // 1. Fetch data with axios
  const res = await axios.get(
    `https://www.googleapis.com/webfonts/v1/webfonts?key=${API_
KEY}&sort=${s
```

```
      ort}`
    );

    // 2. Update state
    setFonts(res.data.items.slice(0, 10));
    };

    getFonts();
    }, [sort]);

    return fonts;
};

export default useFonts;
```

6. In **`FontList.js`**, we no longer need to import **`useState`**, **`useEffect`**, and **`axios`**. Instead, import **`useFonts`**. Finally, assign the font data from **`useFonts`** to a variable called **`fonts`**:

```
import React from 'react';
import useFonts from './useFonts';

const FontList = props => {
  const fonts = useFonts(props.sort);

  return (
  <div className="card">
  {fonts.map((font, index) => (
    <a
    href={`ttps://fonts.google.com/specimen/${font.family.replace(
    ' ',
    '+'
    )}`}
    className="card__item"
    key={index}
    >
    {font.family}
    </a>
  ))}
  </div>
  );
```

```
};

export default FontList;
```

With the custom hook, we should see output as in the following screenshot, with two buttons and a font list upon initial rendering. When you click on the button, the list of fonts should get updated, either by **`popularity`** or **`trending`**:

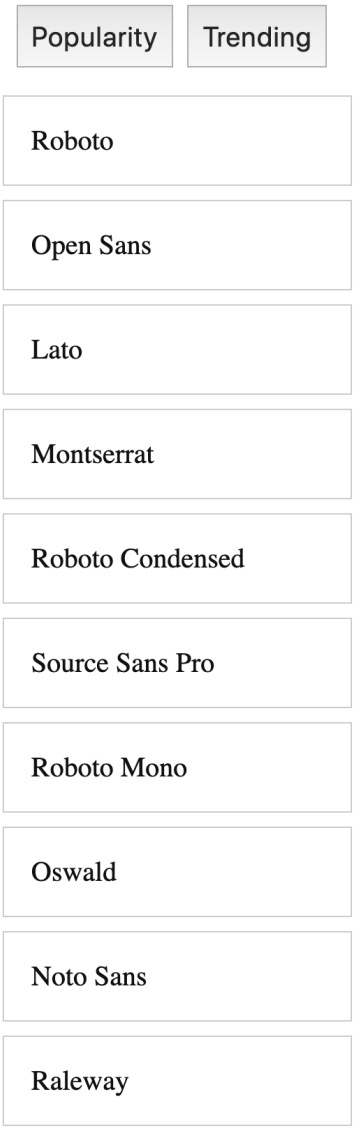

Figure 16.10: Output with a custom hook

We have learned how to fetch data both in the initial and follow-up renders in class-based components. We also discussed how to use hooks to achieve the same results in function components. Furthermore, we learned how to refactor using the custom hook so that we can make the code more reusable by utilizing other components.

ACTIVITY 16.01: CREATING AN APP USING POTTER API

The aim of this activity is to create an app displaying characters from the Harry Potter API. The list of characters can be re-fetched by clicking on one of four buttons that represent each house (**`Gryffindor`**, **`Slytherin`**, **`Hufflepuff`**, and **`Ravenclaw`**). Initially, the app will display the name of the characters, but upon clicking on the name, it will further fetch the details of the characters, such as the role and the house. To fetch the data, we are going to use the Harry Potter API (https://www.potterapi.com/), which provides the list of houses and characters in the Harry Potter franchise.

The complete app should look like this:

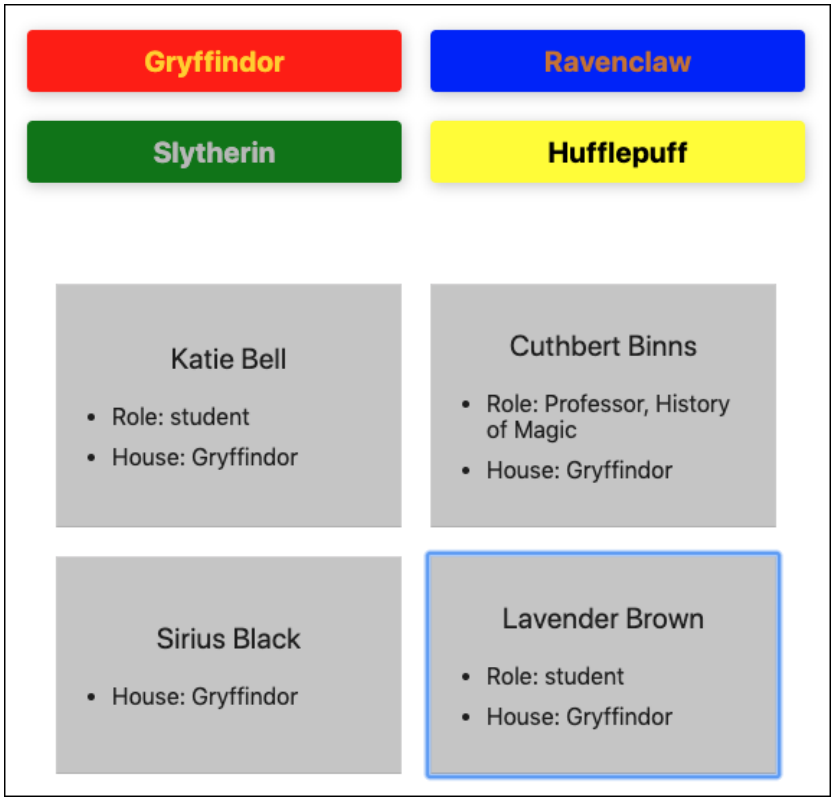

Figure 16.11: Outcome of the activity

To use the Potter API, we need to sign up and get the API key first. To sign up, go to the sign-up page (https://www.potterapi.com/login/). Once signed up, you will be taken to the page where you can get the API key.

Once you are registered, head to the documentation page (https://www.potterapi.com/#introduction), where you can get all the necessary details of the Harry Potter API. For this activity, you need to take a closer look at the house routes and character routes. If you need to get the data for an individual character, add **`characterId`** to the character route's endpoint. To fetch the data, use **`async`**/**`await`** with axios. As we will fetch the data using the same base endpoint (https://www.potterapi.com/v1/), please make use of the custom hook so that we can share the hook to fetch data from different endpoints.
The recommended file structure is as follows:

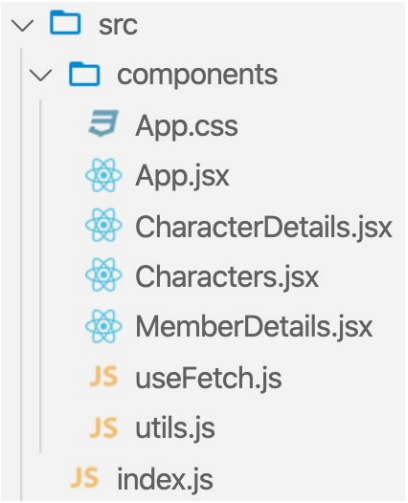

Figure 16.12: The file structure of the activity

For the styles, please use the following code, **App.css**:

App.css

```
.container {
margin: 100px auto;
width: 600px;
}

.house {
display: flex;
flex-wrap: wrap;
justify-content: center;
 }

 .house__button {
 border: 0;
 border-radius: 4px;
 box-shadow: 1px 2px 9px #ccc;
```

The complete code can be found at: https://packt.live/2Z58gF4

The following steps will help you to complete the activity:

1. Start a new React application
2. Import the styles using the **App.css** file provided above.
3. Add four buttons, one labeled for each house (**Gryffindor**, **Slytherin**, **Hufflepuff** and **Ravenclaw**).
4. Use the **Potter API** endpoint to fetch the data pertaining to the houses.
5. Fetch data using **async/await** in the **useEffect** hook.
6. Save the data in a variable (let's call this **houses**) defining the state.
7. Display the buttons with the four house names. Each button should have a key and a name.
8. Apply the color names with the color value. Check whether the color is **scarlet** or **bronze**. Create a utility file called **utils.js** and put the **colorconverter** function inside it.
9. Create a custom hook called **useFetch** to fetch the data from the Potter API. Create a separate file called **useFetch.js** to put the code of the custom hook inside it.

10. Display the data fetched from the API.

> **NOTE**
>
> The solution of this activity can be found on page 738.

SUMMARY

This chapter talked about how to fetch data on initial rendering and how to avoid the infinite loop when updating the state after fetching data. Throughout this chapter, we have used the **`Google Font API`** to fetch data on initial rendering with the class-based and functional component. We have also covered how to use custom hooks to share the common fetching functionality with components.

Firstly, we learned that the **`componentDidMount()`** life cycle method is the best place to fetch data upon initial rendering. Also, we learned how to use async/await with **`componentDidMount()`** and how to display the data using the **`map()`** method in the **`render()`** method.

Secondly, we used **`componentDidUpdate()`** when fetching data on updating components. We also established that updating state inside **`componentDidUpdate()`** will cause the infinite loop. To avoid this, we compared the previous prop with the current prop and updated the state only when those two prop values were different.

Thirdly, we learned how to use React hooks (such as **`useState`** and **`useEffect`**) to fetch data on initial rendering in the function components. Furthermore, we also learned how to use **`async/await`** with **`axios`** in **`useEffect`** and how to avoid the infinite loop when using this hook.

Lastly, we learned how to further refactor fetching data using a custom hook. Using the custom hook allows other engineers to easily reuse the custom hook by sharing the custom hook with components such as fetching data from servers.

Throughout the previous and current chapter, we learned all about fetching data from servers, and you should now have a great understanding of how to fetch data in React. Knowing this will help you build more advanced React applications, which could be used to communicate with the outside world. In the next chapter, we are going to learn how to use refs (references) in React.

17

REFS IN REACT

OVERVIEW

This chapter will introduce you to how to use references in React. You will be able to apply the knowledge gained from the chapter to implement different ways of applying references. By the end of this chapter, you will be able to use Refs in your existing React application in an effective way.

INTRODUCTION

In the previous chapters, we learned about the various methods we can use to fetch data from external APIs and how to include that data inside your React application. This process required you to send XMLHttpRequest(XHR) requests over the network, take the result, and store it in your component.

A web page is often composed of more than just a React app; for example, an application can be embedded in a website where there are various elements in a web page (including our React app) all working together, like different form elements or HTML components. With all these elements that live outside your React components, you might think: *could we access these other elements that are not controlled by the React application itself?*

This question leads to the following section. We actually can control these elements "living outside" of a React application through References(Refs).

WHY REACT REFS?

In React, we use Refs to access, manipulate, and interact with the DOM directly. Direct references to HTML elements allow us to perform tasks that are commonly encountered when creating client-side applications. Those tasks include, but are not limited to:

- Handling native events on DOM elements, such as focus and hover
- Measuring DOM element dimensions inside the browser directly
- Locking scrolling on view containers outside the React app

In this chapter, you will learn about the usage of Refs, different methods for how and where to apply them, and finally, how to abstract and hide their implementation details.

In order to understand the purpose of using React Refs, we have to first understand what DOM elements are and the meaning of references in relation to them.

REFERENCES

For JavaScript to interact with and manipulate a DOM element, it needs a reference to that element. This reference is simply an object representation of a DOM element. Through the properties of the reference, we can read and write to the element's attributes, which, in turn, define its appearance and behavior in the browser.

You might wonder why we would want to manipulate a DOM node directly if we are already creating these using React in the first place. Refs can and should be used, when either the element that we wish to handle was created outside our app's scope or when the React element's properties do not suffice to achieve a specific goal.

Let's take a look at one of the scenarios that cannot be achieved with the properties provided by the React element as the first exercise of this chapter. In the following exercise, we will look into creating custom upload buttons.

EXERCISE 17.01: CREATING CUSTOM UPLOAD BUTTONS WITH REFS

In this exercise, we will use a Ref to create a styled input button, which could not be styled using plain CSS. We will first create the file input and then add a custom button component that will trigger the file upload functionality of the file input when it is clicked. To do that, let's perform the following steps:

1. Create a new React app using the **`create-react-app`** CLI:

```
$ npx create-react-app upload-button
```

2. Move into the new React applications folder and start the **`create-react-app`** development server:

```
$ cd upload-button/ && npm start
```

3. Inside the **`App.js`** file, change the **`App`** component to only return a plain **`input`** component:

```
import React from "react";
class App extends React.Component {
 render() {
 return <input />;
 }
}
export default App;
```

4. To make the input trigger the browser's file upload functionality, add the file type to the **`input`** component:

```
import React from "react";
class App extends React.Component {
  render() {
    return <input type="file" />;
  }
}
export default App;
```

When we run the application at **`http://localhost:3000`**, we should see the following output:

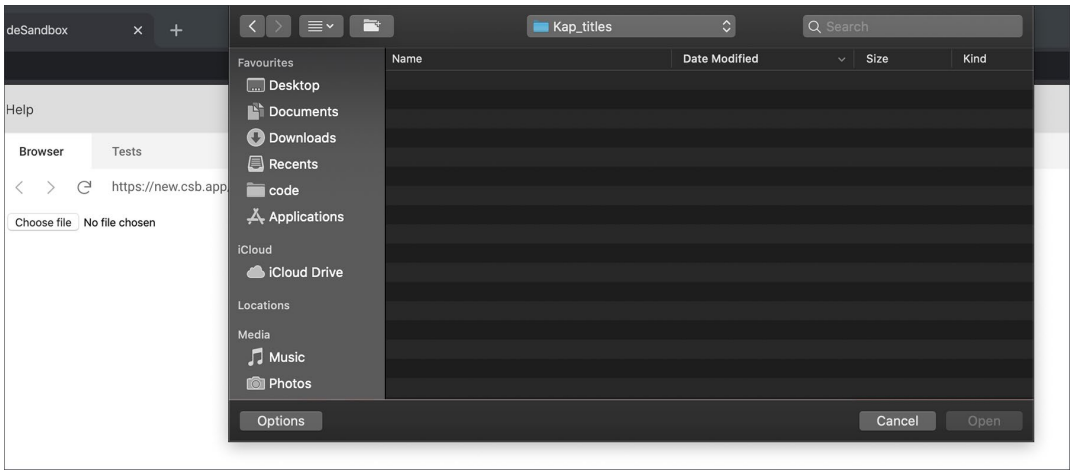

Figure 17.1: App output

If you click this input button, you should be prompted by the browser to select a file to upload.

5. Below the **`input`** component, add a **`button`** component and add the styles for a gray background and white font color:

```
class App extends React.Component {
  render() {
  return (
    <div>
    <input type="file" />
    <button style={{ backgroundColor: "gray", color: "white" }}>
```

```
        upload document
      </button>
     </div>
    );
    }
}
```

Note that you have to wrap the input and button components inside a div element, because each React component is required to have a single root sub-component. This div element is only used to display the buttons in the view layer.

6. Pass it the **`hidden`** attribute in order to hide the native input file element:

```
class App extends React.Component {
  render() {
    return (
    <div>
    <input type="file" hidden={true} />
    <button style={{ backgroundColor: "gray", color: "white" }}>
      upload document
    </button>
   </div>
  );
  }
}
```

To make the button component trigger the upload prompt, you have to create a Ref to the input component and then trigger the native click function on this reference.

7. Start by creating a class field called **`inputRef`** to store the Ref:

```
class App extends React.Component {
  inputRef;
  render() {
  return (
    <div>
    <input type="file" hidden={true} />
    <button style={{ backgroundColor: "gray", color: "white" }}>
      upload document
```

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

8. Create an inline function that takes a parameter called **`refParam`** and assign this parameter to the **`inputRef`** class field of the **`App`** component. Pass this function to the input component using the Ref prop:

```
class App extends React.Component {
  inputRef;
  render() {
  return (
    <div>
    <input
      ref={refParam => this.inputRef = refParam}
      type="file" hidden={true}
    />
    <button style={{ backgroundColor: "gray", color: "white" }}>
      upload document
    </button>
    </div>
  );
  }
}
```

9. Lastly, add an **`onClick`** handler to the button component. This **`onClick`** handler should call the **`click`** function on the **`inputRef`** class field reference to the input component:

```
class App extends React.Component {
  inputRef;

  render() {
  return (
    <div>
    <input
      ref={refParam => this.inputRef = refParam}
      type="file" hidden={true}
     />
    <button
```

```
          onClick={() => this.inputRef.click()}
          style={{ backgroundColor: "gray", color: "white" }}>
          upload document
        </button>
      </div>
    );
    }
}
```

The output is as follows:

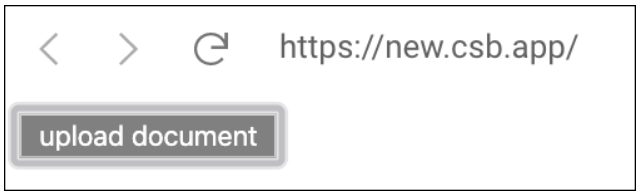

Figure 17.2: The upload button shown on the app

Now, click the button component and you will be prompted with the file upload window, because the custom button now triggers the file input's native functionality.

As you can observe, you have utilized a React Ref to bind the functionality of a native input that is hardly customizable to a button component that you can style as you wish. Now, after this first success, we should have a closer look at what you have just done. In particular, we will inspect the function that was passed as the ref object to the input component.

Now that we have practiced using Refs in a hands-on exercise, let's take a look at the various ways of creating Refs.

WAYS OF CREATING REACT REFS

There are thee approaches to creating React Refs:

- Using a callback function

  The first approach is used on class-based components where you can save the reference to a class field. In the previous exercise, we followed this approach when we created a Ref by passing a callback function as the reference to an input element:

```
<input ref={refParam => this.inputRef = refParam} />
```

This works because the ref object is automatically called by React when rendering and passes a reference to the created DOM element back into this function. We then persisted this reference into a class field called **`inputRef`**, so that it is accessible everywhere in the component's instance.

- Creating a Ref using the **`React.createRef`** function

  As it turns out, the second way of creating a Ref also relies on class-based components. Unlike the callback type, here, the ref object is not only passed as an argument to the callback function but can also be initialized beforehand and can be populated as soon as React renders the component. To implement this manner, React provides a utility function called **`createRef`**. This method creates a reference that we can assign to a class field and pass it to the component we want to access though the ref. The following code snippet shows how a ref is created:

```
class App extends React.Component {
inputRef = React.createRef();
render() {
  return (
    <input ref={this.inputRef} />
  );
  }
}
```

As we can see from the preceding code, the return value of the **`React.createRef`** function is an object with a single key called current. This current key, when initialized, is either empty or contains the values that were passed to the **`createRef`** function. The value of the current key will be replaced with the reference to the selected DOM element.

This would require us to change the code from the previous exercise to not calling the native click function on the Ref directly, but rather on the **`Ref.current`** object, as shown in the following code:

```
<button
  onClick={() => this.inputRef.current.click()}
  style={{ backgroundColor: "gray", color: "white" }}
  >
  upload document
</button>
```

- Initializing a Ref using the **`React.useRef`** hook

  The last approach to creating React Refs is by using a React hook and can therefore only be used with functional components.

  In our existing code example, we could restructure our class-based component to a function and leverage the **`useRef`** hook to achieve the same result of a button triggering another DOM element's functionality through a reference:

```
const App = () => {
  const inputRef = React.useRef();
  return(
  <div>
    <input ref={inputRef} type="file" hidden={true} />
    <button
    onClick={() => inputRef.current.click()}
    style={{ backgroundColor: "gray", color: "white" }}>
      upload document
    </button>
    </div>
  )
}
```

  The result of applying this **`useRef`** hook is very similar to the **`createRef`** utility function. That is, the return value is the same, namely, an object with only one key called current. And the value of current is either null or the value that was passed as a parameter to the **`useRef`** hook. Yet there is one key difference that makes the **`useRef`** hook more suitable for functional components; **`useRef`**, as opposed to **`createRef`**, preserves the reference across renders. This means that even if the function rendering a component is executed multiple times, the **`useRef`** hook always returns the same Ref prop that was created initially, regardless of how often the hook was triggered. All we did in the preceding code snippet was change the class syntax to a function syntax and use the **`React.useRef`** hook instead of the React. **`createRef`** function to initialize our ref.

Since we have explored the three ways to create references to DOM elements in our applications, we should go on and practice each of the approaches in an exercise.

We can use the class-based components and **`createRef`** using a very common use case for accessing HTML elements: to measure them, their height, and width.

EXERCISE 17.02: MEASURING THE DIMENSIONS OF A DIV ELEMENT IN A CLASS-BASED COMPONENT

In this exercise, we will create a merely visual div element and read its dimensions by creating a reference to this DOM node. Knowing the dimensions, such as the **`height`**, **`width`**, or **`offset`** of an element, comes in handy when this cannot be done using the properties, we access on React components. Therefore, it's necessary to access the DOM elements directly using references to them.

1. Create a new React app using the **`create-react-app`** CLI:

```
$ npx create-react-app custom-button
```

2. Move into the new React applications folder and start the **`create-react-app`** development server:

```
$ cd custom-button/ && npm start
```

3. Inside the **`App.js`** file, change the App component to only return an empty **`div`** element:

```
import React from "react";
class App extends React.Component {
  render() {
    return <div />;
  }
}
export default App;
```

4. To visualize the **`div`**, add a **`height`**, **`width`**, and **`border`** to the **`div`** style:

```
import React from "react";
class App extends React.Component {
  render() {
  return <div
    style={{ width: 80, height: 20, border: "4px solid black" }}
  />;
  }
}
export default App;
```

You end up with a black-bordered box on your screen.

5. To access the dimensions, you need to create a reference using the **`createRef`** function and assign it to a class field called **`divRef`**:

```
import React from "react";
class App extends React.Component {
  divRef = React.createRef();

  render() {
  return <div
    style={{ width: 80, height: 20, border: "4px solid black" }}
    />;
  }
}
export default App;
```

6. To populate the empty **`divRef`** element, you have to pass the reference to the **`div`** element as the **`Ref`** prop:

```
import React from "react";
class App extends React.Component {
  divRef = React.createRef();

  render() {
  return <div
    ref={this.divRef}
    style={{ width: 80, height: 20, border: "4px solid black" }}
    />;
  }
}
export default App;
```

7. Lastly, when clicking **`div`**, this should print the result of the **`getBoundingClientRect`** function on the current reference. This printed object holds the dimensions of the **`div`** element:

```
import React from "react";
class App extends React.Component {
  divRef = React.createRef();

  render() {
  return <div
    ref={this.divRef}
```

```
        onClick={() => console.log(
          this.divRef.current.getBoundingClientRect()
        )}
        style={{ width: 80, height: 20, border: "4px solid black" }}
      />;
      }
    }
    export default App;
```

If you click on the **`div`** element, you will see a log to your browser console that looks similar to this one:

```
DOMRect {x: 8, y: 8, width: 88, height: 28, top: 8, …}
x: 8y: 8
width: 88h
eight: 28
top: 8
right: 96
bottom: 36
left: 8
```

These are the dimensions of the element in the browser. You might wonder why the height is **`28`** pixels instead of the **`20`** pixels that you defined in *step 4*. This happened because the actual height dimension is the sum of the element height and the height of the borders, which is **`4`** pixels each for the top and bottom borders. The same is true for all the other dimensions as well.

In this exercise, we used a reference to an element and measured its size and offsets, which otherwise cannot be done without accessing the DOM directly. For this exercise, we applied a class-based component. However, in this component, there is no state to be managed. Therefore, we could also use a functional component and reduce the slight code overhead caused by the class syntax. Let's change the structure in the next exercise.

EXERCISE 17.03: MEASURING THE ELEMENT SIZE IN A FUNCTIONAL COMPONENT

This exercise will start where the previous exercise left off. Here, you will create a Ref in a functional component and use it to get the dimensions of a DOM element (information that is typically only accessible through the DOM directly). This will demonstrate how we can measure the dimensions of an element using a functional component. Therefore, you will not need to create yet another React application but can continue with the last output of *Exercise 17.02*, *Measuring the Dimensions of a div Element in a Class-Based Component*. The ref will give us access to the DOM element for the component, which is what we need to access the size of the rendered element in the browser:

1. Delete the existing **`App`** component and create a new **`App`** component as a function that returns an empty **`div`**:

```
import React from 'react'
const App = () => {
  return <div />
}
export default App;
```

2. Next, add the same styling to the **`div`** component as in the previous exercise:

```
const App = () => {
  return <div
    style={{ height: 20, width: 80, border: "4px solid black" }}
  />
}
```

3. Create a reference using the **`useRef`** hook, assign the return value to a variable called **`divRef`**, and pass this variable to div as the Ref prop:

```
const App = () => {
  const divRef = React.useRef();

  return(
    <div
      ref={divRef}
      style={{ height: 20, width: 80, border: "4px solid black" }}
    />)
}
```

4. Once again, pass an **`onClick`** handler, which displays the result of the **`getBoundingClientRect`** method of the current **`divRef`** reference:

```
const App = () => {
  const divRef = React.useRef();
  return (
    <div
    ref={divRef}
    onClick={() => console.log(
    divRef.current.getBoundingClientRect())}
    style={{ height: 20, width: 80, border: "4px solid black" }}
    />)
}
```

When clicking the div element, the exact same dimensions as at the end of *Exercise 17.02*, *Measuring the Dimensions of a div Element in a Class-Based Component* will be displayed:

```
DOMRect {x: 8, y: 8, width: 88, height: 28, top: 8, …}
x: 8
y: 8
width: 88
height: 28
top: 8
right: 96
bottom: 36
left: 8
```

So far, we have seen how to create and pass Ref when the element that we want to interact with via a reference is created within the render method of the component that also uses this Ref. But what if we wanted to access a reference of a component that is wrapped by a reusable higher-order component (HOC) and this HOC simply adds functionality to its children components by caring about Ref objects? In such a case, we would not want to change the HOC in such a way that it handles the implementation detail of manipulating the reference.

In order to preserve the encapsulation of components, we can use a proxying technique referred to as Forwarding Refs.

FORWARDING REFS

In this section, we are going to discuss the methodologies used in order to implement *Forwarding Refs*. These are as follows:

Composition

One of the key concepts of React is the composition of components. That means we want to be able to encapsulate logic without creating dependencies between components.

Let's look at the following code snippet, in which the **`SayHello`** component adds a button that says hello to literally any child component that you pass to it. This way, the child component only needs to know about what it is supposed to know, in the same way the **`SayHello`** component only cares about the logic and the UI responsible for saying hello:

```
import React from "react";
const SayHello = props => {
  return (
  <div>
    { props.children }
    <button onClick={() => console.log("Hello!")}>
      Say Hello!
    </button>
  </div>
  )
}
const App = () => {
  return (
    <SayHello>
      <p>I wish I could say hello...</p>
    </SayHello>
  )
}
export default App;
```

Following this example, we could replace the paragraph wrapped by the **`SayHello`** component with anything else that we want.

However, if we want to access the button component that is created in the **`SayHello`** component from outside, we can't do that because the components are composed.

Luckily, React provides a utility function to get hold of the reference inside another component as well. This utility is named **`React.forwardRef`** and lets us pass a reference through a proxy to a child component that we wish to interact with.

To proxy a Ref to the **`SayHello`** button, we need to first wrap the functional component inside the **`React.forwardRef`** utility, as shown in the following code:

```
const SayHello = React.forwardRef(props => {
  return (
  <div>
    { props.children }
    <button onClick={() => console.log("Hello!")}>
    Say Hello!
    </button>
  </div>
  )
})
```

Unlike all other functional components, **`forwardRef`** not only takes a single props parameter, but also a second parameter, which obviously holds the value of the reference we want to pass. We will call this secondary parameter **`forwardedRef`**, and pass it to the button component inside the render method of the **`SayHello`** component:

```
const SayHello = React.forwardRef((props, forwardedRef) => {
  return (
  <div>
    { props.children }
    <button
      ref={forwardedRef}
      onClick={() => console.log("Hello!")}
    >
    Say Hello!
    </button>
  </div>
  )
})
```

This way, we would be able to access the **`Say Hello`** button from the **`App`** component. With this forwarded reference, we could mimic trigger the **`Say Hello`** button from another button in the **`App`** component, as shown in the following code:

```
const SayHello = React.forwardRef((props, forwardedRef) => {
  return (
  <div>
    { props.children }
    <button
    ref={forwardedRef}
    onClick={() => console.log("Hello!")}
    >
    Say Hello!
    </button>
  </div>
  )
})

const App = () => {
  const buttonRef = React.useRef();

  return (
  <SayHello ref={buttonRef}>
    <p>I wish I could say hello...</p>
    <button
    onClick={() => buttonRef.current.click()}
    >
    Also say hello!
    </button>
  </SayHello>
  )}
```

The output is as follows:

Figure 17.3: The Sayhello component

As you can see from the above figure, forwarding the Ref from the **`App`** component is no different to passing the Ref as a prop, as we have already done in the preceding sections. We can now trigger the **`onClick`** handler of the **`Say Hello!`** button by clicking the button. It's even possible to pass the reference down multiple levels through multiple **`React.forwardRef`** functions.

Let's grab the opportunity to practice the forwarding of Ref objects. To do so, we will refer back to the previous exercise, where we measured the dimensions of an element; only this time, we will proxy the reference through another component.

EXERCISE 17.04: MEASURING BUTTON DIMENSIONS USING A FORWARDED REF

In this exercise, we will create a custom component that encapsulates the styles of a button component. We will measure its dimensions from outside the custom component using **`useRef`** and **`forwardRef`**. Let's perform the following steps:

1. Create a new React app using the create-react-app CLI:

```
$ npx create-react-app proxy-styles
```

2. Move into the new React applications folder and start the **`create-react-app development`** server:

```
$ cd proxy-styles/ && npm start
```

3. Inside the **`App.js`** file, delete the **`App`** component and create a new empty **`App`** component and also another empty component called **`StyledButton`**:

```
import React from "react";
const StyledButton = props => {
  return null;
};
const App = () => {
  return null;
};
export default App;
```

4. Adapt the **`StyledButton`** component to return a native button component and pass the **`StyledButton`** children received as props inside:

```
import React from "react";
const StyledButton = props => {
  return <button>{props.children}</button>;
};
const App = () => {
  return null;
};
export default App;
```

5. Inside the **`App`** component, use **`StyledButton`** to wrap any text you want, something like This is the button text:

```
import React from "react";

const StyledButton = props => {
  return <button>{props.children}</button>;
};
const App = () => {
  return <StyledButton>This is the button text</StyledButton>;
};
export default App;
```

6. To really have a styled button, you have to pass styles to the **`button`** component inside **`StyledButton`**:

```
import React from "react";
const StyledButton = props => {
  return (
  <button
  style={{ backgroundColor: "grey", color: "white" }}
  >
    {props.children}
  </button>
  );

};
const App = () => {
  return <StyledButton>This is the button text</StyledButton>;
};
export default App;
```

7. When opening your **App** component in the browser, you should see your styled button. Next, in order to display the button's dimensions, you need to first turn the **StyledButton** component from a functional component to a **forwardRef** call that takes the functional component as a parameter:

```
const StyledButton = React.forwardRef((props, forwardedRef) => {
  return (
  <button style={{ backgroundColor: "grey", color: "white" }}>
    {props.children}
  </button>
  );
});
```

8. Also, remember to add the second parameter to the functional component. Call the **Ref** parameter **forwardedRef** and pass it to the **button** component

```
const StyledButton = React.forwardRef((props, forwardedRef) => {
  return (
  <button
    ref={forwardedRef}
    style={{ backgroundColor: "grey", color: "white" }}
  >
    {props.children}
  </button>
  );
});
```

9. Back in the **App** component, create a Ref using the **useRef** hook, assign the return value to a variable named **buttonRef**, and pass this variable to the **StyledButton** component as a Ref prop:

```
const App = () => {
  const buttonRef = React.useRef();

  return (
  <StyledButton ref={buttonRef}>
  This is the button text
  </StyledButton>
  );
};
```

10. Finally, use the **`useEffect`** hook inside the **`App`** component to log the **`getBoundingClientRect`** method of the current reference, **`buttonRef`**:

```
const App = () => {
  const buttonRef = React.useRef();

  React.useEffect(() => {
  console.log(buttonRef.current.getBoundingClientRect());
  });

  return <StyledButton ref={buttonRef}>This is the button
  text</StyledButton>;
};
```

Simply running this code in your browser will display the following **`DOMRect`** object in the console:

```
DOMRect
x: 8
y: 9
width: 124.359375
height: 18
top: 9
right: 132.359375
bottom: 27
left: 8
```

This might be familiar to you, since this a similar output to what you saw in the previous exercises earlier. Specifically, these are the dimensions of the **`button`** element inside the **`StyledButton`** component. You got hold of these measurements by proxying a reference from the App component to the **`StyledButton`** component using the **`React.forwardRef`** function.

We will now put the knowledge we have acquired from this chapter to the test and will try our hands at solving the following activity.

ACTIVITY 17.01: CREATING A FORM WITH AN AUTOFOCUS INPUT ELEMENT

Your task will be to create a user-date form that automatically focuses the first input field. You will begin by accessing plain input fields via React Refs. From there, you will move on to create a custom input component that can forward the reference to an input field. In the end, you will encapsulate the logic to auto focus and handle the ref by creating a controllable form component.

The following steps will help you to complete the activity:

1. Create a new React app using the create-react-app CLI.
2. Inside the **`App.js`** file, change the App component to only return a plain **`form`** component.
3. Inside the **`form`** component, add three input fields: one for the first name, one for the last name, and a third one for the email.
4. Create a reference for an input field as a class field and pass the reference to the first input field. On **`componentDidMount`**, trigger the **`focus`** function of the current **`Ref`**.
5. Create a new function component called **`FocusableInput`**, which can forward a reference. Make this component simply return **`null`**.
6. Now, instead of **`null`**, the **`FocusableInput`** component should now return an **`input`** component that is of the **`text`** type. This input takes the **`placeholder`** prop passed to **`FocusableInput`**. The input also gets passed the **`forwardedRef`** parameter as a Ref prop.
7. The first input field within the App component should now be replaced with an instance of the **`FocusableInput`** component. This **`FocusableInput`** should get passed both the reference and the placeholder that were passed to the original input field.

   Hint: You could replace all native input fields in the **`App`** component with the **`FocusableInput`** component. Just make sure not to pass the same Ref to all **`FocusableInput`** components. By the way, the Ref prop on the **`FocusableInput`** component is optional; you don't have to pass it.
8. Create a new functional component called **`FocusableForm`**. This component takes over all of the logic from the **`App`** component's **`render`** function.

9. The **App** component now only returns **FocusableForm**. Also, the **componentDidMount** function and the **inputRef** class field are no longer needed in the App component.
10. In order to access the input field in the **FocusableForm** component, you have to create a new Ref and pass it to the first input field.
11. Now, use the **useEffect** hook to trigger a focus on the current reference, **inputRef**.
12. Make the **FocusableForm** component controllable by passing an **autoFocus** prop from the App component to **FocusableForm** and using the prop in the **useEffect** hook to decide whether to focus the reference.

    The final output is as follows:

Figure 17.4: Final output of the app

Following this activity, you have a controllable component that hides all of the imperative code to handle the focus of an input component depending on whether or not a specific prop is passed to this component. This activity is a perfect use case on how to create clean logic to access DOM elements directly using Refs and even proxy them to abstract the complexity even further.

> **NOTE**
>
> The solution of this activity can be found on page 755..

SUMMARY

In this chapter, we have covered a really interesting part of React. The reality is that Refs are not something you are regularly going to encounter unless you are responsible for creating component libraries or working with more design-focused elements.

It is an interesting part of React because, as you might have noticed, Refs do not really fit into React's declarative style. Every time we use Refs, we use them to tell the browser how to do things instead of what we want to see. While not necessarily a problem, it can get complicated and we need to be clear about our intent when using Refs.

The best we can do is to hide most of what we are doing with Refs and ensure that anyone using our code only interacts with our components via props. If we need to access details more closely related to how the browser renders the component (such as focus, scrolling, or dimensions), that will require Refs because React can't handle it without them, but ultimately, the consumer of our code should not be aware of how we achieved these things. In the next chapter, we will look at Refs in more detail.

18

PRACTICAL USE CASES OF REFS

OVERVIEW

This chapter will introduce you to some use cases of Refs and how to leverage their functionalities in our code. You will be able to identify the scenarios in which to use Refs so that you can manipulate DOM elements directly. You will be able to combine Refs with other React functionalities, such as **`createPortal`** and **`cloneElement`**, and use them effectively.

INTRODUCTION

In the previous chapter, we learned about the functionalities of React Refs, and we discussed a few scenarios where refs could be used. In this chapter, we will continue to explore other such practical use cases and look at situations when we might want to use refs. A few such scenarios include binding external libraries to React, triggering animations, handling DOM actions, such as focusing on a text field, text selection, and media playback. Imagine a scenario where you have designed a React application that requires the integration of the functionalities of the Google Maps library. Refs in React make the process easier since we can access the DOM nodes of this library with the help of Refs. Since Refs are often best used in co-operation with other React utilities, we will take a brief look at a couple of such utilities, too, in particular, **`cloneElement`** and **`createPortal`**. During this chapter, we will get to know how to make use of Refs in our code and solve some problems we regularly encounter while building React applications.

RECAP OF REACT REFS BASICS

Before we dive right into handling Refs, we should recap some basics regarding Refs and React in general. We will be refreshing our memory on the following topics:

- Native versus custom React components
- Encapsulation via props

This will support us in grasping the usage of Refs as a whole. Furthermore, this will help us in utilizing the knowledge acquired to solve problems related to DOM manipulations efficiently.

Native versus Custom React Components

As you might recall, there are two different types of JSX tags that we frequently write. These are the **custom components** that we entirely implement ourselves; for example, a specialized input field that has custom properties and is composed of other components. As a standard convention, we represent them as a capitalized JSX variable, as shown in the following code snippet:

```
// JSX custom Components
<MyCustomInput />
<SpecialButton />
```

The other type of components are elements that are native to the browser, like any HTML elements outside the scope of the React application, known as **native components**. These, in contrast to the convention we defined previously, are described as lowercase variables in JSX, as shown in the following code snippet:

```
// JSX native HTML tags
<input />
<button />
```

To clarify further, let's take a look at an example from the previous chapter. We used a ref and the **`useEffect`** hook to create a custom input field that automatically receives focus without user interaction. For this, we used the following code snippet previously:

```
import React from "react";
const AutoFocusInput = () => {
  const inputRef = React.useRef(undefined);
  React.useEffect(() => {
    inputRef.current.focus();
  }, []);
  return <input ref={inputRef} />;
};
const App = () => {
  return <AutoFocusInput />;
};
export default App;
```

As we can see from the preceding code, when we apply the **`native-custom-distinction`**, we end up with a **custom** component, **`AutoFocusInput`**, that internally uses the **native** tag, **`input`**. During the render, while turning our React components into proper DOM elements that the browser can understand and display, React dissembles all of the custom components into their native HTML representations. Now, this process and the result of it usually aren't of interest to developers unless we want access to the DOM elements' properties to manipulate them, just like we did in the preceding example using Refs.

Custom components become especially powerful when we separate concerns and abstract logic from each other. The common *React way* to do so is by letting data flow by using *Props*. Let's recapitulate how the data flows through props in the next section.

ENCAPSULATION VIA PROPS

Props let us hide implementation details and the complexity of a component while still keeping them controllable from the outside, such as from the component's parent. The parent in this case would want to decide whether its child should be in a specific state. Naturally, the same idea applies to the elements that manage sub-components via Refs.

In the following example, which is an extension of the previous one, observe how to encapsulate the logic to manipulate the focus of the input field by passing **`AutoFocusInput`** as props. This is shown in the following code:

```
import React from "react";
const AutoFocusInput = props => {
  const inputRef = React.useRef(undefined);
  React.useEffect(() => {
    if (props.autoFocus) {
      inputRef.current.focus();
    }
  }, []);
  return <input ref={inputRef} />;
};
const App = () => {
  return <AutoFocusInput autoFocus={true} />;
};
export default App;
```

Now, in order to create a controllable version of the **`AutoFocusInput`** component, we need to pass the props from the parent component, **`App`**. Inside the component itself, we should only apply the autofocus when the **`autoFocus`** prop is **`true`** or the input field has focus.

This way, any React component that chooses to include our **`AutoFocusInput`** input field can control it via the **`autoFocus`** prop and it can do so without knowing nor caring about how to manipulate the native input's focus attribute.

Maybe you have already noticed it, but in observing the preceding steps, we concealed an *imperative* action and made it look like a *declarative* one — something we discussed in the previous chapter, *Chapter 17*, *Refs in React*.

After taking care of hiding all the complexity from the outside world, we should now focus on the inner implementation of the components that handle Refs to manipulate DOM elements in the next sections.

DOM MANIPULATION HELPERS

As mentioned beforehand, there are two functions in particular that we usually use together with Refs to gain full access and manipulate the DOM even outside our React applications' scope. These utilities are **`createPortal`** and **`cloneElement`**. The former is provided by the React DOM package and the latter comes bundled with React itself.

THE CLONEELEMENT FUNCTION IN REACT

Whenever we want to change a given React component's immutable attributes – for example, its passed props – we can fall back to **`React.cloneElement`** and create a copy of this particular component and change it as we wish.

The function's signature is very similar to the fundamental element of React **`React.createElement`**. However, instead of passing **`type`** as the first parameter, we pass an **`element`**, such as a component. The return value is the same as returned by **`createElement`** which is basically a React element. **`cloneElement`** can be defined as:

```
React.cloneElement(element, [props], […children]): React Element
```

A typical use case for cloning an element is when we want to enhance children with specific properties. In the following example, we are cloning the children passed to the **`RedClones`** component and giving them different props (a *red background color* in this case). Let's take a look at the following code snippet:

```
const RedClones = props => {
  const newProps = {
  style: {
      backgroundColor: "red"
    }
  };
  const redChildrenClones = React.Children.map(props.children, child =>
    React.cloneElement(child, newProps)
  );
  return redChildrenClones;
};
const App = () => {
  return (
    <RedClones>
      <div>A Box</div>
```

```
      <p>Another Box</p>
    </RedClones>
  );
};
```

As we can see from the preceding code, firstly, we have defined **`RedClones`** as a parent component and, using **`cloneElement`**, we have created the cloned children components called **`redChildrenClones`**. We have passed the **`backgroundColor`** attribute as **`red`** to these children components.

Cloning the children gives us the ability to add **any props to any number of child components of any type** without knowing what kind of component the child is. This, in turn, leads to very clean and reusable logic for adding specific props when the children are wrapped by the *cloner component*. In our case, this means that we can set a red background without the children knowing that they should have a colored background. The output looks like this:

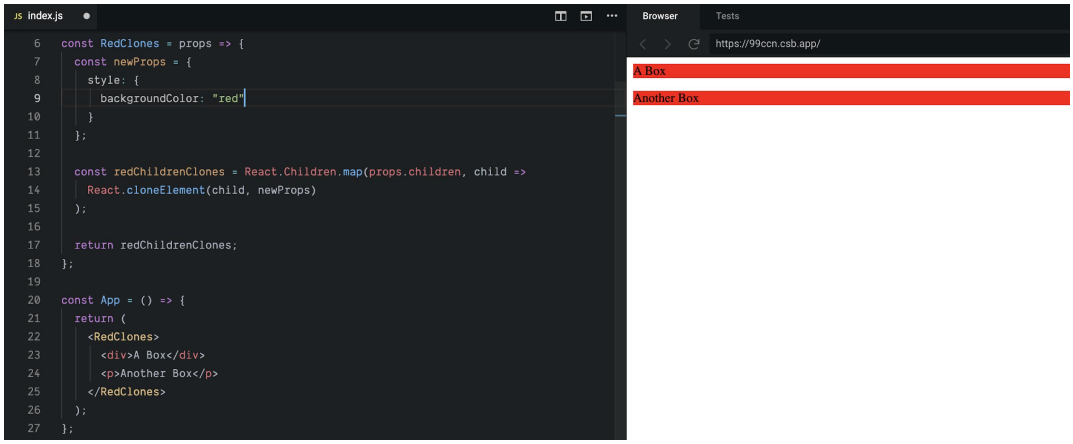

Figure 18.1: The result of the cloning children example

To consolidate our newly acquired knowledge in terms of adding arbitrary props via the **`cloneElement`** method, we will now practice its usage in an exercise.

EXERCISE 18.01: CLONING AN ELEMENT AND PASSING IT AN ONCLICK PROP

In this exercise, we will practice how to pass a cloned element to an **`onclick`** event handler as a prop. We will create two event handlers and attach the property. Let's go through the following steps to do so:

1. Create a new React app using the **`create-react-app`** CLI:

```
$ npx create-react-app click-handler
```

2. Move into the new React applications folder and start the **`create-react-app`** development server:

```
$ cd click-handler/ && npm start
```

3. Create an **`App`** component that returns **`null`**. Also, create **`WithClickHandler`**, which does nothing apart from pass its children:

```
const WithClickHandler = props => {
  return props.children
}
const App = () => {
  return null;
};
```

4. Let the **`App`** component return the **`WithClickHandler`** component with two native input components of the *button* type. One of the inputs should have the text **`first Button`**, and the other one should have the text **`second Button`**:

```
const WithClickHandler = props => {
  return props.children
}
const App = () => {
  return (
    <WithClickHandler>
      <input type="button" value="first Button" />
      <input type="button" value="second Button" />
    </WithClickHandler>
  );
};
```

5. Now, enhance the children with an **`onClick`** handler. To do so, create a new object, **`clickProps`**, with an **`onClick`** property and pass it an empty function. Then, clone each of the children using the **`cloneElement`** function and pass the new props to the clone:

```
const WithClickHandler = props => {
  const clickProps = {
    onClick: () => {}
  };
  const clickableChildren = React.Children.map(
    props.children,
    child => React.cloneElement(child, clickProps)
  );
  return clickableChildren;
}
const App = () => {
  return (
    <WithClickHandler>
      <input type="button" value="first Button" />
      <input type="button" value="second Button" />
    </WithClickHandler>
  );
};
```

6. Finally, make the **`onClick`** function print the input's text, in other words, its value. For this to happen, you have to change the function to accept the click event as a parameter and then select the event target's value by accessing the **`event.target.value`** property:

```
const WithClickHandler = props => {
  const clickProps = {
    onClick: event => {
      console.log("you clicked the", event.target.value)
    }
  };
  const clickableChildren = React.Children.map(
    props.children,
    child => React.cloneElement(child, clickProps)
  );

  return clickableChildren;
```

```
}
const App = () => {
  return (
    <WithClickHandler>
      <input type="button" value="first Button" />
      <input type="button" value="second Button" />
    </WithClickHandler>
  );
};
```

If you run the preceding code on the browser, the output is as follows:

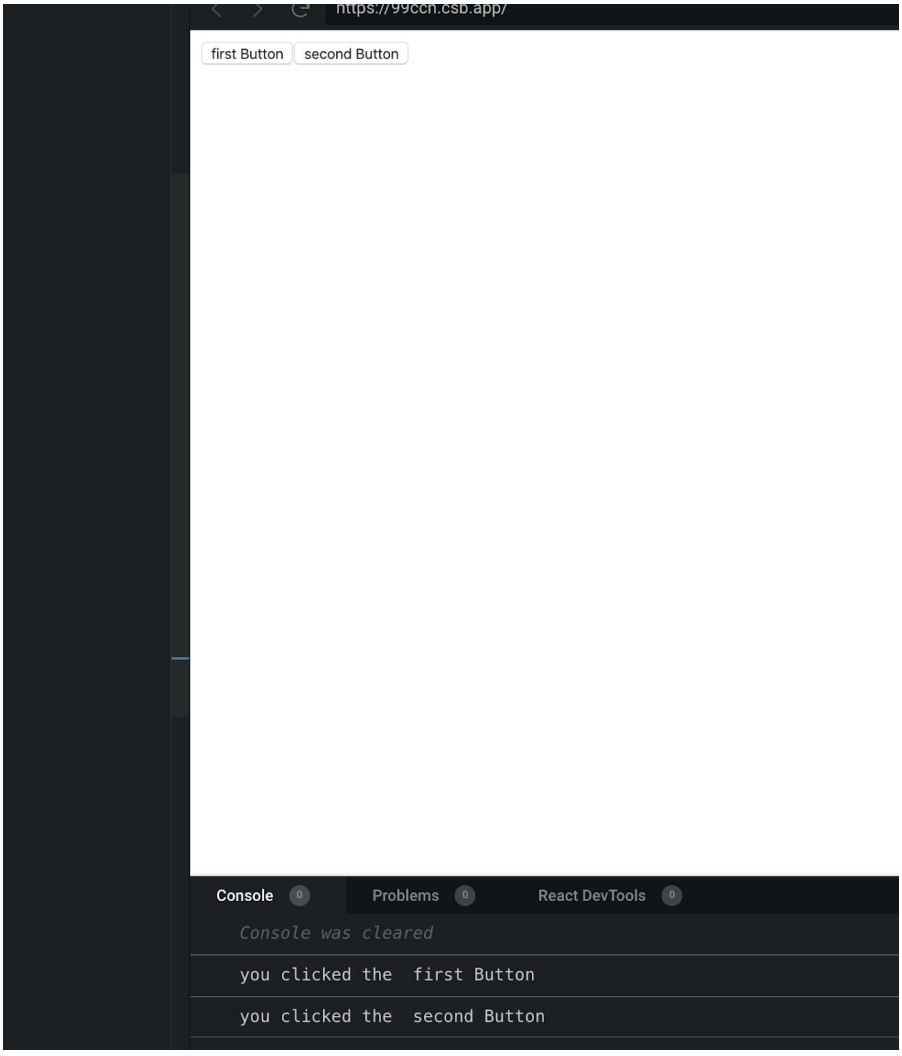

Figure 18.2: App output

If you followed along, you should see two logs in your browser's console: one indicating that you clicked the first button, and the other one confirming that you clicked the second button as well.

In this exercise, you built a highly variable and reusable higher-order component that lets you append props to any number and any kind of children. You just made **`cloneElement`** solve the big problem of reusability for you, with ease.

Before combining **`cloneElement`** with React Refs, you should learn about yet another functionality that often helps to leverage the full potential of Refs – **`createPortal`**.

THE CREATEPORTAL FUNCTION IN REACTDOM

This is an excerpt from the official React documentation: "Portals provide a first-class way to render children into a DOM node that exists outside the DOM hierarchy of the parent component". Let's explore what this means with the help of an example.

Creating a portal requires a child component, which has to be a React Element, and a container, which is simply an element, that is, either an HTML element or a React Element. What gets returned is a specialized type of React Element called a **`React Portal`**:

```
ReactDOM.createPortal(child, container): React Portal
```

> **NOTE**
>
> **`createPortal`** is a function of the **`ReactDOM`** package, not the React package.

Here, you will find a demonstration of the usage of portals. We are going to mount our React application onto a **`div`** container with the ID root, just like we normally do. Additionally, we will create a portal into another **`div`** container that has nothing to do with our root mounting point.

For this example, we need to change our standard index.html file and add another **`div`** class inside. To make the portal more apparent, we give the new **`div`** element a red background and an ID of **`portalContainer`**. Let's take a look at the following HTML file, where the **`id`** of the **`div`** element contains the **`portalContainer`** element:

```
// index.html
<!DOCTYPE html>
<html lang="en">
  <body>
    <div id="root"></div>
    <div id="portalContainer" style="background-color: red;"></div>
  </body>
</html>
```

Our React component will look as follows:

```
import React from "react";
import ReactDOM from "react-dom";
const Portal = () => {
  const portalSelector = document.querySelector("#portalContainer")
  const portalChild = <span>Hello</span>;
return ReactDOM.createPortal(portalChild, portalSelector)
}
const App = () => {
  return (
    <div>
      <Portal />
      <p>World</p>
    </div>
  )
};
const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

The **`Portal`** component selects the **`portalContainer`** using a query selector, creates a small custom React element, and then passes both to the **`createPortal`** function. As usual, the **`App`** component includes our custom element and a paragraph.

Look at the code and keep in mind that we put **`Portal`** before the paragraph. We assume that the order of the rendered components was congruent with their JSX representation. This means that **`Hello`** (*rendered inside* **`Portal`**) should be above the **`World`** paragraphs.

You can run this code in your browser and confirm our assumption:

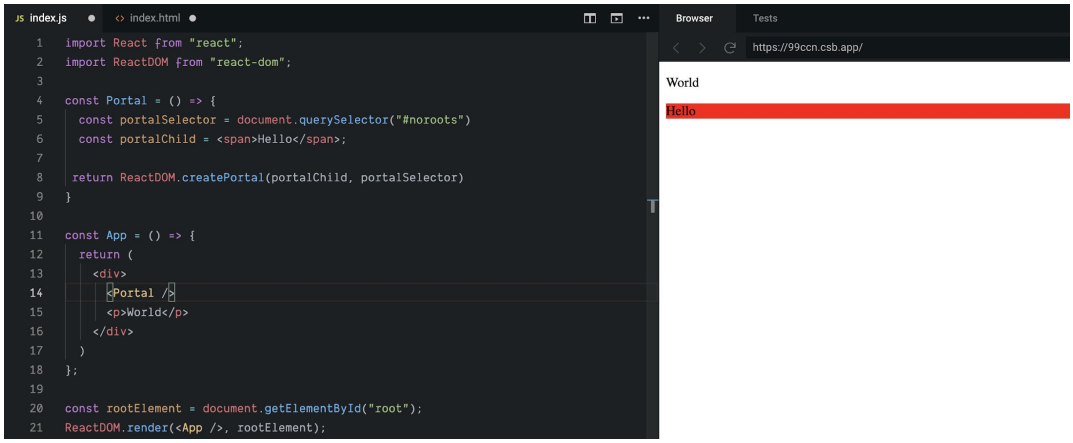

Figure 18.3: App screenshot

Unexpectedly, the rendered output does not match our assumption regarding the order of the elements. This behavior is because the portal lives outside our React application; it was mounted onto the **`portalContainer`**, which, in the HTML markup, is located **below** the root mounting point of our **`App`** component.

This is the great power of React portals (and a big caveat as well). We can *beam* components defined inside our application to an outer scope, which means that portals enable us to create, for example, overlays, modals, or tooltips that are supposed to be placed in a location that is not directly accessible from our application. Similar use cases arise when working with composable frontends (also referred to as micro-frontends).

In the next section, let's look at an interesting exercise regarding the creation of a global overlay that will contain a greeting message and will be displayed on top of the web page.

EXERCISE 18.02: CREATING A GLOBAL OVERLAY USING PORTALS

In this exercise, we will create a global overlay containing a salutation to the user of our application. This salutation component will also disable all the scrolling so that the user can focus on the overlay and not get distracted by scrolling elements in the background.

1. Create a new React app using the **`create-react-app`** CLI:

```
$ npx create-react-app global-overlay
```

2. Move into the new React applications folder and start the **`create-react-app`** development server:

```
$ cd global-overlay/ && npm start
```

3. Create an **`App`** component that includes a **`div`** element with a bunch of placeholder text:

```
import React from "react";
import ReactDOM from "react-dom";

const App = () => {
  return (
    <div>
      <p>Lorem ipsum...</p>
    </div>
  )
};
```

4. Add two more components, called **`Overlay`** and **`Salutation`**. Inside the **`Overlay`** component, create a variable that holds the return value of a query selector, selecting the DOM body element. Additionally, make the **`Salutation`** component return a **`div`** element with some saluting words inside it:

```
const Salutation = () => {
  return <div style={{backgroundColor:'white', width: "22vw", height:
"10vh", display: "flex", justifyContent: "center", alignItems:
"center", borderRadius:4}}>Welcome to the React Workshop!</div>
}
const Overlay = () => {
  const bodySelector = document.querySelector("body");
}
const App = () => {
  return (
```

```
      <div>
        <p>Lorem ipsum...</p>
      </div>
    )
};
```

Now, spawn a saluting portal inside the body element that you stored in the variable called **`bodySelector`**.

5. To achieve this, you need to call the **`createPortal`** function with the container element (the **`body`** element) and the **`Salutation`** component. Don't forget to include the **`Overlay`** component in the **`App`** component:

```
const Salutation = () => {
  return <div style={{backgroundColor:'white', width: "22vw", height:
"10vh", display: "flex", justifyContent: "center", alignItems:
"center", borderRadius:4}}>Welcome to the React Workshop!</div>;
}
const Overlay = () => {
  const bodySelector = document.querySelector("body");

  return ReactDOM.createPortal(<Salutation />, bodySelector);
}
const App = () => {
  return (
    <div>
      <Overlay />
      <p>Lorem ipsum...</p>
    </div>
  );
};
```

Now, you have created a portal and are displaying the salutation to the user, but it is not an overlay yet. Creating an overlay requires you to add styling, which will cover the entire viewport, and will center the salutation.

6. For simplicity, create a new **`OverlayContainer`** component that you can apply the styles to and wrap the **`Salutation`** component inside:

```
const Overlay = () => {
  const bodySelector = document.querySelector("body");
  bodySelector.style.overflowY = "hidden";

  const OverlayContainer = (
```

```
      <div style={overlayStyles}>
        <Salutation />
      </div>
    );
    return ReactDOM.createPortal(OverlayContainer, bodySelector);
};
```

The last step is to disable the scrolling on the document. For demonstration purposes, you first need to make the **App** component larger than the current viewport, otherwise there would be no scrolling in any event.

7. Give the outermost **div** element a height of twice the viewport **height** parameter:

```
const overlayStyles = {
  width: "100vw",
  height: "100vh",
  top: 0,
  left: 0,
  position: "fixed",
  display: "flex",
  alignItems: "center",
  justifyContent: "center",
  backgroundColor: "rgba(0,0,0,.7)"
};
const Salutation = () => {
  return <div style={{backgroundColor:'white', width: "22vw", height:
"10vh", display: "flex", justifyContent: "center", alignItems:
"center", borderRadius:4}}>Welcome to the React Workshop!</div>;
};
const App = () => {
  return (
    <div style={{ height: "200vh" }}>
      <Overlay />
      <p>Lorem ipsum...</p>
    </div>
  );
};
```

If you tried to scroll, you would have observed that scrolling up and down is possible.

8. To prevent scrolling, we have to set the body's overflow to **`hidden`**. We already have access to the **`body`** element via our **`bodySelector`** variable:

`App.js`

```
16 const Salutation = () => {
17  return <div style={{backgroundColor:'white', width: "22vw", height:
"10vh", display: "flex", justifyContent: "center", alignItems: "center",
borderRadius:4}}>Welcome to the React Workshop!</div>;
18 };
19 const Overlay = () => {
20  const bodySelector = document.querySelector("body");
21  bodySelector.style.overflowY = "hidden";
22
23  const OverlayContainer = (
24    <div style={overlayStyles}>
25      <Salutation />
26    </div>
27  );
28  return ReactDOM.createPortal(OverlayContainer, bodySelector);
```

`The complete code can be found here:` https://packt.live/3d8ncpm

You should end up with a simple, yet powerful, overlay, which disables all scrolling on the body and displays a centered salutation.

The output is as follows:

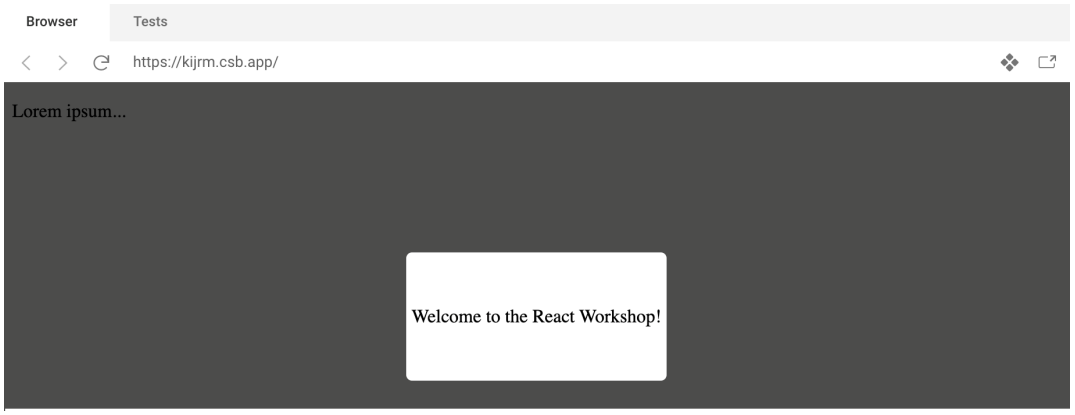

Figure 18.4: Final app output

Finishing this section, you have collected all the relevant information and skills to effectively solve this chapter's activity and use Refs in more realistic use cases. You will be able to do so in a professional manner using **`cloneElement`** and **`createPortal`**.

Let's put everything that we have learned so far in the chapter together and get started with an activity.

ACTIVITY 18.01: PORTABLE MODALS USING REFS

The aim of this activity is to create a modal that covers the entire viewport and disables scrolling within it. The viewport, however, is not going to be a hardcoded DOM body element but can be passed as a Ref to the overlay modal. The overlaying component should be mounted into the Ref using a React portal.

Before you begin, make sure you start with a new **`create-react-app`** boilerplate application.

Here is the **`.css`** file that you can work with for this activity:

`App.css`

```
1 body {
2  height: 100vh;
3 }
4 .App {
5  height: 200vh;
6  position: absolute;
7  }
8 .Viewport {
9  height: 50vh;
10  width: 100vw;
11  position: relative;
12  overflow-y: scroll;
```

`The complete code can be found here:` https://packt.live/2LuvxIK

The following steps will help you complete the activity:

1. Create a class-based **`App`** component, initialized with a **`viewPortRef`** Ref and a state that includes a **`showModal`** property that defaults to false.
2. Create another class-based component called **`ModalOverlay`**. This component receives three props: **`onCloseHandler`**, **`showModal`**, and **`mountingPoint`**. Create yet another functional component called **`Modal`**. The **`Modal`** component includes a **`div`** element with the **`Modal`** CSS class. This **`div`** container simply wraps the **`props.children`** components.

3. Inside the **`ModalOverlay`** component, create a local component called **`ModalPortal`**, which contains a **`div`** with the **`ModalOverlay`** CSS class. This **`div`** container wraps the Modal component. Inside Modal, map the children of the **`ModalOverlay`** component, clone each of the children, and pass an **`onClick`** prop that gets assigned the **`this.props.onCloseHandler`** parameter.
4. Create a portal where **`ModalPortal`** is the element that should be mounted and the **`props.mountingPoint`** variable of **`ModalOverlay`** is the container for the portal. Return this portal from the **`ModalOverlay`** component's **`render`** method if **`props.showModal`** is **`true`**, otherwise, return **`null`**.
5. On the **`App`** component, create a class method called **`toggleModal`**. This function should set **`state.showModal`** to its opposite.
6. For the **`render`** method of **`App`**, you can use the following code snippet:

```
render() {
  return (
    <div className="App">
      <button /> { /* open button */ }
      <div className="Viewport" />
      <ModalOverlay>
        { /* close button */ }
      </ModalOverlay>
    </div>
  );
}
```

7. To prevent errors, when there is no mounting point passed to **`ModalOverlay`**, you should select the DOM body element and use this as fallback instead.

The final output should look like this:

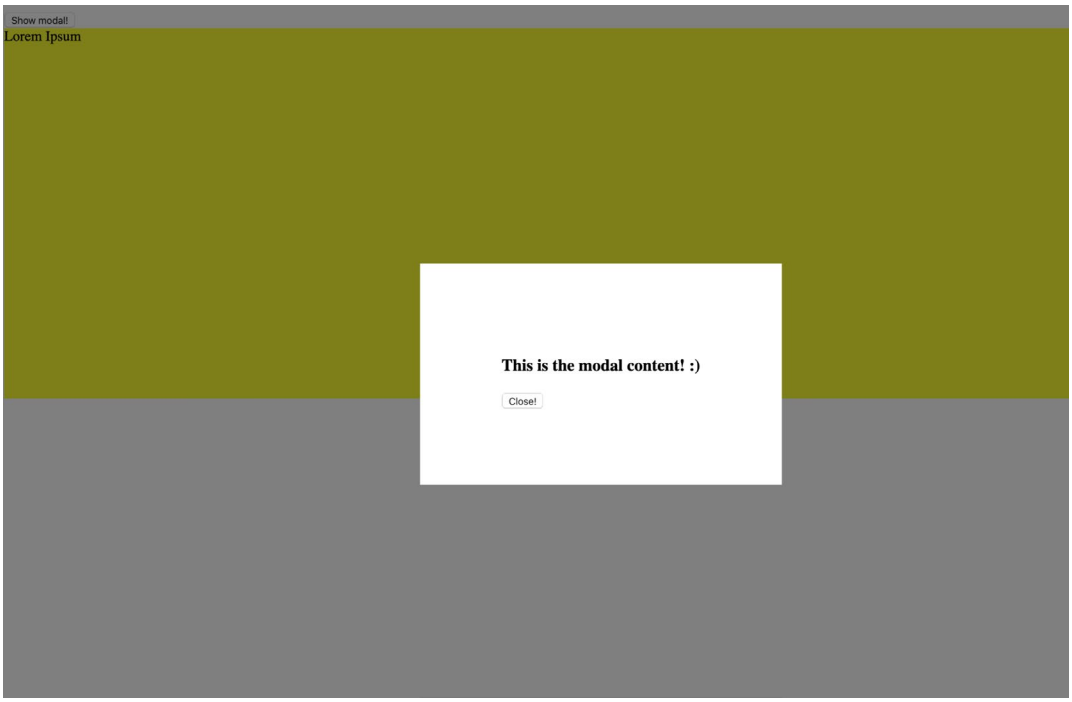

Figure 18.5: Final App

In both cases, you should get back to the default state when you click the close button (or even the text, since we passed it an **`onClick`** handler as well). From here, you can go on and experiment with some further features. Perhaps you could even combine the outcome of this activity with the ones from the previous chapters.

> **NOTE**
>
> The solution of this activity can be found on page 766.

SUMMARY

In the last chapter of *The React Workshop*, we started by taking a look at some previously discussed fundamentals about Refs and React in general. Concepts such as encapsulation via props and the different ways to use Refs led to you finally gaining traction with Refs. From there, you went on to not only use Refs for practical use cases but also explored **`cloneElement`** and **`createPortal`**, which are often encountered when applying Refs in React source code.

Afterward, you explored more sophisticated concepts of React when you leveraged a higher-order component to enhance child components by cloning and adding custom props to them. On top of that, you beamed components from your React application to an outer scope using React portals. Finally, you put all the pieces together and implemented a modal that encapsulates all the logic and that is controllable via props. The modal is so flexible and reusable that it can be put anywhere in the source code and be passed a mounting point either inside or outside the actual React application.

With that, this marks the end of this book. We have spent a lot of time and gone over everything you need in your repertoire to be able to deliver high-quality React applications that incorporate all of the latest standards and techniques. We started off simple and explored the techniques to build React components with JSX and discussed the different techniques to create your own custom JSX components with React, whether through classes or functions. From there, we explored how to make your components interact with your browser and how to handle those events appropriately. We also explored the different ways to attach conditions to your React components and display multiple components based on data and state and explored how to interact with React components at each stage of a component's life cycle.

From there, we transitioned into talking more about data flows through a React application and the different types of components, as each type of component has its own preferred methods of handling data passing (whether via state, props, or contexts). We then moved on to talk about different techniques and libraries to supplement your React applications, diving into some of the most common libraries, such as React Router, to handle how to compose multiple React components into complicated applications.

We then moved on to more modern and advanced React techniques, specifically focusing on the newest React feature: Hooks. We explored the functionalities of using React Hooks before diving into the more advanced usage of Hooks, including at the level of writing our own custom Hooks to make our applications simpler, cleaner, and more reusable.

From there, we turned our attention toward one of the most common (and trickiest) portions of writing any web application with React: communicating with the outside world. Making API requests is difficult and requires a lot of special knowledge and the ability to handle a lot of different states and cases, and we focused on how to use libraries such as Axios and functionality native to JavaScript, such as **`async`**/**`await`**, to create fully-featured API-driven React components.

We closed out with how to properly attach our React components to the DOM and interact with what is displayed to the user via the browser directly through another React feature: Refs. With this, we were able to close the loop on React, interacting with the outside world, interacting with the browser, and interacting with components internally.

All of this adds up to a full library of React knowledge; everything we need to be able to build professional React applications that scale well, are clean and easy to maintain, and are the kind of applications that you would be proud to share with your peers. All that is left now is for you to go out there and build the React application of your dreams.

APPENDIX

CHAPTER 1: GETTING STARTED WITH REACT

ACTIVITY 1.01: DESIGN A REACT COMPONENT

Solution:

1. Verify that Node.js and Create React App are installed on your machine:

```
$ node --version
```

Output:

```
V12.6.0
$ npx create-react-app --version
```

Output:

```
2.1.1
```

2. Create your project, called **`buystuff`**, via the Create React App CLI:

```
$ npx create-react-app buystuff
```

3. Delete all the unnecessary files for our project.

Delete **`App.css`**, delete the contents of **`App.js`** except for the import and export statements, and delete **`logo.svg`**.

4. Build the **`App`** React component as a class component but leave it blank.

5. In **`App.js`**, change the import statement at the top:

```
import React, {Component} from "react";
```

6. Then add the following component definition:

```
class App extends Component {
  render() {
    return <div className="App"/>;
  }
}
```

7. Build the **`Header`** React component as a functional component inside **`App.js`**. Its only prop should be **`title`**, which contains the store's name.

```
const Header = (props) => <h1>{props.title}</h1>;
```

8. Build the **`InventoryItem`** React component as a functional component inside **`App.js`**. It should contain props that consist of the item name and price.

```
const InventoryItem = (props) => (
  <div className="InventoryItem>
    <strong>{props.itemName}</strong>
    <hr />
    <p>{props.itemPrice}</p>
  </div>
);
```

9. In **`App.js`**, change the **`App`** component to have a constructor and a starting state with two items in it and include your **`InventoryItem`** component in the **`render()`** function twice and your **`Header`** component once.

```
class App extends Component {
  constructor(props) {
    super(props);

    this.state = {
      items: [
        { itemName: "Shoe", itemPrice: 5 },
        { itemName: "Sock", itemPrice: 3 }
      ]
    };
  }
  render() {
    return (
      <div className="App">
        <Header title="BuyStuff" />
        <InventoryItem
          itemName={this.state.items[0].itemName}
          itemPrice={this.state.items[0].itemPrice}
        />
        <InventoryItem
          itemName={this.state.items[1].itemName}
          itemPrice={this.state.items[1].itemPrice} />
      </div>
```

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

The result should give us a React application that looks like this:

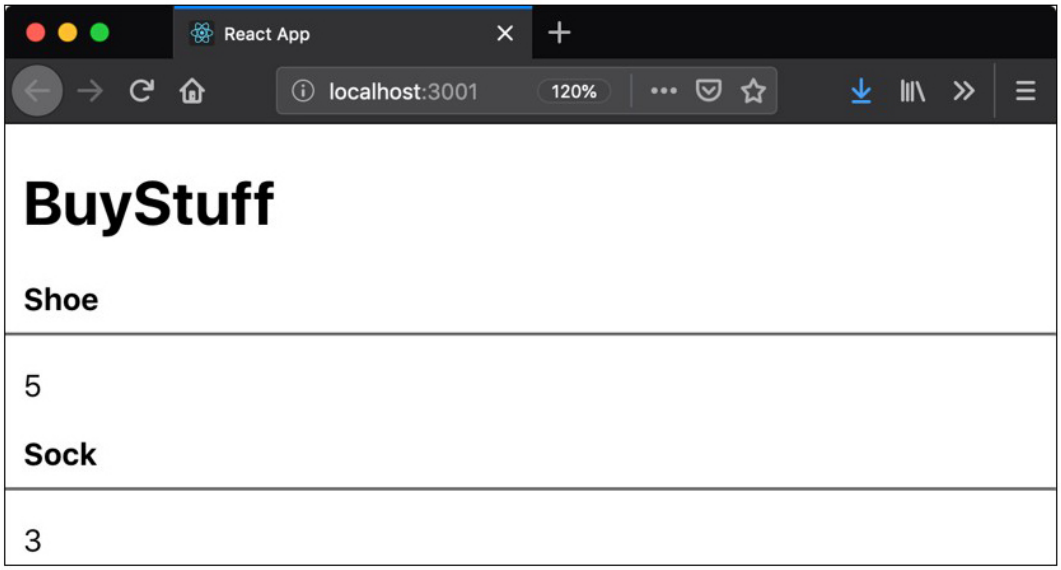

Figure 1.12: BuyStuff React component

CHAPTER 2: DEALING WITH REACT EVENTS

ACTIVITY 2.01: CREATE A BLOG POST USING REACT EVENT HANDLERS

Solution:

1. Create your project, called **`fieldlength`**, via the Create React App CLI, and then start up the application with Yarn:

```
$ npx create-react-app fieldlength
$ cd fieldlength
```

2. Delete all the unnecessary files for our project.
3. Delete **`App.css`**, delete the contents of **`App.js`** except for the import of **`react`** and the export default statements, and delete **`logo.svg`**.
4. In src/App.js, build the App React component as a class component, but leave it blank:

```
import React, { Component } from 'react';

class App extends Component {
render() {
  return (
  <div className="App">
  </div>
  );
}
}
export default App;
```

5. Give the component an initial state with a single element in the state; this will store the input from the **`textarea`**.
6. Still in App.js, add the following code:

```
  constructor(props) {
  super(props);
  this.state = { field: "" };
  }
```

7. Add a **`textarea`** to the component. We are still in App.js, update the render() code:

```
render() {
  return (
  <div className="App">
  <textarea cols="80" rows="25"></textarea>
  </div>
);
}
```

8. Add a function that will act as the event handler. This function will need to accept an event as an argument and should update the state of the component by setting the input from the **`textarea`**. Still in App.js, add the following code:

```
  updateFieldLength(event) {
  const field = event.target.value;
  this.setState({ field });
  }
```

9. And then change the render function to this:

```
  render() {
  return (
    <div className="App">
      <textarea cols="80" rows="25"
      onChange={this.updateFieldLength.bind(this)}></textarea>
    </div>
  );
  }
```

10. Add a function that will return **`N`** characters, wrapped inside JSX, where **`N`** is the length of the input in the **`textarea`**:

```
  renderFieldLength() {
    return <p>{`${this.state.field.length} character(s)!`}</p>
  }
```

11. Add the preceding function to the display of your component. Still in App.js, update the render() function one last time:

```
  render() {
  return (
    <div className="App">
      <textarea cols="80" rows="25"
```

```
        onChange={this.updateFieldLength.bind(this)}></textarea>
        <br />
        {this.renderFieldLength()}
      </div>
    );
    }
```

12. Verify that as you type text into the **`textarea`**, the display is updated.

    The code for App.js will look like this:

```
import React, { Component } from 'react';

class App extends Component {
  constructor(props) {
  super(props);
  this.state = { field: "" };
  }
  renderFieldLength() {
  return <p>{`${this.state.field.length} character(s).`}</p>
  }
  updateFieldLength(event) {
  const field = event.target.value;
  this.setState({ field });
  }
  render() {
  return (
    <div className="App">
      <textarea cols="80" rows="25"
     onChange={this.updateFieldLength.bind(this)}></textarea>
      <br />
      {this.renderFieldLength()}
    </div>
  );
  }
}

export default App;
```

This should leave us with a final UI that resembles the following:

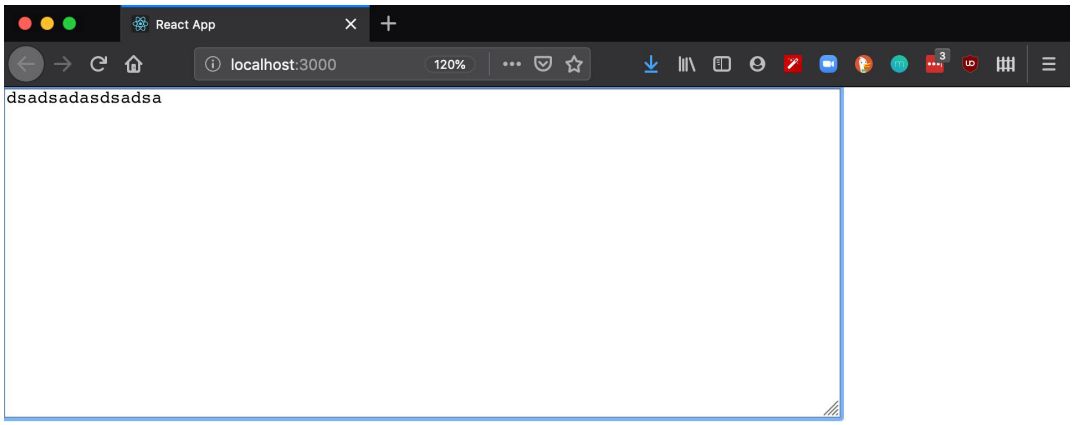

Figure 2.15: The final UI

13. Now we will expand this app a little more to add a **`submit`** button and make it look more like a blog post editor. Above our **`textarea`**, we will add an **`H2`** header titling our app as **`Blog Post Writer`**, add a horizontal rule tag (**`hr`**), and add a bold bit that says to write the post here. You will want to include some instructions to the user that the post must be at least **`100`** characters. We will also want to add a button below the **`textarea`** for submitting the post:

    Here's our new render() function:

```
render() {
  return (
    <div className="App">
      <h2>Blog Post Writer</h2>
      <hr />
      <strong>Write your post here</strong><br />
      <small>Must be at least 100 characters.</small><br />
      <textarea cols="80" rows="25"
    onChange={this.updateFieldLength.bind(this)}></textarea>
      <br />
      {this.renderFieldLength()}
      <br />
      <button>Submit Post</button>
    </div>
  );
}
```

14. Now we are going to add a little more interactivity to this application. We need to change it so that we can only submit our blog post when it exceeds **100** characters. To do this, we will need to add a new submitDisabled state property in our **constructor** and default it to true:

    The updated constructor will be as follows:

```
constructor(props) {
    super(props);
    this.state = { field: "", submitDisabled: true };
}
```

15. Modify the button so that its disabled property is set to the value of our new state property:

```
<button disabled={this.state.submitDisabled}>Submit Post</button>
```

16. Next, we will need to write a validation function that will be able to monitor the length of the field and modify the submitDisabled state depending on the length:

```
validateFieldLength() {
  if (this.state.submitDisabled && this.state.field.length >
   100) {
    this.setState({ submitDisabled: false });
  } else if (this.state.submitDisabled &&
   this.state.field.length <= 100) {
    this.setState({ submitDisabled: true });
  }
}
```

17. We will want to make it so that validation is called when the state is updated to make sure the validation is being updated per each character entered. You can do this in a few different ways:

```
updateFieldLength(event) {
  const field = event.target.value;
  this.setState({ field }, () => {
    this.validateFieldLength();
  });
}
```

18. Finally, add some sort of visual notification or otherwise make it clear when the user clicks the **submit** button that they have submitted the form (an alert will suffice here):

19. First, add a new function for the submit form action:

```
submitForm() {
    alert("Submitting the blog post.");
}
```

20. Then add the call to the **`submit`** button:

```
<button disabled={this.state.submitDisabled}
     onClick={this.submitForm}>Submit Post</button>
```

You should have an application where you cannot submit unless you have written at least **`100`** characters. The end result should be an application that looks like this:

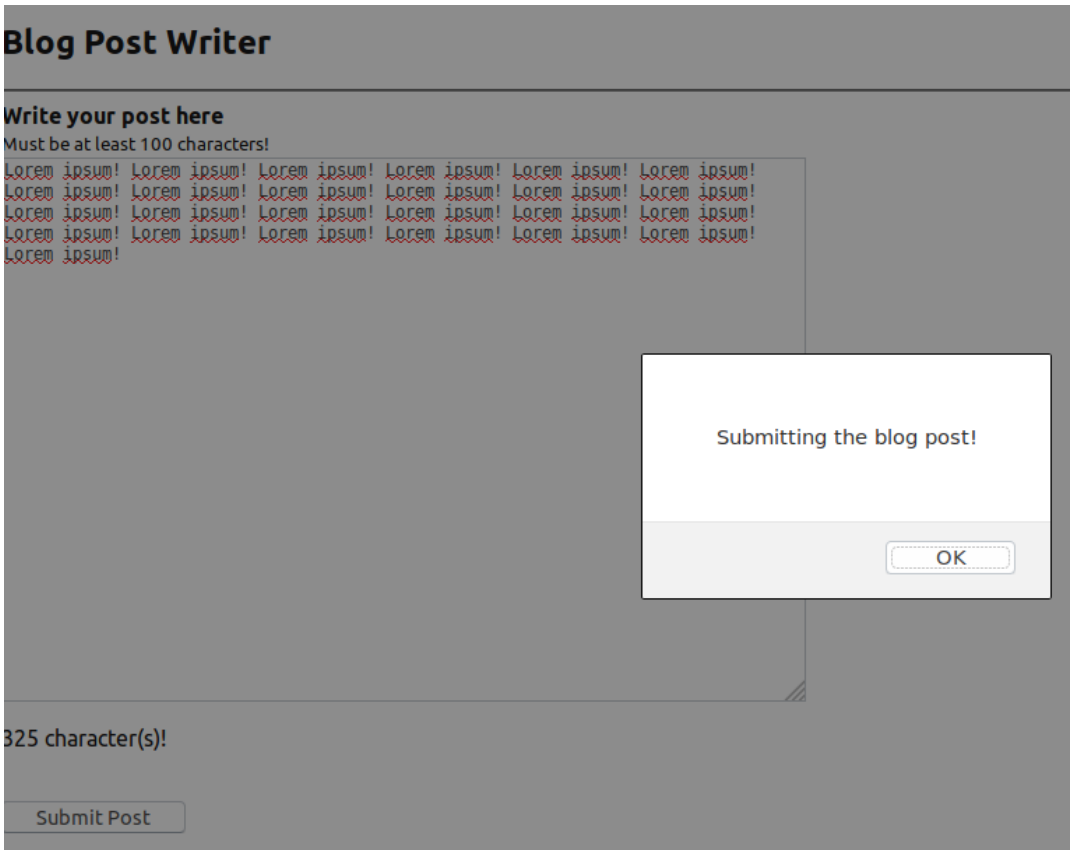

Figure 2.16: The final UI with the Submit button

CHAPTER 3: CONDITIONAL RENDERING AND FOR LOOPS

ACTIVITY 3.01: BUILDING A BOARD GAME USING CONDITIONAL RENDERING

Solution:

1. First, start off by creating your new React project (as we have done so many times before). As per usual, delete **`src/logo.svg`**, and then change the **`import`** statements in **`src/App.js`** to import **`React`** and the **`Component`** class from React. Do not delete the **`import`** statement for **`src/App.css`**. In **`src/App.js`**, the code should look like this:

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
  render() {
  return (
    <div className="App">
      <h1>Memory Game</h1>
    </div>
  );
  }
}

export default App;
```

The CSS that we used here is as follows:

```
.Tile {
width: 200px;
height: 200px;
font-size: 64pt;
border: 2px solid #aaa;
text-align: center;
margin: 10px;
padding: 10px;
float: left;
cursor: pointer;}
```

Additionally, we need to write the code for a hover modifier, a flipped tile class, tiles that match, and some for the **`New Game`** button. The relevant code is presented in the next few steps.

2. Write the code for a hover modifier that highlights the tile the user is over with a red outline:

```
.Tile:hover { border: 2px solid #f00; }
```

3. Write the code for a flipped tile class that changes the background:

```
.Tile.flipped { background: #aaa; }
```

4. Write the code for a matched tile that changes the tile to look distinctly like it has been previously matched:

```
.Tile.matched { border: 2px solid #040; background: #afa; }
```

5. Some CSS to spruce up the **`New Game`** button that we will add later:

```
button.reset { padding: 10px; width: 200px; }
```

6. Set a constant for the number of tiles to generate at the start of your games. Put this at the top to make it global:

```
const TILE_COUNT = 24;
```

7. Now set up the constructor for this project. This should start off by passing the props to the parent's constructor, and then defaulting our state. We will need to keep track of the tiles in our game board, what the last tile flipped over was, and how many clicks the player has made so far:

```
constructor(props) {
  super(props);

  // Our initial state should have a blank list of tiles, no previous
flipped tile, and the number of clicks for the user starting at 0
  this.state = {
    tiles: [],
    lastFlipped: null,
    clicks: 0
  };
}
```

8. Add on to our **`render`** method by displaying the number of player-clicks and creating a **`New Game`** button (which will not do anything yet):

```
render() {
  // And finally, our render method.
  return (
    <div className="App">
```

```
        <h1>Memory Game</h1>
        <strong>Clicks: {this.state.clicks}</strong>
        <br />
        <button className="reset">New Game</button>
      </div>
    );
  }
```

9. Write the function that will generate the tiles when the **`New Game`** button is clicked:

```
resetTiles() {
    // Start off with a blank tileset
    let tiles = [];
    // And start off with our numbering at 0
    let number = 0;
    // We're going to create two of the same tile for each
     number
    for (let i = 0; i < TILE_COUNT; i += 2) {
        number++;
        // Create two tiles
        let tileOne = { flipped: true, matched: false, number
    };
        let tileTwo = { flipped: true, matched: false, number
    };
        // And add those to the list of tiles
        tiles = [...tiles, tileOne, tileTwo];
    }
    // Then randomize the tiles!
    for (let i = 0; i < tiles.length; i++) {
        // For each tile, pick a random one to switch it with
        const swapWith = Math.floor(Math.random() *
     tiles.length);
        // Swap the two tiles in place
        [tiles[i], tiles[swapWith]] = [tiles[swapWith],
    tiles[i]];
    }
    // Then update the state so the game starts over
    this.setState({ clicks: 0, tiles });
}
```

10. Add a **`bind`** statement for the **`resetTiles`** function in the **`constructor`**:

```
this.resetTiles = this.resetTiles.bind(this);
```

11. Add a call to **`resetTiles`** to our **`New Game`** button:

```
<button onClick={this.resetTiles} className="reset">New Game</button>
```

12. Add a loop to render multiple tiles below the **`New Game`** button. This can call an arbitrary function to actually create the tiles as JSX by which we can just call **`renderTile`**:

```
<hr />
{this.state.tiles.map((tile, index) => this.renderTile(tile,
   index))}
```

13. Write the **`renderTile(tile, index)`** function. We will start off simple and build it piece-by-piece. We will start off by giving it a list of CSS classes that will apply to it (the first being a **`Tile`** class) and if the tile is "flipped" we will give it an appropriate CSS class to mark it as **`flipped`**:

```
renderTile(tile, index) {
    let classes = ["Tile"];
    if (tile.flipped) {
        classes = [...classes, "flipped"];
    }
```

14. To the **`renderTile()`** function, we will also add a class if the tile is matched:

```
    if (tile.matched) {
        classes = [...classes, "matched"];
    }
```

15. Now set a unique key for the tile to avoid render errors:

```
let key = `tile-${index}`;
```

16. Next, only display the number of the tile if the tile has been flipped or matched. In addition, if a tile is clicked, it should trigger a different function to flip the tile, where the majority of our game/scoring logic will go:

```
renderTile(tile, index) {
    let classes = ["Tile"];
    if (tile.flipped) {
        classes = [...classes, "flipped"];
    }

    return (
        <div key={key} className={classes.join(" ")}
    onClick={() => this.flipTile(index)}>
```

```
            {!tile.flipped && tile.number}
            </div>
        );
    }
```

As a check, our example game should look similar to this before the **`new game`** button is clicked:

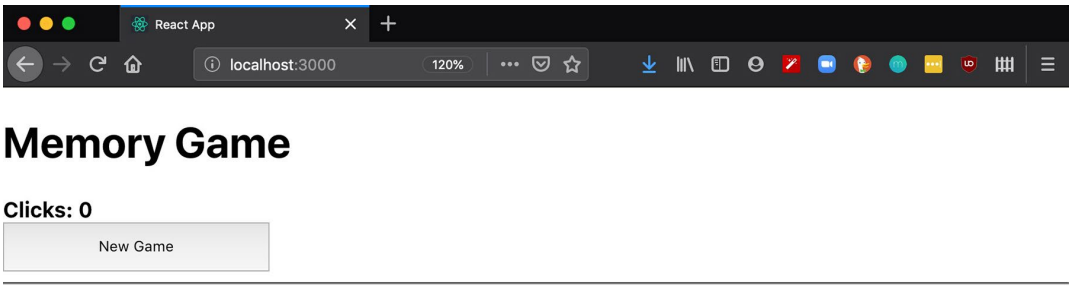

Figure 3.13: Basic memory game app

After the **`New Game`** button is clicked, the game should look like this:

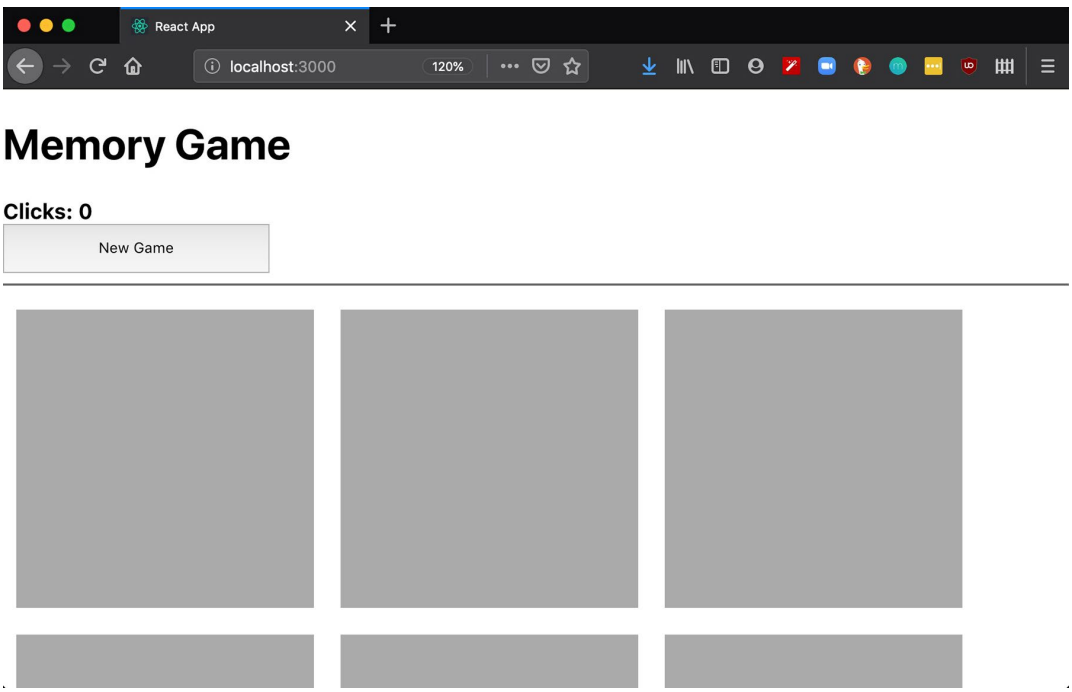

Figure 3.14: Building Memory Game

In the **`flipTile()`** function, we need to write logic to do quite a few things, so again we will step through it piece by piece. Start off by creating a variable to act as a temporary store for the list of tiles:

```
flipTile(index) {
    let tiles = this.state.tiles;
```

17. Grab the current tile and store that:

```
    let tile = tiles[index];
```

18. Now increment the number of clicks from the state and store the last tile that we flipped over:

```
    let clicks = this.state.clicks + 1;

let lastFlipped = this.state.lastFlipped;
```

19. If the last tile flipped is **`null`** (that is, we have not flipped over any tiles yet), then reset all of the tiles to be face down (this makes sure we can't have too many tiles face up). Then, you will toggle the flipped state of the tile that was just clicked on:

```
    if (lastFlipped === null) {
        tiles = this.flipAllBackOver(tiles);
        tile.flipped = !tile.flipped;
        lastFlipped = index;
    }
```

20. Now add an **`else`** statement to cover situations when we do have a previously flipped tile:

```
else {
      tile.flipped = !tile.flipped;
      let lastFlippedTile = this.state.tiles[lastFlipped];
```

21. Next, if the last flipped tile's number matches the number of the tile we just clicked on, we will update both tiles' **`matched`** properties to **`true`**, and update the tile in the list:

```
          if (lastFlippedTile.number === tile.number) {
            lastFlippedTile.matched = true;
            tile.matched = true;
            tiles[lastFlipped] = lastFlippedTile;
}
```

22. Mark the **`lastFlipped`** variable as **`null`** again, since we can only flip over two tiles at a time:

```
          lastFlipped = null;
 }
```

23. Now update the state of the current tile, and set the state of the component with all of the temporary variables we've been updating:

```
    tiles[index] = tile;
this.setState({ clicks, tiles, lastFlipped });}
```

24. Add a **`bind`** statement in the constructor for the **`flipTile`** function:

```
this.flipTile = this.flipTile.bind(this);
```

25. Write a helper function to flip all of the tiles back over unless they're matched:

```
flipAllBackOver(tiles) {
  // For each tile, we want to see if it is matched. If it isn't, we
flip it back over.
  // Otherwise we leave it be.
  tiles.forEach(tile => {
    if (!tile.matched) {
      tile.flipped = true;
    }
  });
  return tiles;
}
```

This will give us a final build that should look something like the following game:

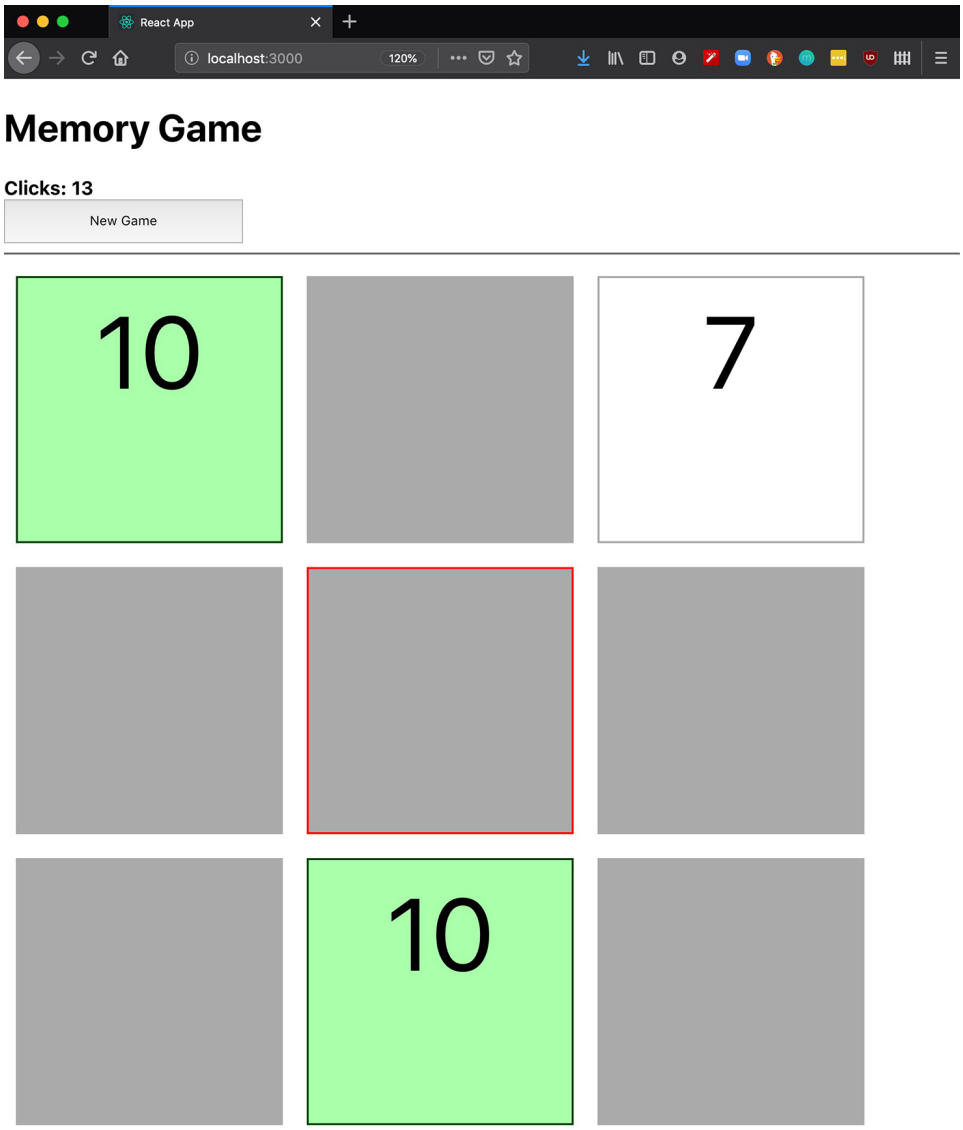

Figure 3.15: Memory Game app

CHAPTER 4: REACT LIFECYCLE METHODS

ACTIVITY 4.01: CREATING A MESSAGING APP

Solution:

1. Start off by creating a new React project and take a look at the browser window that should have opened for you automatically:

```
$ npx create-react-app ajax-simulation
$ cd ajax-simulation
$ yarn start
```

2. Next, strip out the stuff we don't need. Delete **`src/logo.svg`** and delete the contents of **`src/App.css`**. Clean out the contents of the **`App`** component and replace it with a class component instead. Since we are using a class component here, we will need to change the **`import`** statement at the top to **`import {Component} from React`**:

```
import React, { Component } from 'react';
import "./App.css";

    class App extends Component {
      render() {
        return (
          <div className="App">Messaging App</div>
        );
      }
    }
export default App;
```

3. Include a new file in your **`src`** directory called **`AjaxLibrary.js`**. The file can be found here https://packt.live/2N3BEoA

   You will need to start building up this app's functionality, so we will start off by implementing the base user count display functionality.

4. Create a new file, **`src/UserCountDisplay.js`**, where we will start implementing the **`User Count`** functionality with the help of the preceding library. You will need to implement three lifecycle methods: **`constructor()`**, **`componentDidMount()`**, and **`render()`**:

```
import React, { Component } from 'react';

import { fetchUserCount } from './AjaxLibrary';

class UserCountDisplay extends Component {
  constructor(props) {
  super(props);
  this.state = { userCount: 0, loadingUserCount: false };
  }
  async componentDidMount() {
   this.setState({ loadingUserCount: true });
   const userCount = await fetchUserCount();
   this.setState({ userCount, loadingUserCount: false });
  }
  render() {
  if (this.state.loadingUserCount) {
    return <p>Loading user count...</p>;
  } else {
    return <p>Users in the app: {this.state.userCount}</p>;
  }
  }
}

export default UserCountDisplay;
```

Note that in the preceding library, the **`sleep`** function is implemented through **`async`**/**`await`**, so your **`componentDidMount`** declaration will need the **`async`** keyword in front of it.

Your **`UserCountDisplay`** component has a few requirements: while the AJAX call is loading, you must display a message indicating that it is loading the user count, the initial user count should be **`0`**. Finally, after the component is done loading, it should display a message like the following: **`Users in the app: 2`**.

5. Now return to **`src/App.js`**, add your imports for the **`UserCountDisplay`** component, and add them to your **`render`** function:

```
import UserCountDisplay from './UserCountDisplay';

class App extends Component {
  render() {
    return (
      <div className="App">
        <h1>Messaging App</h1>
        <UserCountDisplay />
      </div>
    );
  }
}
```

The app should currently look like this:

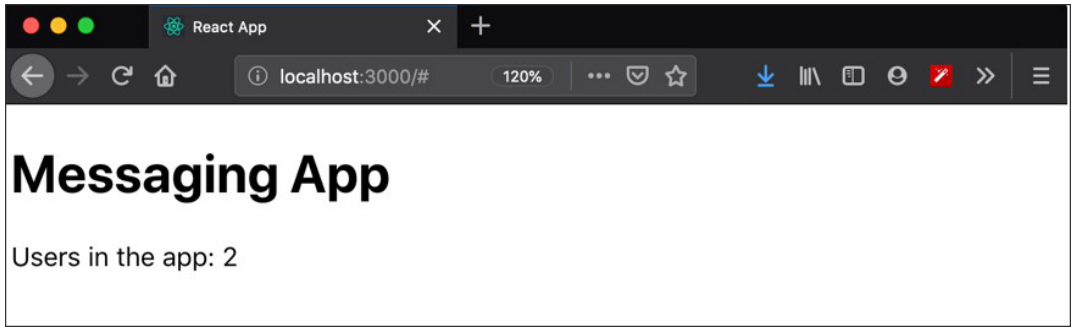

Figure 4.12: The messaging app showing the number of users

6. Implement the **`Login Display`** component next. This will be another new component that we will name **`LoginDisplay`**.

7. Create **`src/LoginDisplay.js`**. This component also has a few requirements attached to it that we will step through, but we will start with the basic skeleton first:

```
import React, { Component } from 'react';
class LoginDisplay extends Component {
}
export default LoginDisplay;
```

8. First, before a login happens, we should display a **`login`** form to the user. This will only contain a single form element for **`username`**:

```
class LoginDisplay extends Component {
  render() {
  return (
      <div className="LoginDisplay">
        <div className="Login">
        Login Username:
        <input type="text" />
        <button>Login</button>
        </div>
      </div>
  );
  }
}
```

   If we log in with a username specified in the library, we should set the state in the component to store that particular user as a result of a call to the **`AjaxLibrary`** file's **`fetchUser(username)`** function.

9. To do that, we will need to import the **`AjaxLibrary`** file in our React component:

```
import { fetchUser } from './AjaxLibrary';
```

10. Set up the initial state for the component to include a login **`username`**, whether we're loading the login form or not, and the user that we're logging in as follows:

```
  constructor(props) {
  super(props);
  this.state = { loginUsername: null, loadingLogin: false,
  user: null };
  }
```

11. Add an event to update the login **`username`** as the user types. This will be triggered by the **`onChange`** event on the **`username`** textbox (the changes have been highlighted):

```
class LoginDisplay extends Component {
  constructor(props) {
  super(props);
  this.state = { loginUsername: null, loadingLogin: false,
  user: null };
  this.updateLoginForm = this.updateLoginForm.bind(this);
```

```
    }
    updateLoginForm(evt) {
    const loginUsername = evt.target.value;
    this.setState({ loginUsername });
    }
    render() {
    return (
      <div className="LoginDisplay">
          <div className="Login">
          Login Username:
          <input type="text" onChange={this.updateLoginForm} />
          <button>Login</button>
          </div>
      </div>
    );
    }
  }
```

12. Now, add an event listener to the button after the text field that will attempt to log in by calling out to the **`fetchUser`** function we imported earlier:

```
  class LoginDisplay extends Component {
    constructor(props) {
    super(props);
    this.state = { loginUsername: null, loadingLogin: false, user: null
  };
    this.updateLoginForm = this.updateLoginForm.bind(this);
    this.doLogin = this.doLogin.bind(this);
    }
    updateLoginForm(evt) {
    const loginUsername = evt.target.value;
    this.setState({ loginUsername });
    }
    async doLogin() {
    this.setState({ loadingLogin: true });
    const user = await fetchUser(this.state.loginUsername);
    this.setState({ user, loadingLogin: false });
    }
    render() {
    return (
      <div className="LoginDisplay">
          <div className="Login">
```

```
          Login Username:
          <input type="text" onChange={this.updateLoginForm} />
          <button onClick={this.doLogin}>Login</button>
          </div>
      </div>
    );
    }
  }
```

13. To make our code a little more organized, move the login form to a separate function that will display the form:

```
    loginForm() {
    return (
      <div className="Login">
        Login Username:
        <input type="text" onChange={this.updateLoginForm} />
        <button onClick={this.doLogin}>Login</button>
      </div>
    );
    }
    render() {
    return <div className="LoginDisplay">{this.loginForm()}</div>;
    }
```

14. In addition, if we are logged in, we should display a **`Logout`** button that sets the user state back to **`null`**. We should also display a message that tells the user if we are currently attempting to log in:

```
  import React, { Component } from 'react';
  import { fetchUser } from './AjaxLibrary';
  import UserDisplay from './UserDisplay';

  class LoginDisplay extends Component {
    constructor(props) {
    super(props);
    this.state = { loginUsername: null, loadingLogin: false,
    user: null };
    this.updateLoginForm = this.updateLoginForm.bind(this);
    this.doLogin = this.doLogin.bind(this);
    this.doLogout = this.doLogout.bind(this);
    }
    updateLoginForm(evt) {
```

```
  const loginUsername = evt.target.value;
  this.setState({ loginUsername });
  }
  async doLogin() {
  this.setState({ loadingLogin: true });
  const user = await fetchUser(this.state.loginUsername);
  this.setState({ user, loadingLogin: false });
  }
  doLogout() {
  this.setState({ user: null });
  }
  loginForm() {
  if (this.state.loadingLogin) {
    return 'Trying to login, please wait...';
  } else {
    if (!this.state.user) {
      return (
        <div className="Login">
          Login Username:
          <input type="text" onChange={this.updateLoginForm} />
          <button onClick={this.doLogin}>Login</button>
        </div>
      );
    } else {
      return <button onClick={this.doLogout}>Logout</button>;
    }
  }
  }
  render() {
  return (
    <div className="LoginDisplay">
      {this.loginForm()}
      <hr />
      <UserDisplay user={this.state.user} />
    </div>
  );
  }
}
export default LoginDisplay;
```

15. Now, return to **`src/App.js`**, include an **`import`** for **`LoginDisplay`**, and add the **`LoginDisplay`** component to your **`render`** function:

```
import UserCountDisplay from './UserCountDisplay';
import LoginDisplay from './LoginDisplay';

class App extends Component {
  render() {
    return (
      <div className="App">
        <h1>Messaging App</h1>
        <UserCountDisplay />
        <LoginDisplay />
      </div>
    );
  }
}
```

The form should look like this:

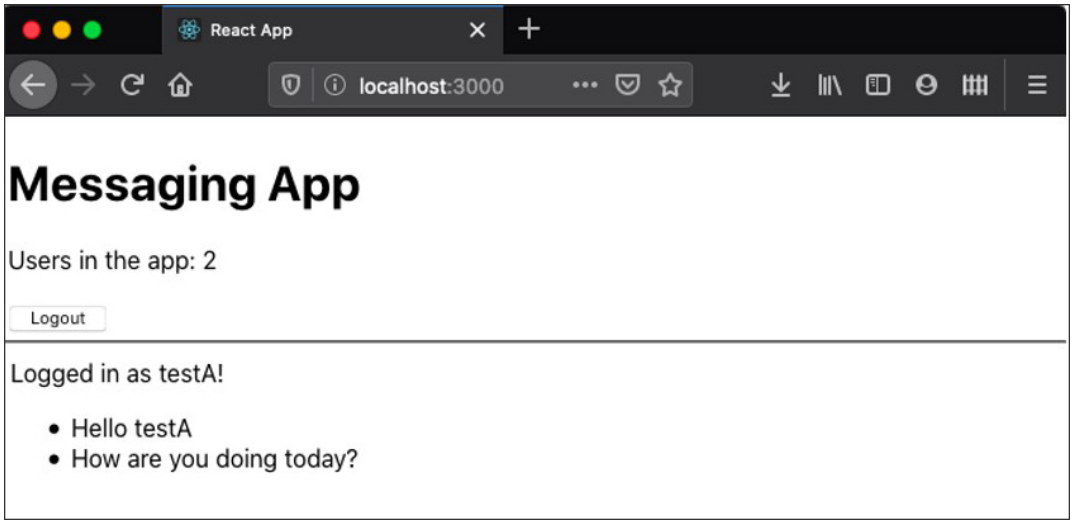

Figure 4.13: Login display component

If we successfully log in, we should see this UI:

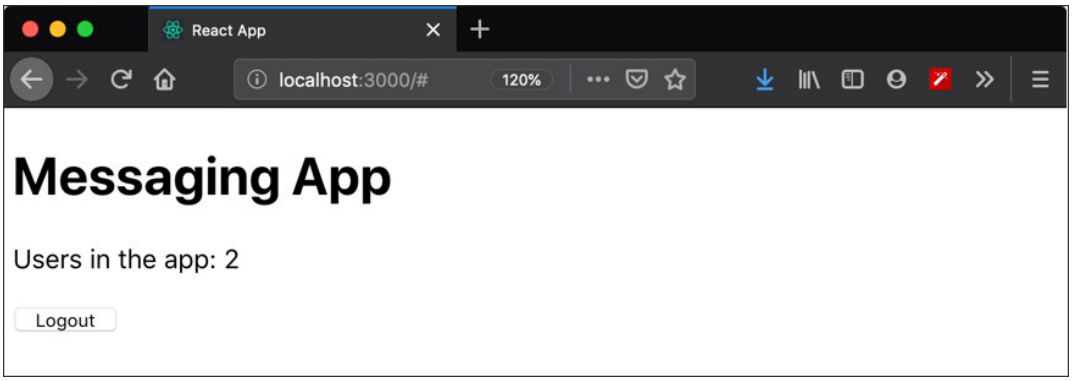

Figure 4.14: Successfully logged in

16. Let's expand the **`Login Display`** component to include the **`User`** information when logged in. We will do this by creating **`src/UserDisplay.js`**. Our **`UserDisplay`** component has the following requirements.

    It should accept **`user`** as a prop, which should be the user from the state of the **`LoginDisplay`** component. If the user is **`null`**, display nothing by returning **`null`**. Otherwise, display a **`Logged in as user.username`** message to the user:

```
import React, { Component } from 'react';

class UserDisplay extends Component {
  render() {
  if (!this.props.user) {
    return null;
  }
  return (
    <div className="UserDisplay">
      Logged in as {this.props.user.username}!
    </div>
  );
  }
}
export default UserDisplay;
```

17. Return to **`src/LoginDisplay.js`** and add a reference to the new **`UserDisplay`** component in the **`render()`** function. This should accept a single prop called **`user`** that should be set to the **`user`** property in **`this.state`**, which should give us the following UI:

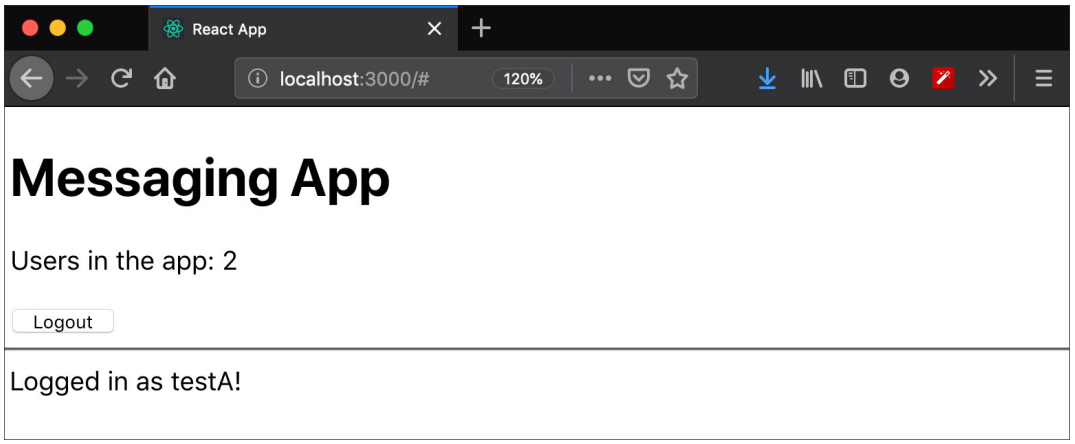

Figure 4.15: Login message and the count of users in Messaging App

Expand the functionality of the app by adding a way to display messages for the user, which we will use to create another new component.

18. Create **`src/MessageDisplay.js`**.

    The component should take a single prop, **`userId`**. This will be used to call the **`fetchMessages(userId)`** function from the fake AJAX library when the component mounts. Similar to before, we should display a loading message when we're fetching the messages through our library and display the messages when they're loaded, so we will need a state for when we're loading messages or not and a state for the list of messages.

19. Let's start our **`render`** function by just displaying no messages:

```
import React, { Component } from "react";
import { fetchMessages } from "./AjaxLibrary";
class MessageDisplay extends Component {
  constructor(props) {
  super(props);
  this.state = { loadingMessages: false, messages: [] };
  }
```

```
  render() {
    return <p>No messages for you!</p>;
  }
}
```

Now we need the **`async`** function responsible for when the component mounts, which will make the **`await`** function call to our library to fetch the messages for the user:

```
  async componentDidMount() {
  this.setState({ loadingMessages: true });
  const messages = await fetchMessages(this.props.userId);
  this.setState({ messages, loadingMessages: false });
  }
```

When the component is loading, display a **`Messages still loading`** message. When the component is complete, display the messages if there are any messages for that user. If the user has no messages and loading is complete, display something to the user indicating that they have no messages.

20. Let's update the **`render()`** function:

```
  render() {
  if (this.state.loadingMessages) {
      return <p>Messages still loading...</p>;
  } else {
      if (this.state.messages.length > 0) {
          return (
        <ul>
          {this.state.messages.map((msg, index) => (
            <li key={`m-${index}`}>{msg}</li>
          ))}
        </ul>
          );
      } else {
          return <p>No messages for you!</p>;
      }
  }
  }
```

21. When the component is unmounted from the page, display a message to the console indicating that the messages component is being removed:

```
  componentWillUnmount() {
  console.log("Unmounting messages");
  }
```

You will also need to go back and update your **`UserDisplay`** component to use this new component and pass in the **`user.id`** from the props to the **`userId`** property on the **`MessageDisplay`** component:

```
import MessageDisplay from "./MessageDisplay";
class UserDisplay extends Component {
  render() {
  if (!this.props.user) {
      return null;
  }
  return (
      <div className="UserDisplay">
          Logged in as {this.props.user.username}!
          <MessageDisplay userId={this.props.user.id} />
      </div>
  );
  }
}
```

22. Compile and start the application. Go to **`http://localhost:3000`** and verify the states of the UI when loaded in as a valid user with messages.

    The output is as follows:

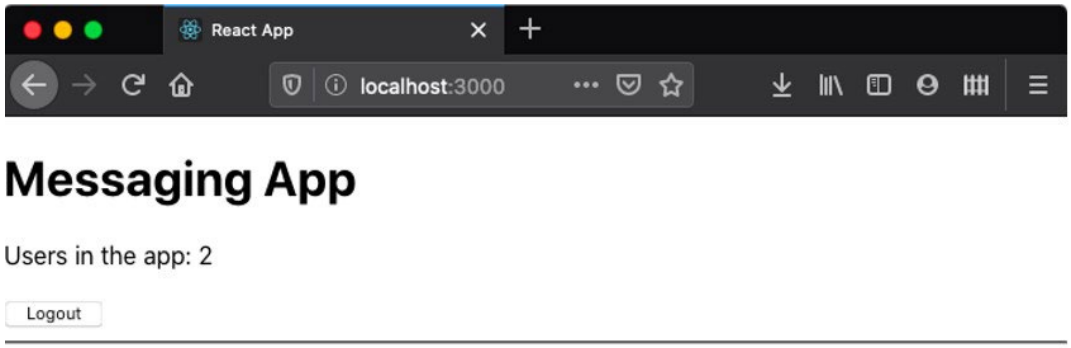

Figure 4.16: Messaging App with the messages

This is what you get when loaded in as a valid user with no messages:

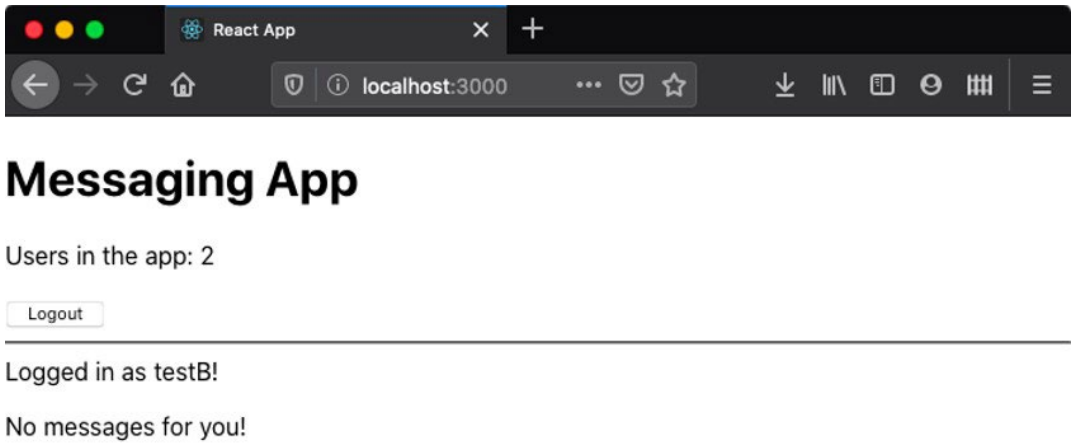

Figure 4.17: Messaging App with no messages

This is what you get when the **`Logout`** button is hit and **`MessageDisplay`** is unmounted:

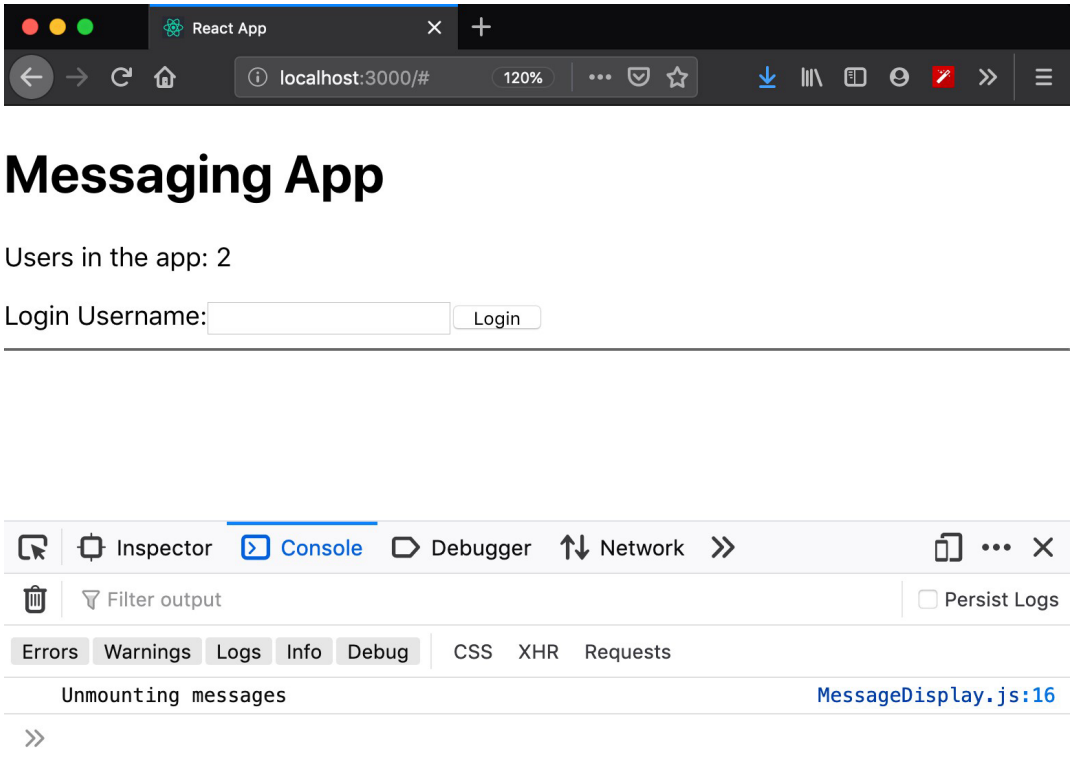

Figure 4.18: Unmounted message displayed

CHAPTER 5: CLASS AND FUNCTION COMPONENTS

ACTIVITY 5.01: CREATING A COMMENTS SECTION

Solution:

1. Structuring the app
2. Looking at the data, the following structure might work:

```
App
  |-CommentSection
    |-Comment
```

   The **`CommentSection`** component holds all the comments. It shows the heading and has a few associated styles.

   The **`comment`** component renders the comment.

3. Creating the app

   In the command line, create the app and navigate to the folder. We can now start running the app by running the **`npm start or yarn start`** command. Open the folder using your favorite code editor:

```
npx create-react-app comment-app
cd comment-app
yarn start
```

4. Adding the data file and deleting **`src/logo.svg`**
5. Delete **`src/logo.svg`** and add the **`src/comments.json`** file provided in the activity.
6. Adding the **`CommentSection`** component in **`src/components/CommentSection/index.js`**

   Now, let's create our container component, which will act as a placeholder component for our comments. Since this can hold multiple comments for a page, we will call it **`CommentSection`**:

```
import React, {Component} from 'react';
// styles
import './styles.css';

class CommentSection extends Component {
  render () {
```

```
      return (
        <section className="comments">
          <h1>Comments</h1>
        </section>
      );
    }
  }
  export default CommentSection;
```

7. Also, we will add some basic styling for our **`comment`** section:

```
  .comments {
    margin: 5% 0;
  }
```

8. Passing props

   We will read data from this file in our **`App.js`** file. We can pass this data to our **`CommentSection`** component as props:

```
  import React from "react";
  // styles
  import "./App.css";
  // components
  import CommentSection from "./components/CommentSection/";
  // data
  import { comments } from "./comments.json";

  function App() {
    return (
      <div className="container">
        <CommentSection comments={comments} />
      </div>
    );
  }

  export default App;
```

9. Add the styling for the component in **`App.css`** file

```
.container {
  max-width: 960px;
  margin: 0 auto;
  padding: 0 10%;
}

.center {
  text-align: center;
}
```

Please note that in our JSX, while using **`CommentSection`**, we provide comments that we extracted from our JSON file as an attribute, **`comments={comments}`**. When we do this, the variable passed like this is made available to the component as a prop.

10. Extracting and rendering comments

Now, we can use these comments to display in our **`CommentSection`** component.

In **`src/components/CommentSection/index.js`**, we will now get the values for comments from our props and use this to show our comments:

```
import React, { Component } from "react";
// styles
import "./styles.css";

class CommentSection extends Component {
  render() {
    const { comments } = this.props;
    if (!comments || comments.length === 0) {
      return null;
    }
    return (
      <section className="comments">
        <h1>Comments</h1>
        {comments.map((comment, key) => (
          <div className="comment" key={`comment_${key}`}>
            <p>{comment.text}</p>
            <strong>
              <small>{comment.name}</small>
```

```
            </strong>
          </div>
        ))}
      </section>
    );
  }
}
export default CommentSection;
```

11. **`Comment`** component

    We have now shown the comments on screen, but keeping our design principles, it is wise to create a new reusable **`comment`** component that receives a comment as an input prop. In **`src/components/Comment/index.js`**, we will move the styles and place them next to the component:

```
import React from "react";
// styles
import "./styles.css";

export const Comment = ({ comment }) => (
  <div className="comment">
    <h2>{comment.name}</h2>
    <p>{comment.text}</p>
  </div>
);
```

    In **`src/components/Comment/styles.css`**, add the following code:

```
.comment {
  border: solid 1px #eee;
  padding: 2rem;
  margin-bottom: 0.5rem;
}
```

    This component can be used in our **`CommentSection`** component:

12. In **`src/components/CommentSection/index.js`**, add the following code:

```
import React, { Component } from "react";
// styles
import "./styles.css";
// components
import { Comment } from "../Comment";
```

```
class CommentSection extends Component {
  render() {
    const { comments } = this.props;
    if (!comments || comments.length === 0) {
      return null;
    }
    return (
      <section className="comments">
        <h1>Comments</h1>
        {comments.map((comment, key) => (
          <Comment key={`comment_${key}`} comment={comment}
      level={1} />
        ))}
      </section>
    );
  }
}
export default CommentSection;
```

The output so far shows a very basic **`Comments`** section with the name and comment:

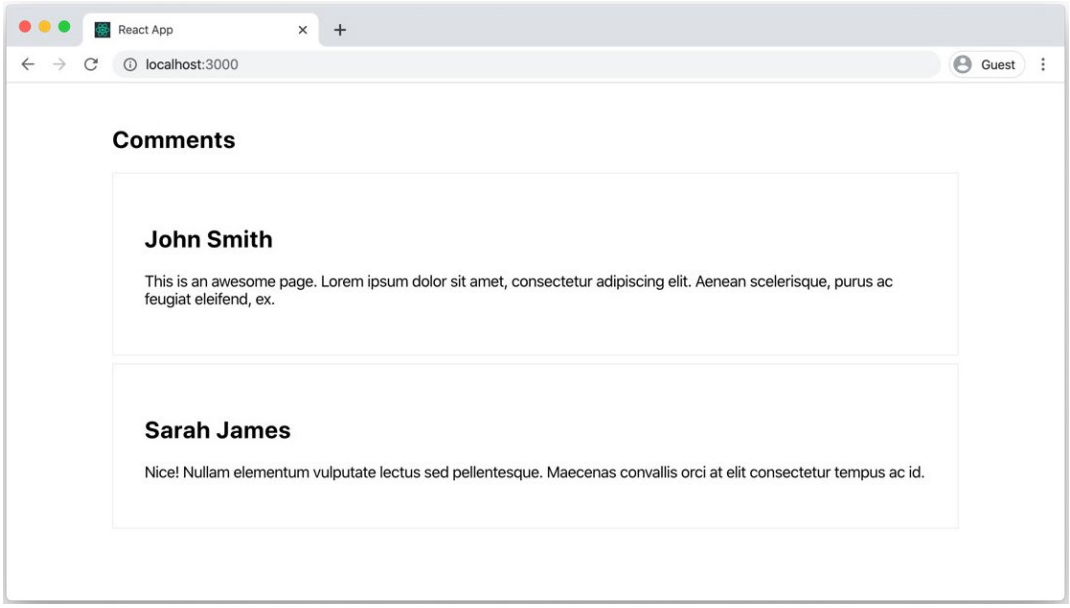

Figure 5.15: Comments app

13. Showing all comments' data and styling

    We will now display more information from comments and also style them.

    We can now add a profile image and the date information. To style the page, we shall tweak the HTML layout and add classes. This allows us to add CSS classes to style the comments:

    The code in **`src/components/Comment/index.js`** looks like this:

```
import React from "react";
// styles
import "./styles.css";

export const Comment = ({ comment }) => (
  <div className="comment">
    <div className="comment_container">
      <img className="comment_profile" src={comment.image}
     alt={comment.name} />
      <div className="comment_content">
        <h2 className="comment_name">{comment.name}</h2>
        <strong
      className="comment_time">{comment.time}</strong>
        <p>{comment.text}</p>
        <button className="comment_button">Reply</button>
      </div>
    </div>
  </div>
);
```

    From the data, we have the URL of the image. We can supply it to the **`img`** element as an argument to the **`src`** attribute.

    Along with the changes to the HTML, we will also need to replace the styles:

14. In **`src/components/Comment/styles.css`** add the following code:

`styles.css`

```
4 .comment_container {
5  border: solid 1px #aaa;
6  padding: 1rem;
7  margin-bottom: 0.5rem;
8  display: flex;
9 }
```

The complete code of this file can be found here https://packt.live/3fKectf

The output should be as follows:

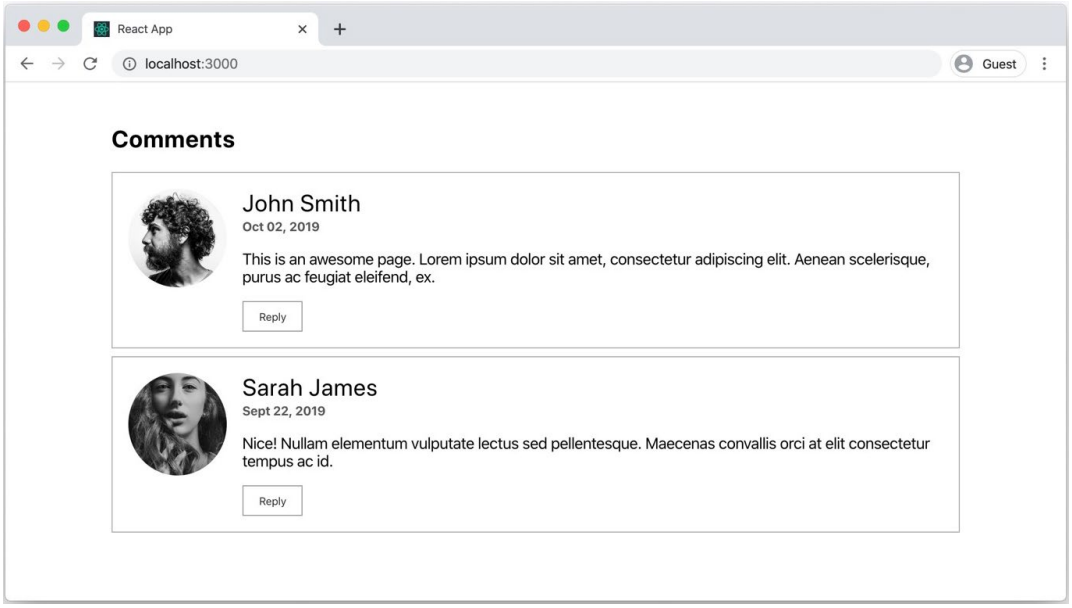

Figure 5.16: Comments section app

This looks much better now and looks like the output we need. However, we have two nested comments in the first comment. And assuming nesting is only allowed for the first level, we need to show the comments with no reply button. So, we can add a prop named **`level`** that can be used to check whether the reply button needs to be shown. We can use a nested **`comment`** component to show a comment within it. We check whether the **`comments`** array is valid before we execute line 17 **`comment.comments && ...`**:

15. In **`src/components/Comment/index.js`**, add the following code:

```
import React from "react";
// styles
import "./styles.css";

export const Comment = ({ comment, level }) => (
  <div className="comment">
    <div className="comment_container">
      <img className="comment_profile" src={comment.image}
alt={comment.name} />
```

```
          <div className="comment_content">
            <h2 className="comment_name">{comment.name}</h2>
            <strong className="comment_time">{comment.time}</strong>
            <p>{comment.text}</p>
            {level === 1 && <button
         className="comment_button">Reply</button>}
          </div>
        </div>

        {comment.comments &&
          comment.comments.map((comment, key) => (
            <Comment key={`comment_${key}`} comment={comment} />
          ))}
      </div>
    );
```

We also need to style the nested comment to indicate that this is a reply to an original comment. This can be done by moving the comment a bit to the right:

16. In **`src/components/Comment/style.css`**, add the following code:

`styles.css`

```
1 .comment .comment {
2  margin-left: 3rem;
3 }
4 .comment_container {
5  border: solid 1px #aaa;
6  padding: 1rem;
7  margin-bottom: 0.5rem;
8  display: flex;
9 }
```

The complete code can be found here https://packt.live/3fKectf

The final output is as follows:

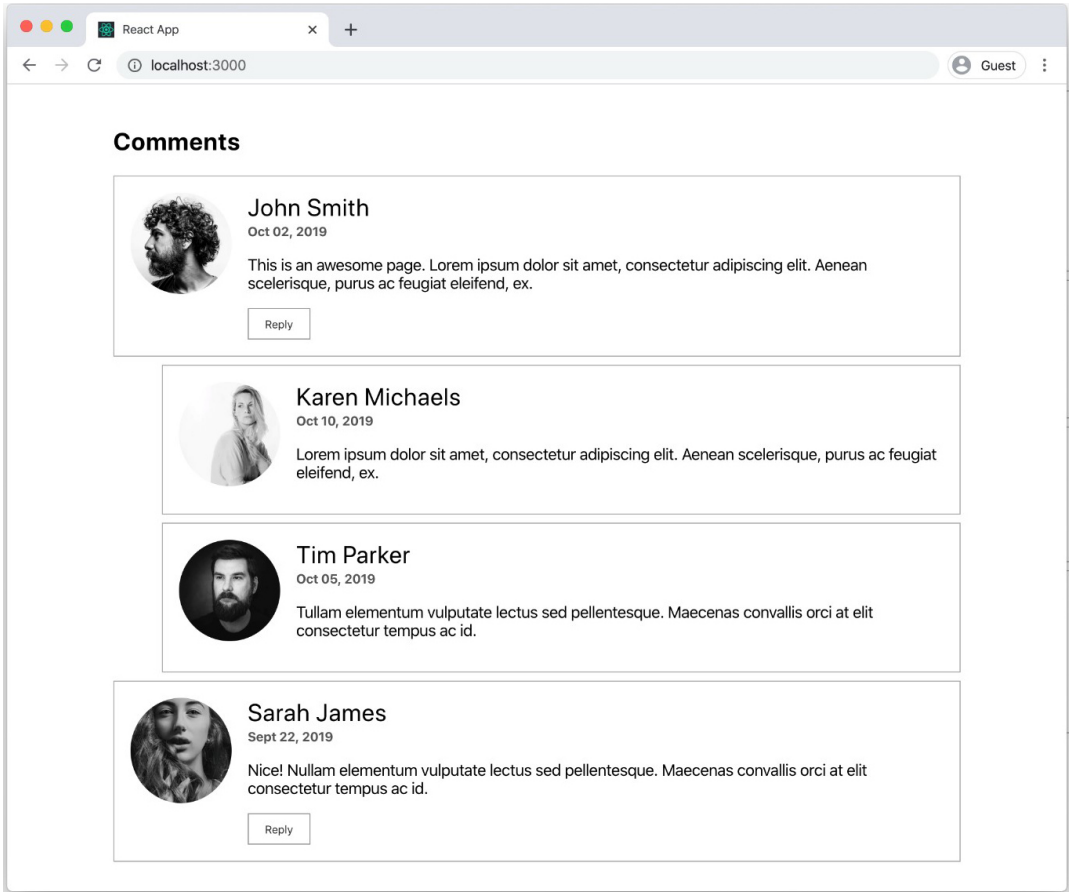

Figure 5.17: Comments section profile

Once we have updated the **`comment`** component and all the styling, the **`Comments`** section now looks the way it was intended.

CHAPTER 6: STATE AND PROPS

ACTIVITY 6.01: CREATING A PRODUCTS APP PAGE

Solution:

1. Create the app's structure. By analyzing the data and its requirements, we can come up with the following structure for the app:

```
App
  |-ProductListing
    |-Product
  |-Tags
    |-Tag
```

   The **`ProductListing`** component holds the **`Product`** component, which, in turn, renders the product information. The **`Tags`** component holds multiple **`Tag`** components. State exists at the **`App`** level in order to hold selected tag information and at the **`Product`** level to toggle description visibility.

2. Create the app in the command line using **`create-react-app`** with the name **`react-state-props`** and run the app in development mode:

```
npx create-react-app react-state-props
cd comment-app
yarn start
```

3. In **`src/App.js`**, import the data from the JSON file and extract the objects from it:

```
import React, { Component } from "react";
// styles
import "./App.css";
// data
import productData from "./products.json";

class App extends Component {
  render() {
    const { products, ingredients } = productData;

    return (
      <div className="container">
        <h1>Products</h1>
```

```
        </div>
      );
    }
  }

  export default App;
```

4. Next, we will create the **`ProductListing`** component, which can hold products and render the received products list. It receives products as props:

   In **`src/components/ProductListing/index.js`**, add the following code:

```
import React from "react";

const ProductListing = props => {
  const { products } = props;

  return (
    <div>
      { products.map((product, key) => (
        <div key={product.name}>{product.name}</div>
      ))}
    </div>
  );
};
export { ProductListing };
```

5. Add the **`ProductListing`** component to the **`App.js`** file:

```
import React, { Component } from "react";
// styles
import "./App.css";
// data
import productData from "./products.json";
// components
import { ProductListing } from "./components/ProductListing";

class App extends Component {
  render() {
    const { products, ingredients } = productData;

    return (
```

```
        <div className="container">
          <h1>Products</h1>
          <ProductListing />
        </div>
      );
    }
  }
  export default App;
```

6. Pass the extracted products as props for **`ProductListing`** in **`src/App.js`**

```
  class App extends Component {
    render() {
      const { products, ingredients } = productData;

      return (
        <div className="container">
          <h1>Products</h1>
          <ProductListing products={products} />
        </div>
      );
    }
  }
```

7. Create the **`Product`** component as a class component. Extract the product information that will be rendered on the screen:

   In **`src/components/Product/index.js`**, add the following code:

```
  import React, { Component } from "react";
  import "./styles.css";

  class Product extends Component {
    render() {
      const { name, price, summary } = this.props.product;
      return (
        <div className="product">
          <div>
            <h2>{name}</h2>
          </div>
          <strong>${price}</strong>
          <div><p>{summary}</p></div>
```

```
        </div>
      );
    }
  }

  export { Product };
```

8. Add styling for our product component: **`src/components/Product/styles.css`**:

```
.product {
  border: solid 1px #eee;
  padding: 20px;
  margin-bottom: 5px;
}
.product h2 {
  margin: 0;
}
```

9. Add state with a **`showDescription`** Boolean variable that can be used to show summary information:

   `src/components/Product/index.js`

```
import React, { Component } from "react";

import "./product.css";

class Product extends Component {
  constructor(props) {
    super(props);
    this.state = {
      showDescription: false
    };
  }
  render() {
    const { name, price, summary } = this.props.product;
    const { showDescription } = this.state;
    return (
      <div className="product">
        <div>
          <h2>{name}</h2>
        </div>
```

```
            <strong>${price}</strong>
            <div>{showDescription && <p>{summary}</p>}</div>
          </div>
        );
      }
    }
    export { Product };
```

10. Add a button, along with a method, to toggle the description when the button is clicked in **`src/components/Product/index.js`**:

`index.js`

```
5 class Product extends Component {
6  constructor(props) {
7    super(props);
8    this.state = {
9      showDescription: false
10    };
11    this.toggleDescription =
    this.toggleDescription.bind(this);
12  }
13  toggleDescription() {
14    this.setState({
15      showDescription: !this.state.showDescription
16    });
17  }
```

The complete code can be found here https://packt.live/2WWvR89

11. Add styling for the new button:

    In **`src/components/Product/styles.css`**, add the following code:

```
    .product_header {
      display: flex;
      justify-content: space-between;
    }
    .product button {
      background: white;
      border: solid 1px #eee;
      padding: 5px 10px;
      font-size: 20px;
    }
```

12. In **`src/components/Tag/index.js`**, create a **`Tag`** component with the required styling:

```
import React from "react";

import "./styles.css";

const Tag = props => (
  <button className="tag">
    {props.children}
  </button>
);

export { Tag };
```

13. Add the **`styles.css`** file to **`src/Components/Tag`**:

```
.tag {
  display: inline-block;
  padding: 8px 15px;
  margin: 15px 0 15px 20px;
  background: #eaeaea;
  border-radius: 5px;
  border: solid 1px #bbb;
}
```

14. Use the **`selectedTag`** prop to show if the tag is active and the **`handleTags`** callback function to set the tag selection:

    In **`src/components/Tag/index.js`**, add the following code:

```
import React from "react";
import "./styles.css";
const Tag = props => (
  <button
    className={`tag ${props.selectedTag === props.children ?
    "active" : ""}`}
    onClick={props.handleTags(props.children)}
  >
    {props.children}
  </button>
);
export { Tag };
```

15. In **`src/components/Tag/styles.css`**, add the styling code for tag is as follows

```
.tag.active {
  background: #333;
  color: white;
}
```

16. Create the **`Tags`** component in **`src/components/Tags/index.js`**, which houses tags. It receives the **`tags`**, **`selectedTag`**, and **`handleTags`** props and passes the values that are used for tag selection:

```
import React, { Component } from "react";
// component
import { Tag } from "../Tag";
class Tags extends Component {
  render() {
    const { tags, selectedTag, handleTags } = this.props;
    return (
      <div>
        Select Filter
        {tags.map((tag, key) => (
          <Tag
            key={`tag${key}`}
            selectedTag={selectedTag}
            handleTags={handleTags}
          >
            {tag}
          </Tag>
        ))}
      </div>
    );
  }
}
export { Tags };
```

17. Use the tags component in **App** and pass the ingredients as tags. Initialize the state with **selectedTag** set to **null**:

App.js

```
8 import { Tags } from "./components/Tags";
9
10 class App extends Component {
11  constructor(props) {
12    super(props);
13    this.state = {
14      selectedTag: null
15    };
16  }
…
23  render() {
24    const { products, ingredients } = productData;
25    const { selectedTag } = this.state;
26
27    return (
28      <div className="container">
29        <h1>Products</h1>
30        <Tags
31          tags={ingredients}
32          selectedTag={selectedTag}
33        />
```

The complete code can be found here https://packt.live/35YW6Pp

18. Add a **handleTags** method to set the tag when the tag is clicked. Pass it to the **Tags** component as a prop:

App.js

```
10 class App extends Component {
11  constructor(props) {
…
16    this.handleTags = this.handleTags.bind(this);
17  }
18  handleTags(tagName) {
19    return () => {
20      this.setState({ selectedTag: tagName });
21    };
22  }
```

The complete code can be found here https://packt.live/35YW6Pp

The component is now functional, and it should show a list of the available products:

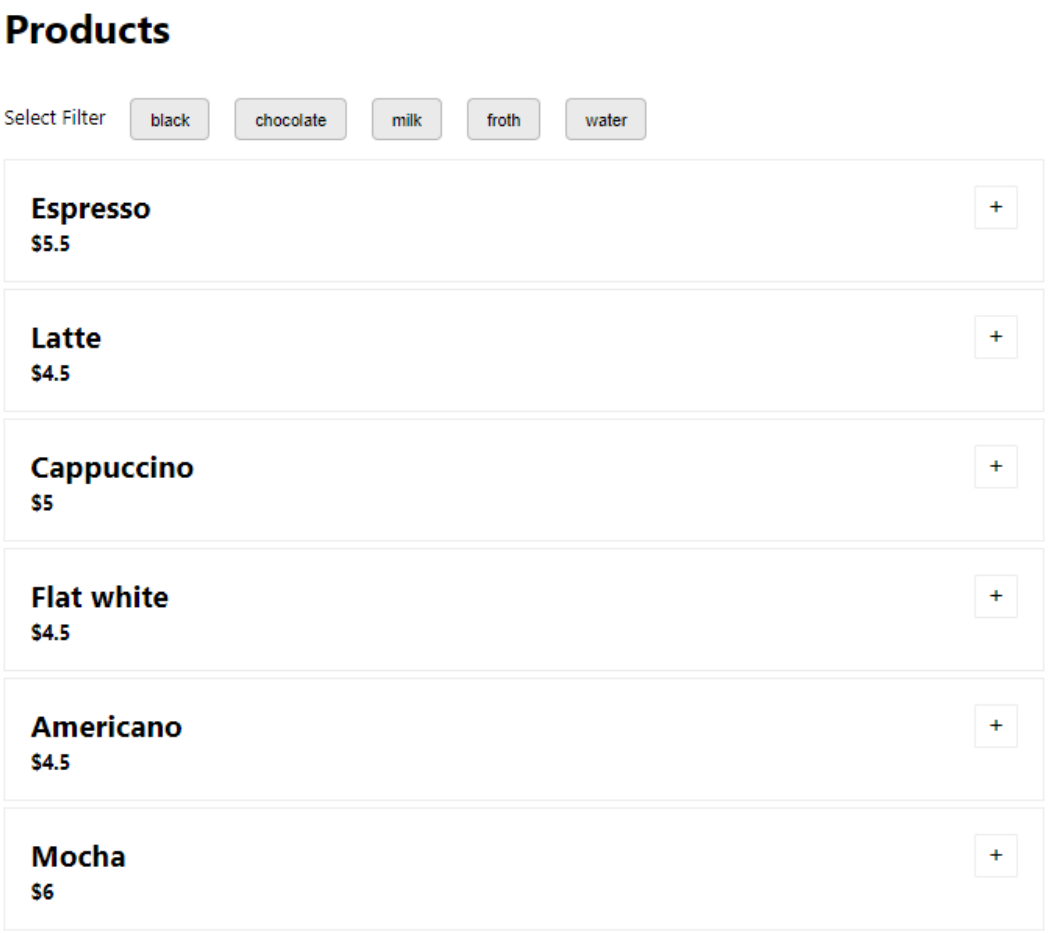

Figure 6.9: Products app page

Click on **`Tag`** to filter the product by tag:

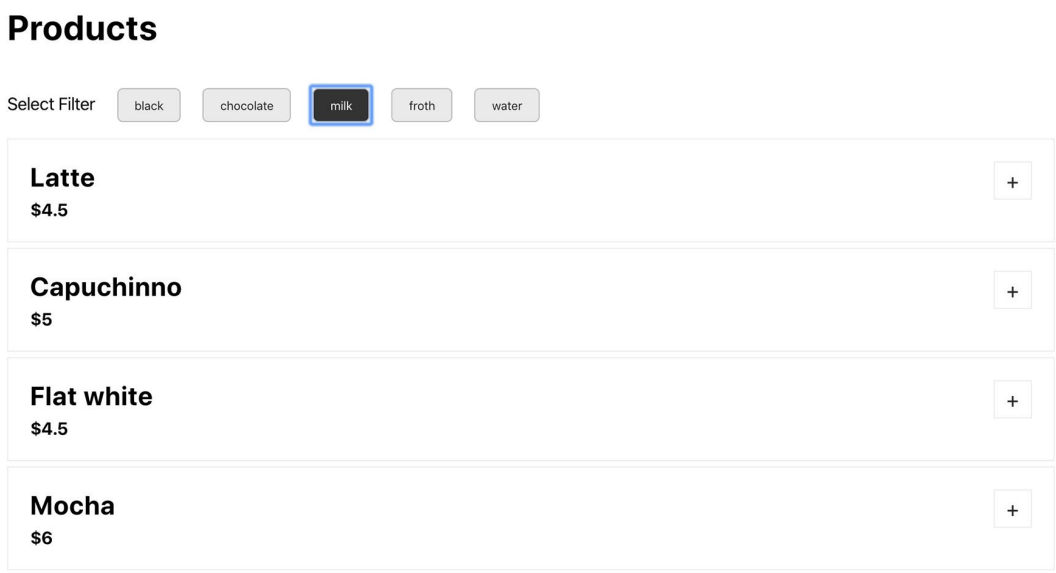

Figure 6.10: Product page showing the list of components

We can also show and hide the description of the product:

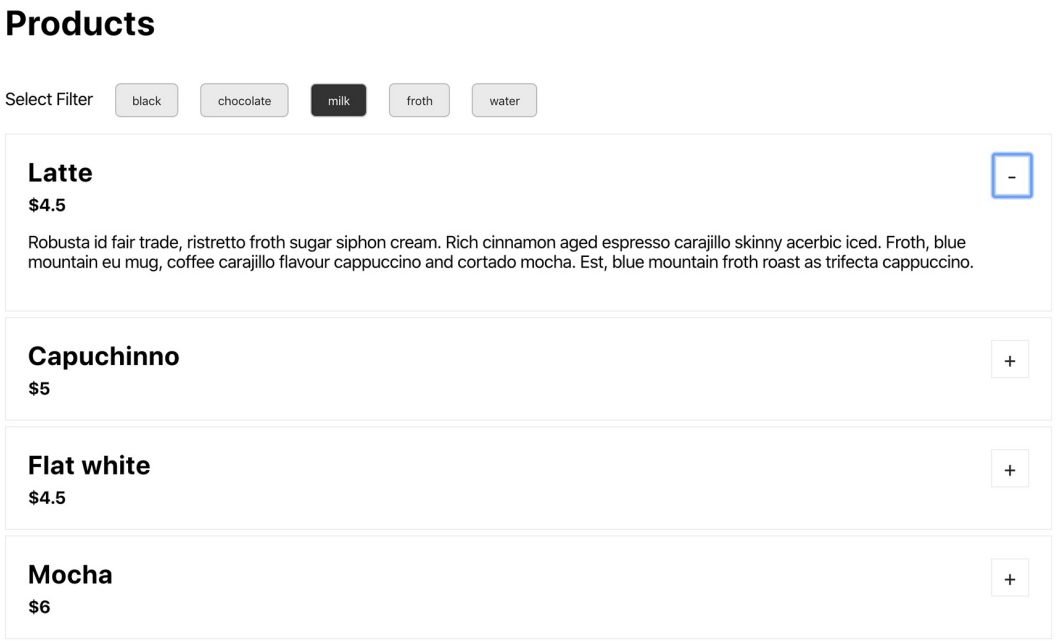

Figure 6.11: Products app component showing a description of each product

CHAPTER 7: COMMUNICATION BETWEEN COMPONENTS

ACTIVITY 7.01: CREATING A TEMPERATURE CONVERTER

Solution:

1. Run **`npx create-react-app temeprature-converter`**.
2. In the **`src`** folder, create a file called **`index.js`** and add the following code. It will import the <App> component and render it in the HTML:

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './components/App';
ReactDOM.render(<App />, document.querySelector('#root'));
```

3. Still in the src folder, create a **`components`** folder and inside the components folder, create a new file called **`App.js`**.
4. In the **`App.js`** file, let's import React and **`App.css`**. Also, add some boilerplate for the **`<App>`** component:

```
import React, { Component } from 'react';
import './App.css';
class App extends Component {
  render() {
    return (
      <div className="container">
        <h1>Temperature Converter</h1>
      </div>
    );
  }
}
export default App;
```

5. In the **`components`** folder, create a new file called **`App.css`** and add the following CSS code:

```
.container {
  text-align: center;
}
label {
  display: inline-block;
  margin: 5px 0;
```

```
}
label > span {
  display: inline-block;
  margin-right: 10px;
  min-width: 100px;
  text-align: right;
}
.status {
  font-size: 20px;
  margin: 20px 0;
}
```

6. At the top of the **<App>** components, add a constructor method and initialize the state with the **celsius** and **fahrenheit** values. We are going to set the default Celsius to **20** and call a function called **getFahrenheit** to get the Fahrenheit **number**:

```
constructor(props) {
  super(props);
  const defaultCelsius = 20;
  this.state = {
    celsius: defaultCelsius,
    fahrenheit: this.getFahrenheit(defaultCelsius)
  };
}
getFahrenheit(c) {
  return parseInt((c * 9) / 5 + 32) ;
}
```

7. Since we are going to calculate the Celsius value too, we will create the **getCelsius** function right underneath the **getFahrenheit** function:

```
getCelsius(f) {
  return parseInt(((f - 32) * 5) / 9) ;
}
```

8. Let's add the **<ConvertForm>** component right below **<h1>** in the render method. Here, we are going to send two props. The first one is **temperature**, which is where the object contains both Celsius and Fahrenheit values, while the other one is **updateTemperature**, which will be used as a callback function in the **<ConvertInput>** component:

```
render() {
  return (
    <div className="container">
      <h1>Temperature Converter</h1>
      <ConvertForm temperature={this.state}
      updateTemperature={this.updateTemperature.bind(this)} />
    </div>
  );
}}
```

9. Create the **updateTemperature** function above the render method. The function will receive two arguments. The first one is **convertTo**, which will be used to detect whether we need to convert into Celsius or Fahrenheit, while the other one is **value**, which is either a Celsius or Fahrenheit value. The value will be assigned zero as the default value to prevent it from being an empty temperature. Inside the function, we will update the state with those received values. To update the state, we will call another function that will take care of assigning the converted temperature values depending on the **convertTo** value:

```
updateTemperature(convertTo, value = 0) {
  this.setState(() => {
    if (value === '' || value === '-') {
      const returnValue = value === '-' ? '-' : '';

      return {
        celsius: returnValue,
        fahrenheit: returnValue
      };
    }
    return this.convertTemperature(convertTo, value);
  });
}
```

10. Let's create the **`convertTemperature`** function right above the **`updateTemperature`** function:

```
convertTemperature(convertTo, value) {
  return convertTo === 'celsius'
    ? { celsius: this.getCelsius(value), fahrenheit:
    parseInt(value) }
    : { celsius: parseInt(value), fahrenheit:
      this.getFahrenheit(value) };
}
```

11. Now, import **`<ConvertForm>`** at the top of the file:

```
import React, { Component } from 'react';
import ConvertForm from './ConvertForm';
import './App.css';
```

12. In the **`components`** folder, create a new file called **`ConvertForm.js`** and add some boilerplate:

```
import React from 'react';
const ConvertForm = ({ temperature, updateTemperature }) => {
  return <div>This is convert form</div>
};
export default ConvertForm;
```

13. First, we are going to add two input fields, one for Celsius and the other for Fahrenheit. Let's import the **`ConvertInput`** component, which we will create in the next step, and add two of them in **`return()`**. We are going to send the two props that we received from the **`<App>`** components, that is, temperature and the **`updateTemperature`** callback function. For the second **`<ConvertInput>`**, we are going to send one additional prop called **`convertTo="celsius"`** so that we can distinguish it from the first one:

```
import React from 'react';
import ConvertInput from './ConvertInput';

const ConvertForm = ({ temperature, updateTemperature }) => {
  return (
    <div>
      <div>
        <div>
          <ConvertInput temperature={temperature.celsius}
updateTemperature={updateTemperature} />
        </div>
```

```
        <div>
          <ConvertInput temperature={temperature.fahrenheit}
convertTo="celsius" updateTemperature={updateTemperature} />
        </div>
      </div>
    </div>
  ); }

};
export default ConvertForm;
```

14. In the **`components`** folder, create a new file called **`ConvertInput.js`** and add some boilerplate code. This component will be a class-based component:

```
import React, { Component } from 'react';
class ConvertInput extends Component {
  render() {
    return <div>This is convert input</div>;
  };
}
export default ConvertInput ;
```

15. First, we are going to add the **`<span>`** and **`<input>`** elements wrapped inside the **`<label>`** element. The **`<span>`** element will contain the title of each input field. As both Celsius and Fahrenheit will share the same **`<ConvertInput>`** component, we will check which title to display depending on the **`convertTo`** prop value. The input element will the number type and include an **`onChange`** event, which will reference to a function called **`handleTemperature`**. We will create this in the next step:

```
import React, { Component } from 'react';
class ConvertInput extends Component {
  render({ convertTo, temperature, updateTemperature } = this.props)
{
    return (
      <label>
        <span>{convertTo === 'celsius' ? 'Fahrenheit: ' : 'Celsius:
'}</span>
        <input type="number" name="temperature" value={temperature}
onChange={this.handleTemperature.bind(this)} />
      </label>
```

```
      );
    }
}
export default ConvertInput;
```

16. Let's create the handleTemperature function. This function will receive the target value and send it to the updateTemperature callback function, along with the convertTo prop value:

```
handleTemperature = event => {
  const { value } = event.target;

  this.props.updateTemperature(this.props.convertTo, value);
};
```

17. So far, we have completed the converter part. You should be able to update both the Celsius and Fahrenheit values and the other field will get updated with the converted value. Now let's complete the temperature status part. To update the status, we are going to use the Context API. Since we want to share the **`Context`** object, we will create a new file called **`StatusContext.js`** in the **`components`** folder.

18. In the **`StatusContext.js`** file, create a Context object, assign it to **`StatusContext`**, and export it. Also, create the Provider and Consumer and export them:

```
import React from 'react';
const StatusContext = React.createContext({});
export const StatusProvider = StatusContext.Provider;
export const StatusConsumer = StatusContext.Consumer;
export default StatusContext;
```

19. Now, let's add the **`Provider`** component in the **`<App>`** component. First, import **`StatusProvider`** and add it to the render method by wrapping the **`<ConvertForm>`** component. We are going to pass the Celsius value from the state:

```
import React, { Component } from 'react';
import ConvertForm from './ConvertForm';
import { StatusProvider } from './StatusContext';
import './App.css';
render() {
  return (
    <div className="container">
```

```
        <h1>Temperature Converter</h1>
        <StatusProvider value={this.state.celsius}>
          <ConvertForm temperature={this.state}
          updateTemperature={this.updateTemperature.bind(this)} />
        </StatusProvider>
      </div>
    ) ;
  }}
```

20. Let's add **<ConvertStatus>** to the **<ConvertForm>** component. Don't forget to import it:

```
import React from 'react';
import ConvertStatus from './ConvertStatus';
import ConvertInput from './ConvertInput';
const ConvertForm = ({ temperature, updateTemperature }) => {
  return (
    <div>
      <div>
        <div>
          <ConvertInput temperature={temperature.celsius}
          updateTemperature={updateTemperature} />
        </div>
        <div>
          <ConvertInput
             temperature={temperature.fahrenheit}
             convertTo="celsius"
             updateTemperature={updateTemperature}
          />
        </div>
      </div>
      <ConvertStatus />
    </div>
  );
};
export default ConvertForm;
```

21. It's time to create the **`ConvertStatus`** component. Create the **`ConvertStatus.js`** file in the **`components`** folder and add a function component. Then, import **`StatusConsumer`**.

    We are going to add the function to compare and find the status.

    Finally, return the status by receiving the props from the Consumer:

```
import React from 'react';
import { StatusConsumer } from './StatusContext';
const ConvertStatus = ({ context }) => {
  const getStatus = value => {
    if (value > 50) {
      return 'Very hot';
    } else if (value > 30) {
      return 'Hot';
    } else if (value > 15) {
    return 'Warm';
    } else if (value > 0) {
    return 'Cool';
    } else if (value > -10) {
    return 'Cold';
    } else if (value <= -10) {
    return 'Very cold';
    }

    return null;
  };

  return <StatusConsumer>{props => <div className="status">Status:
{getStatus(props)}</div>}</StatusConsumer>;
};
export default ConvertStatus;
```

    That's it. Now, when you update either Celsius or Fahrenheit, the other input field will be updated with the converted value and the status will get updated, depending on the Celsius value.

22. Let's update the **<ConvertStatus>** component with **HOC**. To do so, we need to move the Consumer to **<ConvertForm>**. In **ConvertForm.js**, let's import the Consumer. Let's create a HOC function called **withStatus**. The HOC function will take the **ConvertStatus** component as an argument and assign the props back to **ConvertStatus**.

    Now, create the wrapper component with the HOC. Create it so that it's above the **ConvertForm** component:

```
import React from 'react';
import ConvertStatus from './ConvertStatus';
import ConvertInput from './ConvertInput';
import { StatusConsumer } from './StatusContext';
const withStatus = WrappedComponent => {
  return () => {
  return <StatusConsumer>{props => <WrappedComponent context={props}
/>}</StatusConsumer>;
  };
};
const WrapperComponent = withStatus(ConvertStatus);
const ConvertForm = ({ temperature, updateTemperature }) => {
  ... };
```

23. Replace <**ConvertStatus>** with the **<WrapperComponent>** component in **return()**.

    In the **<ConvertStatus>** component, receive **context** as an argument and return the JSX element without the **Consumer** component:

```
import React from 'react';
// import { StatusConsumer } from './StatusContext';
const ConvertStatus = ({ context }) => {
  const getStatus = value => {
    …

    return null;
  };
  // return <StatusConsumer>{props => <div className="status">Status:
{getStatus(props)}</div>}</StatusConsumer>;
  return <div className="status">Status: {getStatus(context)}</div>;
};
export default ConvertStatus;
```

As you can see, we have removed **`Consumer`** from the **`ConvertStatus`** component. Now, we can only keep the logic and JSX element related to what **`ConvertStatus`** really does.

If we run the preceding code, the output will be as follows:

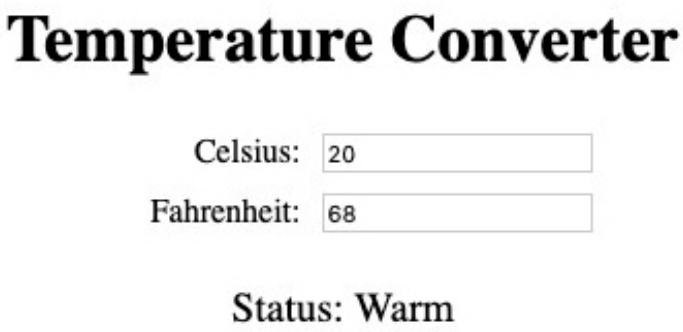

Figure 7.27: Temperature Converter app

CHAPTER 8: INTRODUCTION TO FORMIK

ACTIVITY 8.01: WRITING YOUR OWN FORM USING FORMIK

Solution:

Let's implement the tasks one by one:

1. Create a new file named **`UserRegistrationForm.js`** in the **`src`** folder.
2. Add the following code to the file **`App.js`**:

```
import React, { Component } from 'react';

import { Formik, Field} from 'formik';

class App extends Component {
  render() {
    return (
    <div className="App">
    <Formik
      initialValues={{
      name: '',
      password: '',
      passwordMatch: '',
      email: '',
      acceptTAC: false
      }
      }>
      {({
        handleSubmit,
        handleChange
      }) => (
      <form onSubmit={handleSubmit}>
      </form>
      )}
    </Formik>
    </div>
  );
  }
```

```
}

export default App;
```

3. Create a new name field inside the **`form`** component:

```
<label>
  Name:
</label><br/>
<Field type="text" name="name"
onChange={handleChange} /><br/>
```

4. Create a password field inside the **`form`** component:

```
<label>
  Password:
</label><br/>
<Field type="password" name="password"
onChange={handleChange} /><br/>
```

5. Create a **`Password Match`** field. This should match the first password field for the form to validate. If it does not match, it should display an error message:

```
<label>
  Password Match:
</label><br/>
<Field type="password" name="passwordMatch"
onChange={handleChange} /><br/>
```

6. As for the matching case, because we need access to a different field to perform the comparison, we can use the validate property only for that field:

```
const validate = (values) => {
  let errors = {};
  if (values.password !== values.passwordMatch) {
  errors.passwordMatch = 'Passwords must match';
  }

  return errors;
};
...
<Formik
   validate={validate}
   ...
```

7. Create an **`email`** field inside the form component:

```
<label>
  Email:
</label><br/>
<Field type="text" name="email"
onChange={handleChange} /><br/>
```

8. Create a **`checkbox`** field for accepting the terms and conditions:

```
<label>
  Accept Terms and Conditions:
</label>
<Field type="checkbox" name="acceptTAC"
      onChange={handleChange} /><br/><br/>
```

9. Add a Register button:

```
<input type="submit" value="Register" />
```

10. Add necessary error messages next to or below each field. We use the ErrorMessage component for each field:

```
import { Formik, Field, ErrorMessage } from 'formik';
<label>
  Name:
</label><br/>}

<Field type="text" name="name"
onChange={handleChange} /><ErrorMessage name="name"/><br/>
...
```

11. Add the validation rules for the **`Username`** field. We can use **`Yup`** for this:

```
import * as Yup from 'yup';

const RegisterSchema = Yup.object().shape({
  name: Yup.string()
  .min(6, 'Username too short!')
  .max(24, 'Username too long!')
  .required('Required'),
});
<Formik
  initialValues={{
  name: '',
```

```
    password: '',
    passwordMatch: '',
    email: '',
    acceptTAC: false
    }}
    validateOnChange={false}
    validationSchema={RegisterSchema}}
  >
```

12. Add the validation rules for the password field:

```
  password: Yup.string()
    .min(8, 'Password too short!')
    .required('Required')
    .matches(/(?=.*[a-z])(?=.*[A-Z])(?=.*\d)/, 'At least one lowercase
  ,one uppercase and a digit is required')
```

    The regex involved here is:

    `(?=.*[a-z])`: Check if at least one lowercase letter exists.

    `(?=.*[A-Z])`: Check if at least one uppercase letter exists.

    `(?=.*\d)`: Check if at least one digit exists.

13. Add the validation rules for the email field. We can use a custom test example to validate the uniqueness of the email. Here is an example:

```
  email: Yup.string()
    .email('Valid Email required')
    .required('Required')
    .test('Email unique', 'Email already registered!', (value) => {
    return sleep(200).then(() => {
  // Test emails to check for uniqueness. This should be done in the
  backend server instead. }

      if (['vganesh@mac.com', 'guialbu@msn.com',
          'nasarius@optonline.net', 'scato@icloud.com',
          'dwheeler@optonline.net'].includes(value)) {
      return Promise.resolve(false);
      } else {
      return Promise.resolve(true);
      }
```

```
    })
    })
// Async Validation
const sleep = ms => new Promise(resolve => setTimeout(resolve,
     ms)); }
```

Did you notice the glitch when you perform async validation on a field that has a delay in returning the results? For example, try to increase the sleep period to 1000 ms and make a field fail their validation checks. What do you see? How can we overcome this problem? Let's look at the following steps.

14. Add the following validation rules for the checkbox:

```
acceptTAC: Yup.boolean()
    .test('TAC Accepted', 'TAC needs to be accepted', (value)
     => !!value) }
```

15. When we submit the form, add a 200ms delay to simulate the backend processing. During that time, ensure that the Register button is disabled.

    We add the **`onSubmit`** handler in the Register button and we pass the **`isSubmitting`** prop to the view. To simulate a delay, we use the **`setSubmitting`** callback after **`200`** ms:

`App.js`

```
<Formik
     initialValues={{
     name: '',
     password: '',
     passwordMatch: '',
     email: '',
     acceptTAC: false
     }}
     validate={validate}
     onSubmit={(values, { setSubmitting }) => {
     setTimeout(() => {
       setSubmitting(false);
     }, 200);
     }}
     validateOnChange={false}
     validationSchema={RegisterSchema}
   >
```

`The complete code can be found here` https://packt.live/2T2tBuU

The output is as follows:

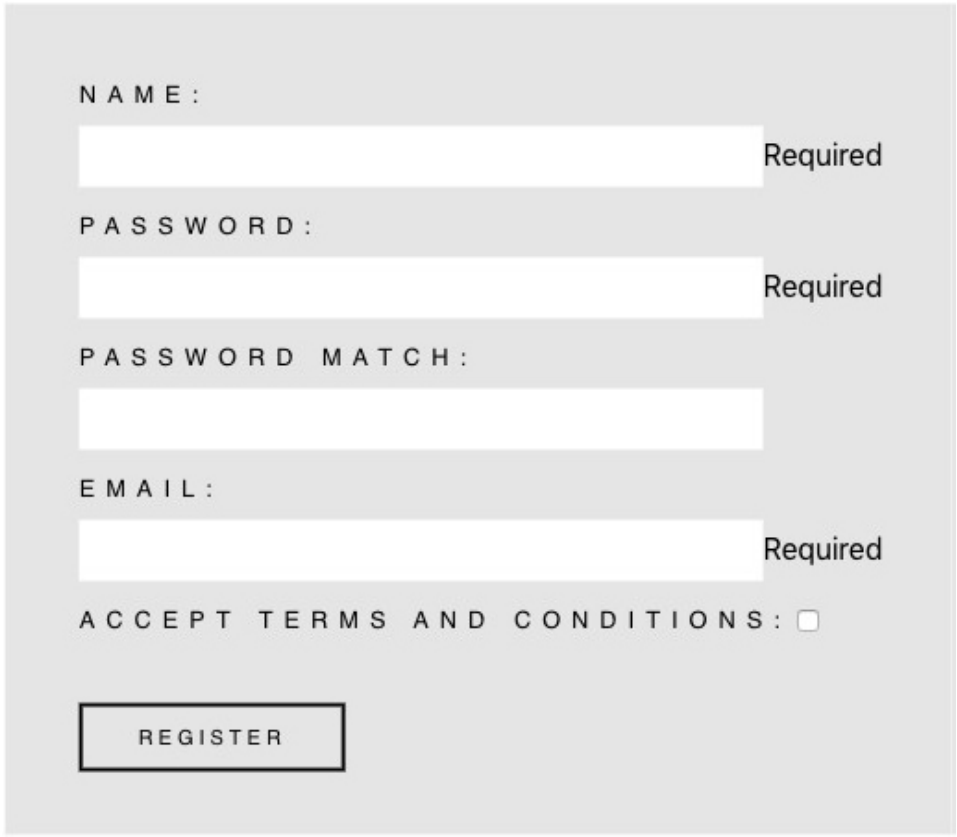

Figure 8.11: Final output

CHAPTER 9: INTRODUCTION TO REACT ROUTER

ACTIVITY 9.01: CREATING AN E-COMMERCE APPLICATION

Solution:

1. Start off by creating your new project; delete **`logo.svg`** and **`App.css`**, and then clear out the contents of App.js:

```
$ npx create-react-app router-store
```

2. Include the React Router imports at the top of **`App.js`**. You will need to include the **`Router`**, **`Switch`**, **`Route`**, and **`Link`** components. You will also need to include a couple of utility functions such as **`useRouteMatch`**:

```
import { BrowserRouter as Router, Switch, Route, Link, useRouteMatch
} from 'react-router-dom';
```

3. Build the **`App`** component to begin with. Wrap it in a Router component and write your **`Switch`** statement. You can flesh out the routes later:

```
const App = () => (
  <Router>
  <div className="App">
    <h1>My Store</h1>
    <hr />
    <Switch></Switch>
  </div>
  </Router>
);
```

Our app should look like this initially:

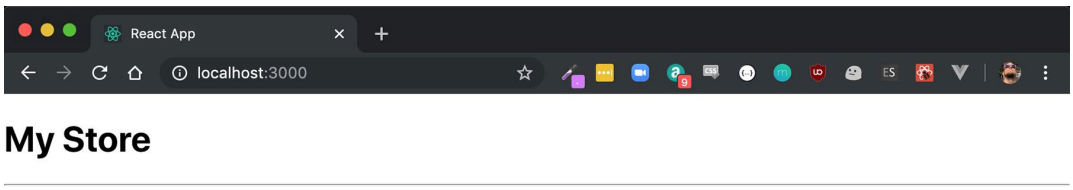

Figure 9.11: The MyStore component

4. Now set up an initial inventory for your store. You can declare this as a variable in **`src/App.js`** and work on it from there:

```
let Inventory = [
  {
  id: 1,
  name: 'Shoes',
  description: 'Some shoes you can buy'
},
  {
  id: 2,
  name: 'Backpack',
  description: 'This backpack can fit so much inside of it'
  },
  {
  id: 3,
  name: 'Travel Mug',
  description: 'A travel mug. Fill it with liquids.'
  }
];
```

5. Now, let's build a **`Store`** functional component. You will need to set up the URL and path from **`useRouteMatch`**() in this function and pass the URL down to the small item display so that it can use that to build out the route for each inventory item:

```
const Store = () => {
  const { path, url } = useRouteMatch();
  return (
  <div className="Store">
    <h2>Items for Sale</h2>
    <hr />
    {Inventory.map(i => (
      <ItemSmall {...i} baseUrl={url} />
    ))}
  </div>
  );
};
const ItemSmall = props => (
```

```
    <div className="Item" key={`item-${props.id}`}>
    <Link to={`${props.baseUrl}/items/${props.id}`}>{props.name}</Link>
    </div>
  );
```

6. Build a quick **`Homepage`** component that will serve as the **`root`** component when a user visits the page. It doesn't need to be complicated:

```
const Homepage = () => <h1>Welcome to my app.</h1>;
```

7. Add in the **`Store`** and **`Homepage`** components to the routes for our app. Use the exact parameter on the **`Homepage`** (the **`root`** component) route to make sure it doesn't encompass all of the other routes:

```
const App = () => (
  <Router>
  <div className="App">
    <h1>My Store</h1>
    <hr />
    <Switch>
      <Route exact path="/">
  <Homepage />
  </Route>
  <Route path="/store">
        <Store />
      </Route>
    </Switch>
  </div>
  </Router>
);
```

8. Add a navigation bar above our **`Switch`** component, in the same way that we did in the previous exercises:

```
      <Link to="/">Home</Link>
       
      <Link to="/store">Store</Link>
      <hr />
```

When rendered, our app should now look similar to this:

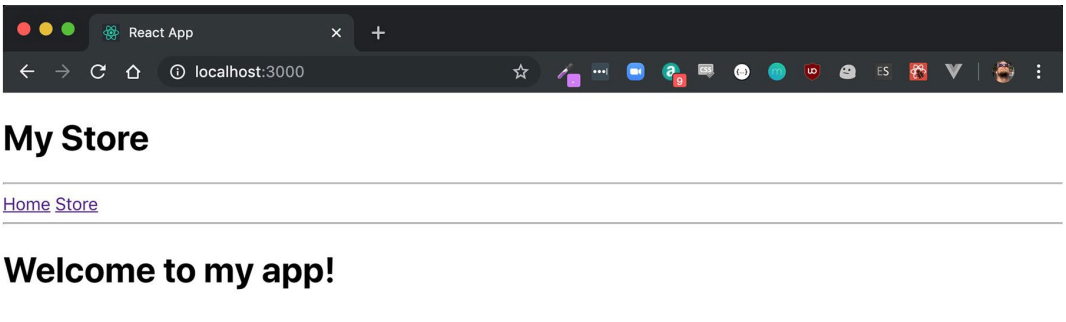

Figure 9.12: The MyStore basic component

9. Now, let's add a **`NotFound`** component that displays a **`We're sorry, we couldn't find that page`** message to the user:

```
const NotFound = () => (
  <div className="NotFound">
  <h2>We're sorry, we couldn't find that page.</h2>
  </div>
);

const App = () => (
  <Router>
  <div className="App">
    <h1>My Store</h1>
    <hr />
    <Link to="/">Home</Link>
     
    <Link to="/store">Store</Link>
    <hr />
    <Switch>
      <Route exact path="/">
        <Homepage />
      </Route>
      <Route path="/store">
        <Store />
      </Route>
      <Route path="*">
        <NotFound />
      </Route>
```

```
      </Switch>
    </div>
    </Router>
  );
```

Here is the actual **`404`** error displayed by the app:

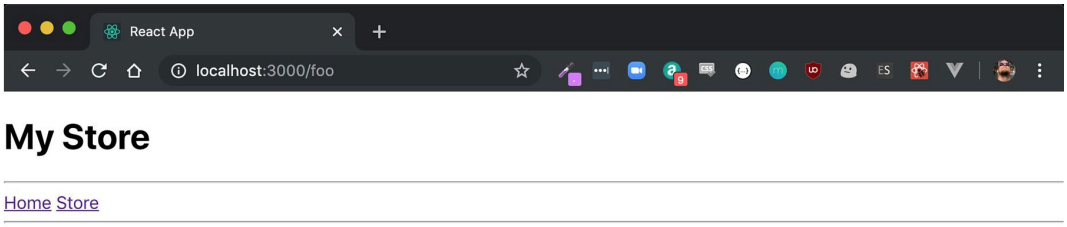

Figure 9.13: The error component

10. Set up the nested routing for the Store component. This should have a couple of requirements: if no item id is specified, display all the components using the minimized item display. If an item id is specified and the item exists, display the larger version of that item. You will need to research and use a parameterized URL for this. Alternatively, if an item id is specified and the item does not exist, display an **`ItemNotFound`** message. If a bad nested route is specified, display an **`ItemNotFound`** message.
11. Here is the code for the **`Store`** component:

```
const Store = () => {
  const { path, url } = useRouteMatch();
  return (
  <Router>
    <div className="Store">
      <h2>Items for Sale</h2>
      <hr />
      <Switch>
        <Route exact path={`${path}`}>
          {Inventory.map(i => (
            <ItemSmall {...i} baseUrl={url} />
          ))}
        </Route>
        <Route path={`${path}/items/:itemId`}>
          <Item />
```

```
          </Route>
          <Route path="*">
            <ItemNotFound />
          </Route>
        </Switch>
      </div>
    </Router>
    );
};
```

12. Create the code for when the item is not found:

```
const ItemNotFound = () => (
  <div className="ItemNotFound">
  <h2>We're sorry, we could not find that item in our
     store.</h2>
  </div>
);
```

13. Now, let's set up a component to display all of the details for an inventory item, which we will use in the next step. It should display the id, name, and description for the specified item. This item will need to use the previous parameterized property of **`itemId`**:

```
const Item = () => {
  let { itemId } = useParams();
  const [item, ...rest] = Inventory.filter(i =>
    i.id.toString() === itemId);
  if (item) {
  return (
    <div className="Item">
      <p>
        <strong>id:</strong> {item.id}
      </p>
      <p>
        <strong>name:</strong> {item.name}
      </p>
      <p>
        <strong>description:</strong> {item.description}
      </p>
    </div>
  );
  } else {
```

```
      return <ItemNotFound />;
      }
    };
```

Here is the final code for the **`Store`** component:

`App.js`

```
25 const Store = () => {
26  const { path, url } = useRouteMatch();
27  return (
28  <Router>
29    <div className="Store">
30      <h2>Items for Sale</h2>
31      <hr />
```

`The complete code can be found here` https://packt.live/2yVjVM5

And that's it! If you visit the store and click on one of the items in the list, you should see all of the details show up for that inventory instead of the minimized **`Inventory`** item display:

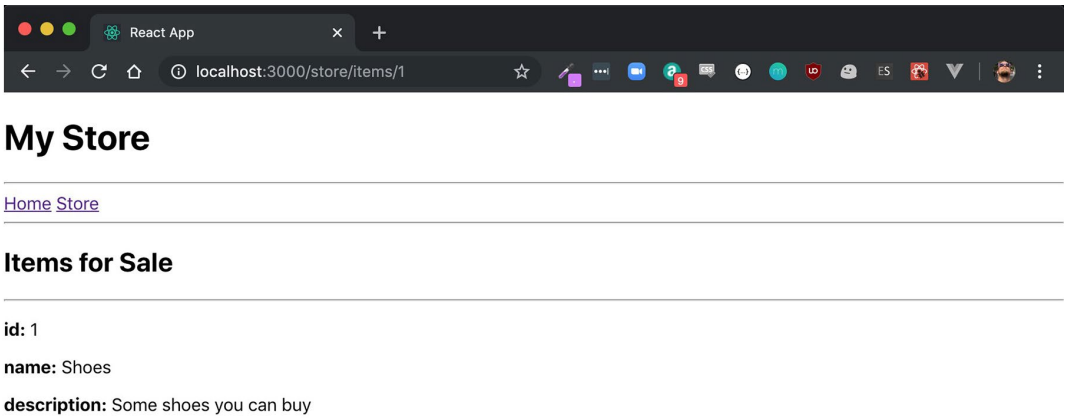

Figure 9.14: The MyStore component

CHAPTER 10: ADVANCED ROUTING TECHNIQUES: SPECIAL CASES

ACTIVITY 10.01: CREATING AUTHENTICATION USING ROUTING TECHNIQUES

Solution:

> **NOTE**
>
> The code for this activity can be found here: https://packt.live/3bAUFrz.

Let's implement the tasks one by one:

1. Create a new React app:

```
$ npx create-react-app [name]
```

2. Go to the **`src/App.js`** file, delete **`logo.svg`** and **`App.css`**, and clear out the contents of **`App.js`**.
3. Inside **`App.js`**, create a new React functional component called **`App`**.
4. Install the required dependencies:

```
$ npm install formik react-router-dom
```

5. Create a 404 page. Using what we have learned about **`404`** pages, we will define our main application with one route:

```
import React from 'react';
import {
  BrowserRouter as Router,
  Route,
  Switch,
  Redirect,
  useLocation
} from 'react-router-dom';
import './App.css';


function App() {
  return (
    <div className="App">
      <Router>
```

```
            <Switch>
              <Route path="/404">
                <NoMatch />
              </Route>
              <Redirect to="/404" />
            </Switch>
          </Router>
        </div>
      );
    }

    function NoMatch() {
      return <h3>404 Sorry!</h3>;
    }
```

6. Create a **`Login`** page. We can use Formik to quickly define our Login Form, like so:

`LoginForm.js`

```
1 import React from 'react';
2 import { Formik } from 'formik';
3 import { Redirect,  } from 'react-router-dom';
4 import authService from './AuthService';
5
6 const LoginForm = () => (
7  <div>
8    <h1>Login</h1>
9    <Formik
10      initialValues={{ email: '', password: '' }}
11      initialStatus={{isRedirectToVerifyPage: false}}
12      validate={values => {
13        const errors = {};
14        if (!values.email) {
15          errors.email = 'Email is Required';
16        }
```

`The complete code can be found here` https://packt.live/2WSRlxv

7. Then, we need to add it to the routes list inside the **`App.js`** file:

```
    import LoginForm from './LoginForm';
    …
    <Route exact path="/">
        <Login />
     </Route>
    …
    function Login() {
      return <LoginForm />;
```

```
}

export default LoginForm;
```

8. Use the **`Auth Service`** to verify the password is correct. Create a new file named **`AuthService.js`** and add a basic implementation that will check the user's email and password from a hardcoded list:

```
const backendUsers = {
  'theo@gmail.com': {
    password: '123',
    token: '321'
  },
  'alex@gmail.com': {
    password: 'admin',
    token: '999'
  },
  'john@gmail.com': {
    password: 'lalala',
    token: '651'
  }
};

export class AuthService {
  isValidPassword(email, password) {
    if (backendUsers[email] && backendUsers[email].password
    === password) {
      return true
    }
    return false
  }
}

export default new AuthService();
```

9. In the **`Login`** component, we need to add the following check in the **`onSubmit`** handler:

```
onSubmit={(values, { setSubmitting, setStatus }) => {
  setTimeout(() => {
    if (authService.isValidPassword(values.email, values.password)) {
      setStatus({isRedirectToVerifyPage: true});
    }
    setSubmitting(false);
  }, 400);
}}
```

10. We also need to add the following property in Formik:

```
initialStatus={{isRedirectToVerifyPage: false}}
```

11. Navigate to the Verify Token page while passing the email as a parameter. When the preceding check is valid, **`isRedirectToVerifyPage`** will become **`true`**, which means we can conditionally render a **`<Redirect>`** component by passing the email as a parameter:

```
{
  status.isRedirectToVerifyPage ? <Redirect to={{
    pathname: "/verify",
    search: `?email=${values.email}`,
     }} /> : null
}
```

12. Create a Verify Token page and verify that the email parameter is passed. All we need to do is create a new component called **`VerifyTokenForm`** and pass the email as a parameter. The form will only have one field for the security token:

`VerifyTokenForm.js`

```
6 const VerifyTokenForm = ({email}) => (
7  <div>
8    <h1>Verify Token</h1>
9    <Formik
10      initialValues={{ email: email, token: '' }}
11      initialStatus={{isRedirectToDashboardPage: false}}
12      validate={values => {
13        const errors = {};
14        if (!values.email) {
15          errors.email = 'Email is Required';
16        }
```

`The complete code can be found here` https://packt.live/33NxKct

13. Now, when we register a new route, we can retrieve the URL parameter for the email and validate that it exists. If not, we redirect the user back to the login page. Add the following code to the **`App.js`** file:

```
import VerifyTokenForm from './VerifyTokenForm';
…
<Route path="/verify">
   <Verify />
</Route>
…
function useQuery() {
  return new URLSearchParams(useLocation().search);
}
function Verify() {
  const query = useQuery();
  const location = useLocation();
  const email = query.get('email');
  if (!email) {
    return <Redirect to={{
      pathname: "/",
      state: {from: location}
    }} />
  }
  return <VerifyTokenForm email={email}/>;
}
```

14. Set that the user is authenticated and navigate to the **`Dashboard`** page. Here, we will use the same technique from the Verify Token page to redirect the user to the **`Dashboard`** view. Add the following code inside the **`VerifyTokenForm`** Formik component:

```
{
  Status.isRedirectToDashboard ? <Redirect to={{
    pathname: "/dashboard",
  }} /> : null
}
```

15. Create a protected route for the **`Dashboard`** page. To do so, we can reuse the **`IsAuthenticatedRoute`** component that we defined in the previous section and copy all the code for the Dashboard component we took from the nested routes section:

```
const IsAuthenticatedRoute = ({ component: Component, ...rest }) => (
  <Route {...rest} render={(props) => (
    authService.isAuthenticated() === true
      ? <Component {...props} />
      : <Redirect to={{
        pathname: '/',
        state: { from: props.location }
      }} />
  )} />
);
```

16. We also need to define the **`isAuthenticated`** method in the **`authService`**. Add the following methods there:

```
constructor() {
    this._isAuthenticated = false;
  }
setIsAuthenticated() {
    this._isAuthenticated = true;
  }
isAuthenticated() {
    return this._isAuthenticated;
}
```

Typically, the authentication check will happen with the backend. In the frontend, we would only pass around an authentication token that carries the authentication status. For now, we just use a local property for convenience.

We also need to call **`setIsAuthenticated`** just before we redirect to the dashboard screen in the **`<VerifyTokenForm>`** component. We are doing this since the protected route will bounce us back.

17. Add an **`Unknown Widget`** page. We need to modify the **`<Dashboard>`** component to explicitly check the valid routes for each widget. Here is the **`Dashboard.js`** component:

```
import React from 'react';
import {Link, useRouteMatch, Switch, Route, useParams, Redirect} from
'react-router-dom';

function Dashboard() {
  let { path, url } = useRouteMatch();
  return (
    <div className="row">
      <div className="sidebar">
        Sidebar
      </div>
      <div className="main">
        <div className="navbar">
          <Link to={`${url}/map`}>Map</Link>
          <Link to={`${url}/chart`}>Chart</Link>
          <Link to={`${url}/table`}>Table</Link>
        </div>
        <Switch>
          <Route exact path={path}>
            <h3>Please select a widget.</h3>
          </Route>
          <Route path={`${path}/unknown`}>
            <UnknownWidget />
          </Route>
          <Route path={`${path}/:widgetName`}>
            <Widget />
          </Route>
        </Switch>
      </div>
    </div>
  )
}

export default Dashboard;
```

18. Then, in the **<Widget>** component, we need to check if the widget is valid and known. If not, we redirect the user to the **UnknownWidget** page:

```
function Widget() {
  let { widgetName } = useParams();
  let { path } = useRouteMatch();
  if (!['map', 'chart', 'table'].includes(widgetName)) {
    return <Redirect to={path.replace(':widgetName', 'unknown')} />
  }

  return (
    <div>
      <h3>Widget: {widgetName}</h3>
    </div>
  );
}

function UnknownWidget() {
  return (
    <div>
      <h3>Unknown Widget</h3>
    </div>
  );
}
```

This way, we can handle known and unknown widgets in nested routes.

Here is the complete code for the App.js component:

App.js

```
39 function useQuery() {
40  return new URLSearchParams(useLocation().search);
41 }
42
43 function Login() {
44  return <LoginForm />;
45 }
46
47 function Verify() {
48  const query = useQuery();
49  const location = useLocation();
50  const email = query.get('email');
```

The complete code can be found here https://packt.live/3fPQqMc

The output is as follows:

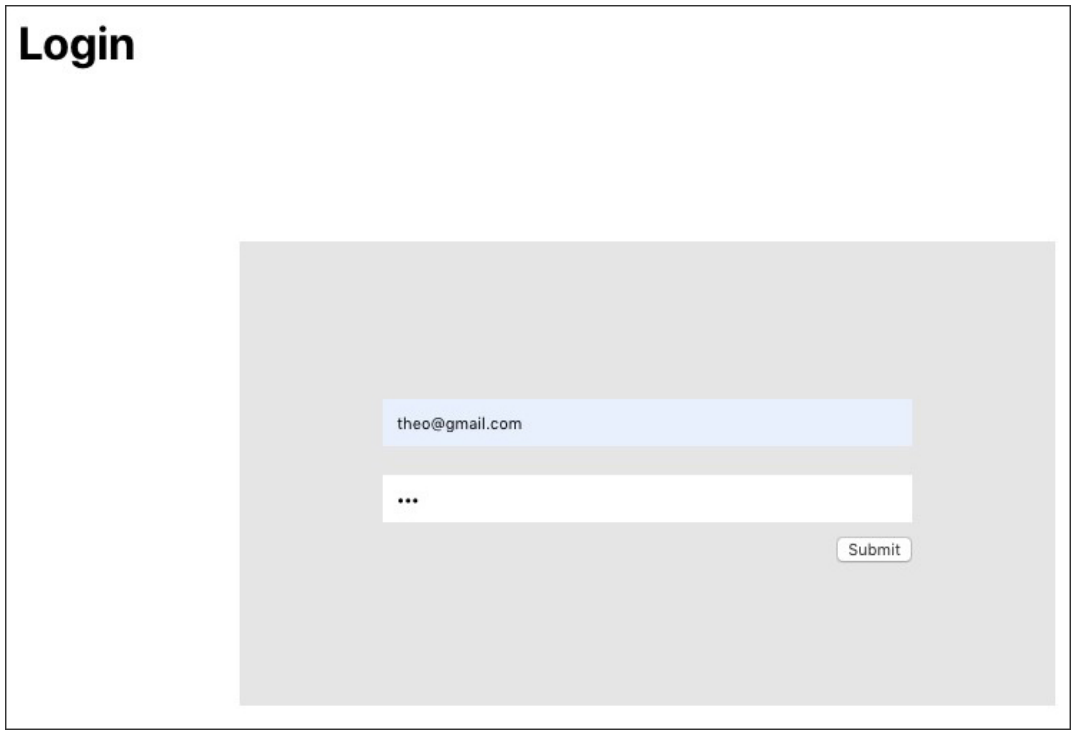

Figure 10.7: Login form app

The following is a screenshot of the Verify Token page:

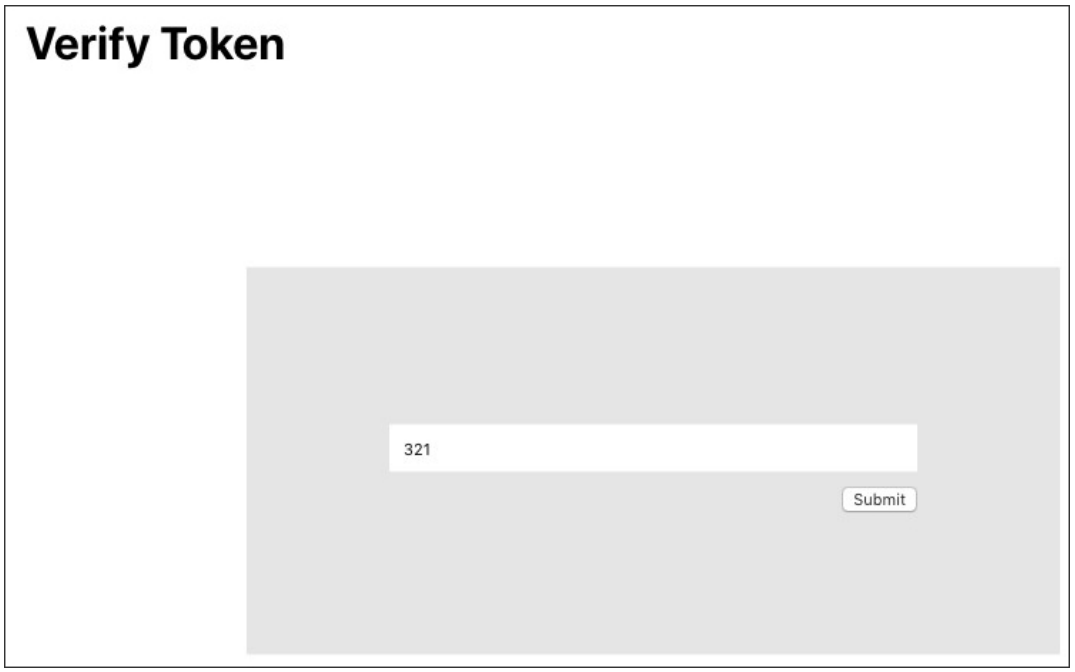

Figure 10.8: Verify token page

The following is a screenshot of the web page displaying the sidebar:

Figure 10.9: App showing the widget map

CHAPTER 11: HOOKS – REUSABILITY, READABILITY, AND A DIFFERENT MENTAL MODEL

ACTIVITY 11.01: CREATING A REUSABLE COUNTER

Solution:

1. Start off by creating a new React project, which we will call counter-hooks, start the project, and take a look at the browser window that should have opened for you automatically:

```
$ npx create-react-app counter-hooks
$ cd counter-hooks
$ yarn start
```

2. Delete **`src/logo.svg`**.

3. Replace the contents of **`src/App.css`** with the following:

```
body {
  margin: 20px;
}

button {
  width: 200px;
  height: 50px;
  background: #4444ff;
  color: white;
  font-weight: bold;
  border: none;
  cursor: pointer;
  margin: 5px;
}
```

4. Replace the contents of **`src/App.js`** with the following:

```
import React from "react";
import "./App.css";
const App = () => (
  <div className="App">
  <h1>Counter</h1>
```

```
    </div>
  );
  export default App;
```

5. Since we want to be able to control our component via props, let's first imagine what it should look like. Let's create a class that does not render anything; only logic lives inside there. Create a new file, **`src/Counter.js`**, and give it the following body:

```
import { Component } from "react";
class Counter extends Component {
  state = { value: 0 };
  render() {
    return null;
  }
}
export default Counter;
```

6. Return to **`src/App.js`**, import the **`Counter`** component, and add it to your **`App`** component:

```
import React from "react";
import "./App.css";
import Counter from "./Counter";
const App = () => (
  <div className="App">
  <h1>Counter</h1>
  <Counter />
  </div>
);
```

7. Now, return to **`src/Counter.js`** and create the functions that are going to control the state and add the ability, so the state is set via props (**`initialValue`**):

```
class Counter extends Component {
  state = { value: this.props.initialValue };
  increment = () => {
this.setState({ value: this.state.value + 1 });
  };
  decrement = () => {
this.setState({ value: this.state.value - 1 });
```

```
  };
  reset = () => {
this.setState({ value: this.props.initialValue });
  };
  render() {
  return null;
  }
}
```

8. Return to **`src/App.js`**, and we will give the **`initialValue`** prop to our Counter component:

```
import React from "react";
import "./App.css";
import Counter from "./Counter";
const App = () => (
  <div className="App">
  <h1>Counter</h1>
  <Counter initialValue={7} />
  </div>
);
export default App;
```

9. Create a **`CounterView`** component where there is no state so we can later glue it with our render props pattern. We will create this as **`src/CounterView.js`**:

```
import React from "react";
const CounterView = props => (
  <div className="CounterView">
  <div>The current value is: {props.value}</div>
  <div>
    <button onClick={props.increment}>Click here to increment
     it</button>
    <button onClick={props.decrement}>Click here to decrement
     it</button>
    <button onClick={props.reset}>Click here to reset
     it</button>
  </div>
  </div>
);
export default CounterView;
```

10. Next, back in **`src/Counter.js`**, update the render function to use the **`View`** component and pass in the increment, decrement, reset, and value properties in the class. Notice as we build this that now we have to pass multiple functions, all bound to the Counter component, down to its children, to be able to separate the concept of a **`View`** layer from our **`Counter`**. Note that we are using JSX now, so we do need to include the React **`import`** this time:

```
import React, { Component } from "react";
import CounterView from "./CounterView";
import Logger from "./Logger";
class Counter extends Component {
  state = { value: this.props.initialValue };
  increment = () => {
this.setState({ value: this.state.value + 1 });
  };
  decrement = () => {
this.setState({ value: this.state.value - 1 });
  };
  reset = () => {
this.setState({ value: this.props.initialValue });
  };
  render() {
  return (
    <Logger watch={this.state.value}>
      <CounterView
        increment={this.increment}
        decrement={this.decrement}
        value={this.state.value}
        reset={this.reset}
      />
    </Logger>
  );
  }
}
export default Counter;
```

11. Let's now tackle logging functionality. As we said in the description of this activity, we also want to have a logger component. We create another component called **`<Logger />`** that will have a **`watch`** prop. We will provide a value to that **`watch`** prop, and when we detect that the prop has changed, we will call a **`console.log`** statement and let the user know that the value has changed. Create **`src/Logger.js`** and give it the following body:

```
import { Component } from "react";
class Logger extends Component {
  componentDidUpdate(prevProps) {
  if (prevProps.watch !== this.props.watch) {
  console.log("Value changed:", this.props.watch);
  }
  }
  render() {
  return this.props.children;
  }
}
export default Logger;
```

12. Return to **`src/Counter.js`** and wrap the **`<CounterView ... />`** JSX inside the **`Counter`** component in our new **`Logger`** component:

```
import React, { Component } from "react";
import Logger from "./Logger";
import CounterView from "./CounterView";

class Counter extends Component {
  state = { value: this.props.initialValue };
  increment = () => {
  this.setState({ value: this.state.value + 1 });
  };
  decrement = () => {
  this.setState({ value: this.state.value - 1 });
  };
  reset = () => {
  this.setState({ value: this.props.initialValue });
  };
  render() {
  return (
    <Logger watch={this.state.value}>
      <CounterView
```

```
            increment={this.increment}
            decrement={this.decrement}
            value={this.state.value}
            reset={this.reset}
          />
        </Logger>
      );
      }
    }
    export default Counter;
```

This functionality now works and does what we expect it to. There are multiple buttons, and each time the value changes, the user sees a Value changed statement in the console. However, this is also incredibly messy and verbose. We can only really reuse our logger by wrapping it in a higher-order component, and that in itself can be difficult to keep track of. Plus, the logic is very heavily tied to the state of the parent component and cascades down two levels of child components, which is even harder to keep track of. This is our current UI:

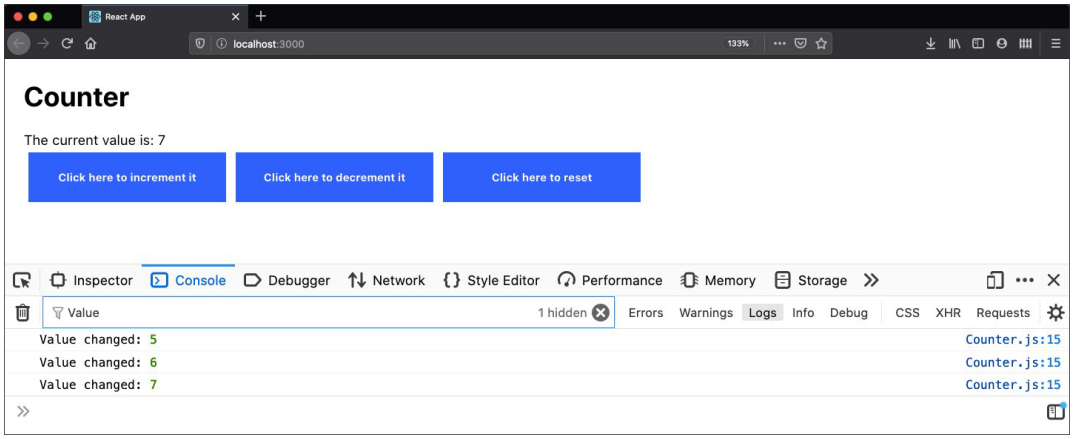

Figure 11.8: App showing the change in values

Now that the render props are complete, we can start doing our cleaner hook implementation. From the previous steps, we know what we want to return and how we want our components to function, so we will start redoing our Counter component to be entirely functional and use Hooks instead.

13. Head back to **`src/Counter.js`**, where we will create a **`useCounter`** custom hook that will set up our state variable and state setter function. Note that it not only sets up the state, it also sets up a few helper functions so that we do not have to worry about how to manipulate the state consistently:

```
const useCounter = initialValue => {
  const [value, setState] = React.useState(initialValue);
  const increment = () => setState(value + 1);
  const decrement = () => setState(value - 1);
  const reset = () => setState(initialValue);
  return { value, increment, decrement, reset };
};
```

14. Similarly, create a **`useLogger`** custom hook that will set up our **`useEffect`** hook:

```
const useLogger = value => {
  React.useEffect(() => {
console.log("Value changed:", value);
  }, [value]);
};
```

15. Now, we can finally convert our old **`Counter`** component to be a functional component using Hooks instead:

```
const Counter = props => {
  const { value, increment, decrement, reset } =
   useCounter(props.initialValue);
  useLogger(value);

  return (
  <View
       increment={increment}
       decrement={decrement}
       value={value}
       reset={reset}
  />
  );
};
```

16. You can now delete **`src/Logger.js`** in its entirety – we no longer need it. The final UI should be the same, but totally hook-based, and without a single class in sight:

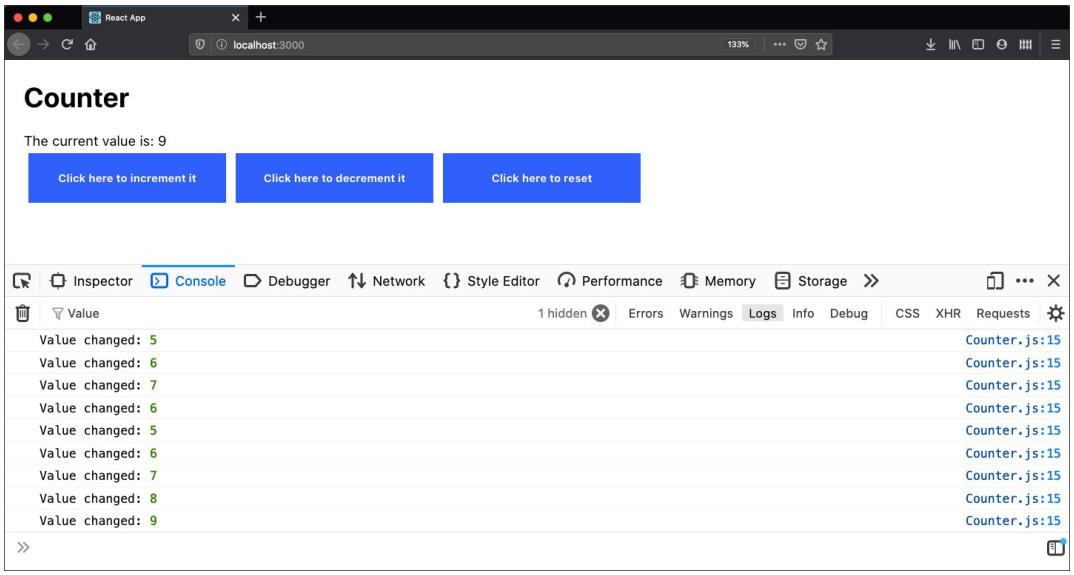

Figure 11.9: Counter app

It is worth having a look at how much work we had to put into implementing this feature with both methods and the amount of code we had to write.

CHAPTER 12: STATE MANAGEMENT WITH HOOKS

ACTIVITY 12.01: CREATING A CHAT APP USING HOOKS

Solution:

1. Start off by creating a new React app:

```
$ npx create-react-app chat-windows
```

2. Go to the **`src/App.js`** file and delete **`logo.svg`**.

3. Replace the contents of **`src/App.css`** with the following:

```
.App {
  margin: 20px;
}

button {
  width: 200px;
  height: 50px;
  background: #4444ff;
  color: white;
  font-weight: bold;
  border: none;
  cursor: pointer;
  margin: 5px;
}
```

4. Replace the contents of **`src/App.js`** to get an empty component to work with:

```
import React from "react";
import "./App.css";

const App = () => {
  return (
    <div className="App">
    <button>Add Chat Window</button>
    </div>
  );
};

export default App;
```

The first thing we want to do is start to build out our actual chat window so that our button actually shows output.

5. Create **`src/ChatWindow.js`** and then create the **`ChatWindow`** component:

```
import React from "react";
const ChatWindow = () => {
  return (
  <div>
    Chat Service is running...
    <br />
    <strong>Messages</strong>
    <hr />
    <ul></ul>
    <hr />
    <button>Close Down</button>
    <button>Clear Messages</button>
  </div>
  );
};

export default ChatWindow;
```

6. Return to **`src/App.js`** and add our **`ChatWindow`** component so that we can verify that things are working so far:

```
import ChatWindow from "./ChatWindow";

const App = () => {
  return (
  <div className="App">
    <button>Add Chat Window</button>
    <ChatWindow />
  </div>
  );
};
```

7. Set up our hooks with **`useReducer`**. Create **`src/ChatHooks.js`** and we will create a custom hook to set up **`useReducer`**. The first piece of state we will keep track of will be **`isInChat`**, which will be a boolean variable representing whether we are in a chat window.

   We will use the switch statement where we will set up two new action types: **`join`** and **`quit`**. These will set **`isInChat`** to **`true`** and **`isInChat`** to **`false`**, respectively. Then, we will need to return the **`dispatch`** and **`state`** functions:

```
import React from "react";
const useChatHook = () => {
  const [state, dispatch] = React.useReducer(
  (state, action) => {
    switch (action.type) {
      case "join":
        return {
          ...state,
          isInChat: true
        };
      case "quit":
        return {
          ...state,
          isInChat: false
        };
      default:
        return state;
    }
  },
  {
      isInChat: false
  }
  );
  return { state, dispatch };
};
export { useChatHook };
```

8. Return to **`src/App.js`** and hook up the dispatch function with the state, and display the **`ChatWindow`** component if **`state.isInChat`** is true. Also, write a few helper functions (**`quit`** and **`join`**), which will call dispatch with an action that has a type of **`quit`** and **`join`**, respectively:

```
import { useChatHook } from "./ChatHooks";

const App = () => {
  const { state, dispatch } = useChatHook();
  const quit = () => dispatch({ type: "quit" });
  const join = () => dispatch({ type: "join" });
  return (
  <div className="App">
    <button onClick={join}>Add Chat Window</button>
    {state.isInChat && <ChatWindow close={quit} />}
  </div>
  );
};
```

Our app should currently look like this if you click **`Add Chat Window`**:

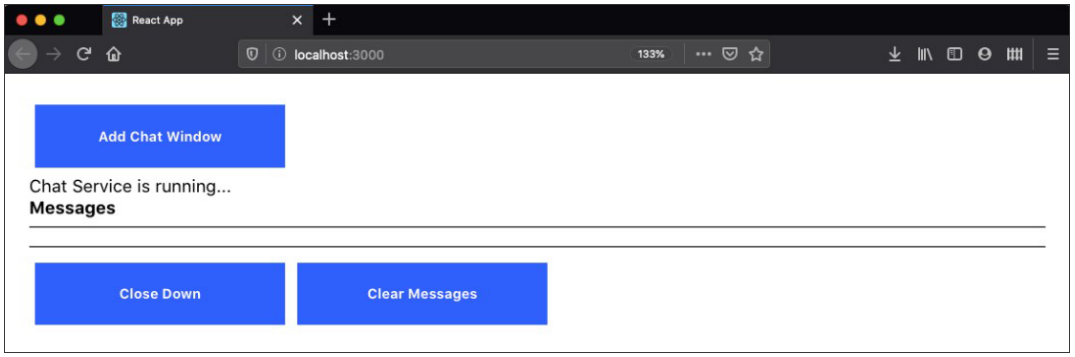

Figure 12.10: Chat Service app

9. Return to **`src/ChatWindow.js`**, add props to the **`ChatWindow`** component, and have the **`Close Down`** button call the **`close`** prop in its **`onClick`** event handler:

```
const ChatWindow = props => {
  return (
  <div>
    Chat Service is running...
    <br />
```

```
        <strong>Messages</strong>
        <hr />
        <ul></ul>
        <hr />
        <button onClick={props.close}>Close Down</button>
        <button>Clear Messages</button>
      </div>
      );
    };
```

10. Now, return to **`src/ChatHooks.js`**, where we will add a new state to handle messages, as well as new action handlers for adding and clearing messages. This will render the rest of our UI essentially functional. Messages will have a default state of an empty array:

`ChatHooks.js`

```
1 import React from "react";
2
3 const useChatHook = () => {
4  const [state, dispatch] = React.useReducer(
5  (state, action) => {
6    switch (action.type) {
7      case "add_message":
8        return {
9          ...state,
10          messages: [...state.messages, action.message]
11        };
```

The complete code can be found here https://packt.live/36a7U1H

11. Return to **`src/ChatWindow.js`**, where we will hook up our new state in **`useReducer`** to the **`ChatWindow`** component. First, import the **`useChatHook`** function from **`src/ChatHooks`**:

```
    import { useChatHook } from "./ChatHooks";
```

12. We will get the state and dispatch out of our **`useChatHook`** custom hook, and then we will set up a **`clearMessages`** function that will handle the dispatch call with the appropriate action type:

```
    const ChatWindow = props => {
      const { state, dispatch } = useChatHook();
      const clearMessages = () => dispatch({ type:
      "clear_messages" });
```

13. Finally, we will add a display to the previously empty **`<ul />`** tag for each message, which will create an **`<li />`** element with a key and the message as the text for it. We will also hook up the **`clearMessages`** function to the **`Clear Messages`** button:

```
  return (
    <div>
    Chat Service is running...
    <br />
    <strong>Messages</strong>
    <hr />
    <ul>
      {state.messages.map((msg, index) => (
        <li key={`m-${index}`}>{msg}</li>
      ))}
    </ul>
    <hr />
    <button onClick={props.close}>Close Down</button>
    <button onClick={clearMessages}>Clear Messages</button>
    </div>
  );
```

14. We won't really be able to hook anything else up until we create a special **`ChatService`** component that will be responsible for managing our subscription to chats. This will be a simple ES6 class that will store an interval and have a **`subscribe`** and **`unsubscribe`** function. Create **`src/ChatService.js`**, where we will set up our initial class definition:

```
class ChatService {
  interval = null;
}

export default ChatService;
```

15. Let's now add a **`subscribe`** function to our service. This will create an interval (which will be stored in the object's **`interval`** property), and the function it will execute on an interval will be an anonymous function that will send a **`Ping`** message to the passed function. We will also have it displayed in the console that it's joining the service:

```
class ChatService {
  interval = null;
  subscribe = fn => {
  console.log("Joining the service");
  this.interval = setInterval(() => {
      fn("Ping");
  }, 1000);
  };
}
```

16. Finally, in our **`ChatService`** component, add an **`unsubscribe`** function. This will just clear the interval and again log a message letting the user know what it's doing:

```
class ChatService {
  interval = null;
  subscribe = fn => {
  console.log("Joining the service");
  this.interval = setInterval(() => {
    fn("Ping");
  }, 1000);
  };
  unsubscribe = () => {
  console.log("Quitting the service");
  clearInterval(this.interval);
  };
}

export default ChatService;
```

17. We are now ready to wrap up this activity. Return back to **`src/ChatWindow.js`**, and we will add a **`React.useEffect`** statement that will mark **`dispatch`** as its only dependency. First, add an import statement for our **`ChatService`** component:

```
 import ChatService from "./ChatService";
```

18. When it executes, it will create a new **`ChatService`** component, call the **`subscribe()`** function on it, and return a function call to unsubscribe from the **`ChatService`** component:

```
  React.useEffect(() => {
  const chatService = new ChatService();
  chatService.subscribe(message =>
    dispatch({ type: "add_message", message })
  );
  return () => {
    chatService.unsubscribe();
  };
  }, [dispatch]);
```

The final code for our **`src/App.js`** component is as follows:

```
import React from "react";
import "./App.css";
import ChatWindow from "./ChatWindow";
import { useChatHook } from "./ChatHooks";
const App = () => {
  const { state, dispatch } = useChatHook();
  const quit = () => dispatch({ type: "quit" });
  const join = () => dispatch({ type: "join" });
  return (
  <div className="App">
    <button onClick={join}>Add Chat Window</button>
    {state.isInChat && <ChatWindow close={quit} />}
  </div>
  );
};
export default App;
```

The final code for our **`src/ChatHooks.js`** file is as follows:

`ChatHooks.js`

```
1 import React from "react";
2
3 const useChatHook = () => {
4  const [state, dispatch] = React.useReducer(
5  (state, action) => {
6    switch (action.type) {
7      case "add_message":
8        return {
9          ...state,
10          messages: [...state.messages, action.message]
11        };
```

The complete code can be found here https://packt.live/36a7U1H

The final code for our **`src/ChatService`**.js file is as follows:

```
class ChatService {
  interval = null;
  subscribe = fn => {
  console.log("Joining the service");
  this.interval = setInterval(() => {
    fn("Ping");
  }, 1000);
  };
  unsubscribe = () => {
  console.log("Quitting the service");
  clearInterval(this.interval);
  };
}

export default ChatService;
```

The final code for our **`src/ChatWindow.js`** file is as follows:

`ChatWindow.js`

```
1 import React from "react";
2
3 import { useChatHook } from "./ChatHooks";
4 import ChatService from "./ChatService";
5
6 const ChatWindow = props => {
7  const { state, dispatch } = useChatHook();
8  const clearMessages = () => dispatch({ type:
    "clear_messages" });
```

`The complete code can be found here` https://packt.live/3cJ1p8x

And now, our final chat app should appear as follows (matching the screenshots at the start of this activity):

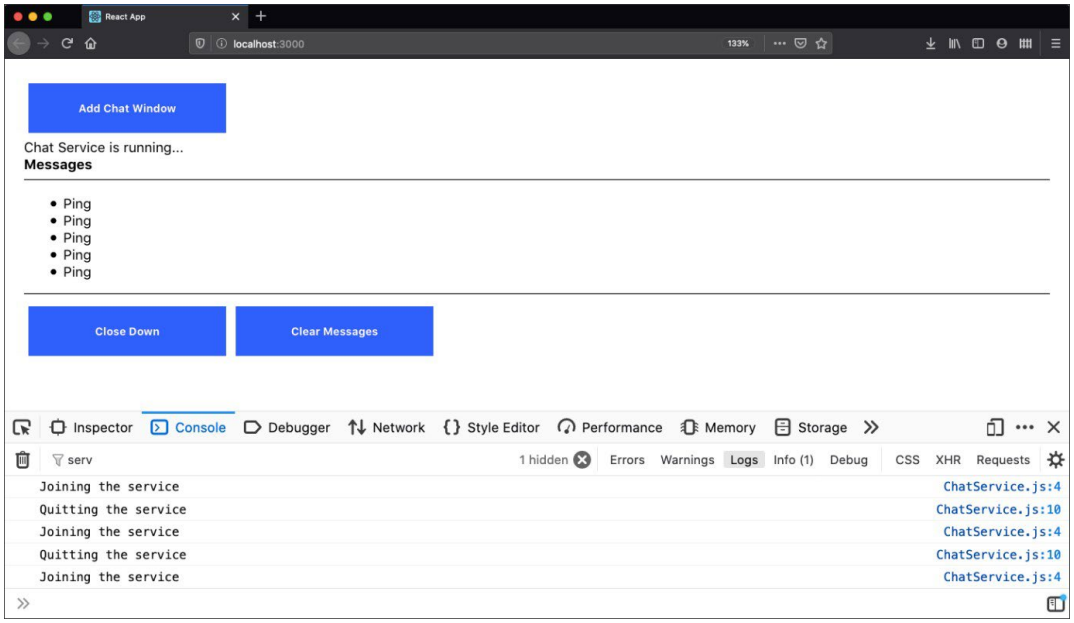

Figure 12.11: Chat window

CHAPTER 13: COMPOSING HOOKS TO SOLVE COMPLEX PROBLEMS

ACTIVITY 13.01: CREATING A RECIPE APP

Solution:

1. Start off by creating a new React app:

```
$ npx create-react-app recipes
```

2. Go to the **`src/App.js`** file and delete **`logo.svg`**. Clear out the contents of **`App.js`** and **`App.css`**. Instead, replace the contents of **`src/App.css`** with the following:

```
.App {
  display: flex;
  justify-content: space-between;
}
.App > div {
  padding: 20px;
}
```

3. Start off with an initial component in **`src/App.js`**:

```
import React from "react";
import "./App.css";
const App = () => {
  return (
  <div className="App">
  </div>
  );
};
export default App;
```

4. Create our left-hand side of the app, our **`RecipeList`** component, in **`src/RecipeList.js`**:

```
import React from "react";
const RecipeList = () => {
  return (
  <div className="RecipeList">
     <h1>Recipes</h1>
    <ul>
        <li>Recipe 1</li>
```

```
          <li>Recipe 2</li>
          <li>Recipe 3</li>
      </ul>
    </div>
    );
};
export default RecipeList;
```

5. Create the right-hand side of our app, the **`Recipe`** component, in **`src/Recipe.js`**:

```
import React from "react";
const Recipe = () => {
  return (
  <div className="Recipe">
      <h1>Selected Recipe</h1>
      <p>A Recipe</p>
  </div>
  );
};
export default Recipe;
```

6. Return to **`src/App.js`** and include both new components:

```
import React from "react";
import "./App.css";
import RecipeList from "./RecipeList";
import Recipe from "./Recipe";
const App = () => {
  return (
  <div className="App">
      <RecipeList />
      <Recipe />
  </div>
  );
};
export default App;
```

7. Create the **`RecipeContext`** and **`RecipeService`** components, which we will create in a single file and export both. Create **`src/RecipeService.js`** and export both (neither as a default export):

```
import React from "react";
const RecipeContext = React.createContext();
const RecipeService = props => {
  return (
  <RecipeContext.provider value={}>{props.children}</RecipeContext.
provider>
  );
};
export { RecipeContext, RecipeService };
```

8. Return to **`src/App.js`** and import the new **`RecipeService`** component. Then, wrap your **`App`** component declaration in the **`RecipeService`** component:

```
import React from "react";
import "./App.css";
import RecipeList from "./RecipeList";
import Recipe from "./Recipe";
import { RecipeService } from "./RecipeService";
const App = () => {
  return (
  <RecipeService>
      <div className="App">
          <RecipeList />
          <Recipe />
      </div>
  </RecipeService>
  );
};
export default App;
```

9. Set up the list of recipes to load first, using the same **`useEffect(..., [])`** syntax we've used previously. Each recipe will have a title and the steps for the recipe, which we will display later in the **`Recipe`** and **`RecipeList`** components.

   We want the effect to execute after **2** seconds to simulate loading from an AJAX request, and we will use the setter function, **`setRecipes`**, to set the list of recipes. Feel free to use whatever recipes you would like. Don't forget to add the recipe's state to the value of your **`RecipeContext.Provider`** call or this won't work:

```
import React from "react";
const RecipeContext = React.createContext();
const RecipeService = props => {
  const [recipes, setRecipes] = React.useState(null);
  React.useEffect(() => {
  setTimeout(() => {
    setRecipes([
      { title: "Ice", steps: ["Take Water", "Freeze the
      Water"] },
      {
        title: "Sandwich",
        steps: [
          "Take two slices of bread",
          "Put stuff in the middle of that bread",
          "Eat your sandwich"
        ]
      }
    ]);
  }, 2000);
  }, []);
  return (
  <RecipeContext.Provider value={{ recipes }}>
    {props.children}
  </RecipeContext.Provider>
  );
};
export { RecipeContext, RecipeService };
```

10. Attach the **`RecipeList`** component to the **`RecipeContext`** component:

```
import React from "react";
import { RecipeContext } from "./RecipeService";
const RecipeList = () => {
  const { recipes } = React.useContext(RecipeContext);
  return (
  <div className="RecipeList">
    <h1>Recipes</h1>
    <ul>
      {recipes &&
        recipes.map((recipe, index) => (
          <li key={`r-${index}`}>{recipe.title}</li>
        ))}
    </ul>
  </div>
  );
};
export default RecipeList;
```

11. Return to **`src/RecipeService.js`**, where we will set up a selected recipe state and context to be used in the two **`Recipe`** components. The default value for this recipe will be a default recipe, which we will declare at the top:

```
import React from "react";
const RecipeContext = React.createContext();
const defaultRecipe = { title: "None selected", steps: [] };
const RecipeService = props => {
  const [recipes, setRecipes] = React.useState(null);
  const [selectedRecipe, setSelectedRecipe] =
   React.useState(defaultRecipe);
  React.useEffect(() => {
  setTimeout(() => {
    setRecipes([
      { title: "Ice", steps: ["Take Water", "Freeze the
      Water"] },
      {
        title: "Sandwich",
        steps: [
          "Take two slices of bread",
          "Put stuff in the middle of that bread",
          "Eat your sandwich"
```

```
          ]
        }
      ]);
  }, 2000);
  }, []);
  return (
  <RecipeContext.Provider
    value={{ recipes, selectedRecipe, setSelectedRecipe }}
  >
    {props.children}
  </RecipeContext.Provider>
  );
};
export { RecipeContext, RecipeService };
```

12. Hook up the new **`setSelectedRecipeIndex`** function to the **`src/RecipeList.js`** component, where we will set it so that when a user clicks on a recipe, the recipe display shows up in the **`Recipe`** component. Also, if **`recipes`** variable is **`null`**, we should display a **`Recipes Loading`** message, so we will do that in the JSX for our **`RecipeList`** component:

```
import React from "react";
import { RecipeContext } from "./RecipeService";
const RecipeList = () => {
  const { recipes, setSelectedRecipe } =
   React.useContext(RecipeContext);
  return (
  <div className="RecipeList">
    <h1>Recipes</h1>
    <ul>
      {recipes != null
        ? recipes.map((recipe, index) => (
            <li key={`r-${index}`} onClick={() =>
     setSelectedRecipe(recipe)}>
              {recipe.title}
            </li>
          ))
        : "Recipes loading..."}
    </ul>
  </div>
```

```
    );
};
export default RecipeList;
```

13. Attach **`RecipeContext`** to the **`Recipe`** component and make it so that we can display the selected recipe to the user. We will grab **`selectedRecipe`** out of the **`RecipeContext`** component and use that to display the current selected recipe, which is easy since we are setting a default value and can guarantee that the **`selectedRecipe`** component will always be set to something:

```
import React from "react";
import { RecipeContext } from "./RecipeService";
const Recipe = () => {
  const { selectedRecipe } = React.useContext(RecipeContext);
  return (
  <div className="Recipe">
    <h1>Selected Recipe</h1>
    <strong>{selectedRecipe.title}</strong>
    <ul>
      {selectedRecipe.steps.map((step, index) => (
        <li key={`s-${index}`}>{step}</li>
      ))}
    </ul>
  </div>
  );
};
export default Recipe;
```

When our component is first loading, we will see the **`Recipes loading`**... message and the blank selected recipe:

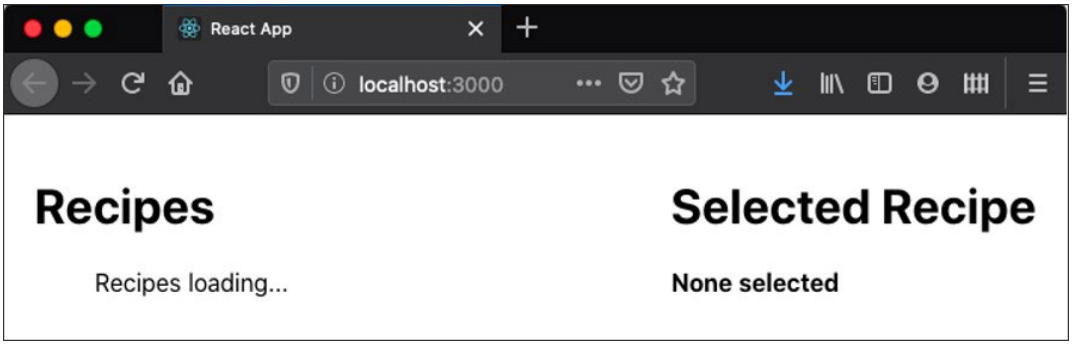

Figure 13.8: Component displaying the Recipes loading... message

Then, we should see the list, but still the blank recipe, until we click on something:

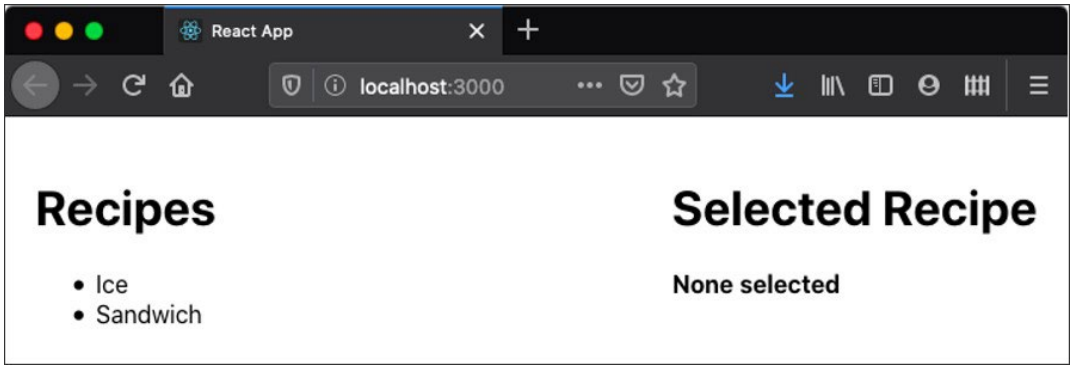

Figure 13.9: Component displaying a blank recipe

Finally, when we click on a recipe, we will see the recipe load on the right-hand side:

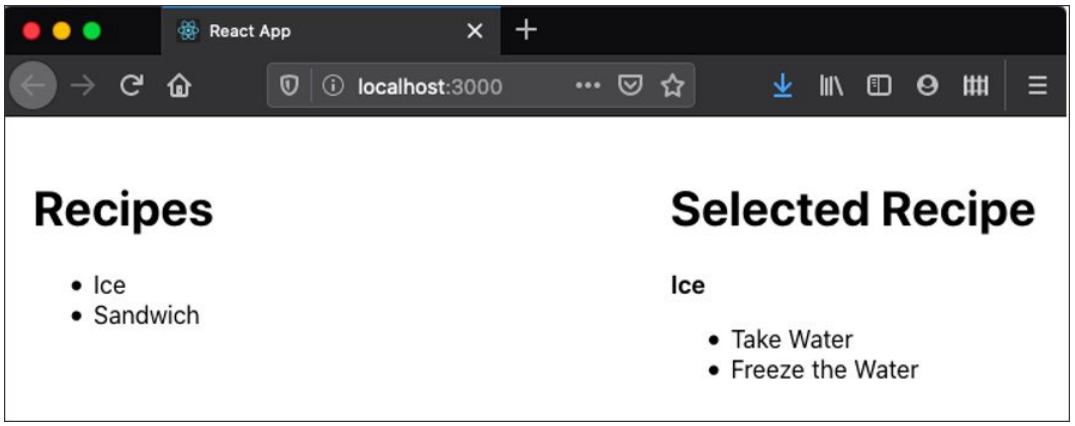

Figure 13.10: Final component output

CHAPTER 14: FETCHING DATA BY MAKING API REQUESTS

ACTIVITY 14.01: BUILDING AN APP TO REQUEST DATA FROM UNSPLASH

Solution

1. Install **`create-react-app`** for the project named **`search-unsplash-images`**.
2. In the **`src`** folder, create a file called **`index.js`** and add the following code. It will import the **`<App>`** component and render it in HTML:

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './components/App';

ReactDOM.render(<App />, document.querySelector('#root'));
```

3. Still in the **`src`** folder, create another file called **`App.js`** and add some boilerplate for the **`<App>`** component:

```
import React, { Component } from 'react';

class App extends Component {
  render() {
    return (
      <div className="page">
        <div className="box">
          <h1>Get a Random Photo</h1>
        </div>
      </div>
    );
  }
}

export default App;
```

4. Let's add a button. Add a button right below the <h1> heading. Let's reference to a function called **`onGetPhoto()`** in the **`onClick`** event. Don't forget to bind the **`this`** keyword.

5. Now let's create the **`onGetPhoto()`** function, which will make requests to the Unsplash API with Axios:

```
onGetPhoto() {
}
```

6. Before adding any Axios code, we need to install Axios first. Go to your Terminal and run the **`yarn add axios`** command. Once installed, import Axios right below where you imported React:

```
import React, { Component } from 'react';
import axios from 'axios';
```

7. To make requests to Unsplash API, we first need to create a developer account. Go to https://unsplash.com/join and sign up. Once signed up, click on the **`Your apps`** link at the top of the menu and click the **`New Application`** box. Read the API Use and Guidelines and click the **`Accept terms`** button. When you click the Accept terms button, a pop-up window will appear. Add your **`Application name`** and **`Description`** and click the **`Create application`** button. In a few seconds, your app will be created. Scroll down and find the **`Access Key`**, which we will be using soon.

8. Now let's get the random photo by clicking on the button in our React app. In the **`onGetPhoto()`** function, add **`axios`** with the **`get()`** method. For the first argument in the get() method, we are going to add the endpoint. We can find the endpoint to get the random photo on the *Get a random photo* https://unsplash.com/documentation#get-a-random-photo documentation page. The HTTP method will be **`GET`** and the URL will be **`/photos/random`**. To get the base URL, head over to the *Location* page and we can get the base URL as https://api.unsplash.com/. So, to get a random photo, we are going to make a request to https://api.unsplash.com/photos/random.

9. If you head over to the *Public actions* https://unsplash.com/documentation#public-actions section, we need to authenticate requests, and to do that, we need to pass the access key via the **`HTTP`** Authorization header as specified on the documentation page. With all of this information, we can add the Axios code as shown in the following snippet. Let's also print the response in the console:

```
onGetPhoto() {
  axios.get('https://api.unsplash.com/photos/random', {
  headers: {
      Authorization: 'Client-ID 123456789'
  }
  }).then(res => {
  console.log(res);
  }).catch(err => {
  console.log(err);
  });
}
```

> **NOTE**
>
> You must replace **`123456789`** with your access key.

10. When you click on the button, you should receive data like this:

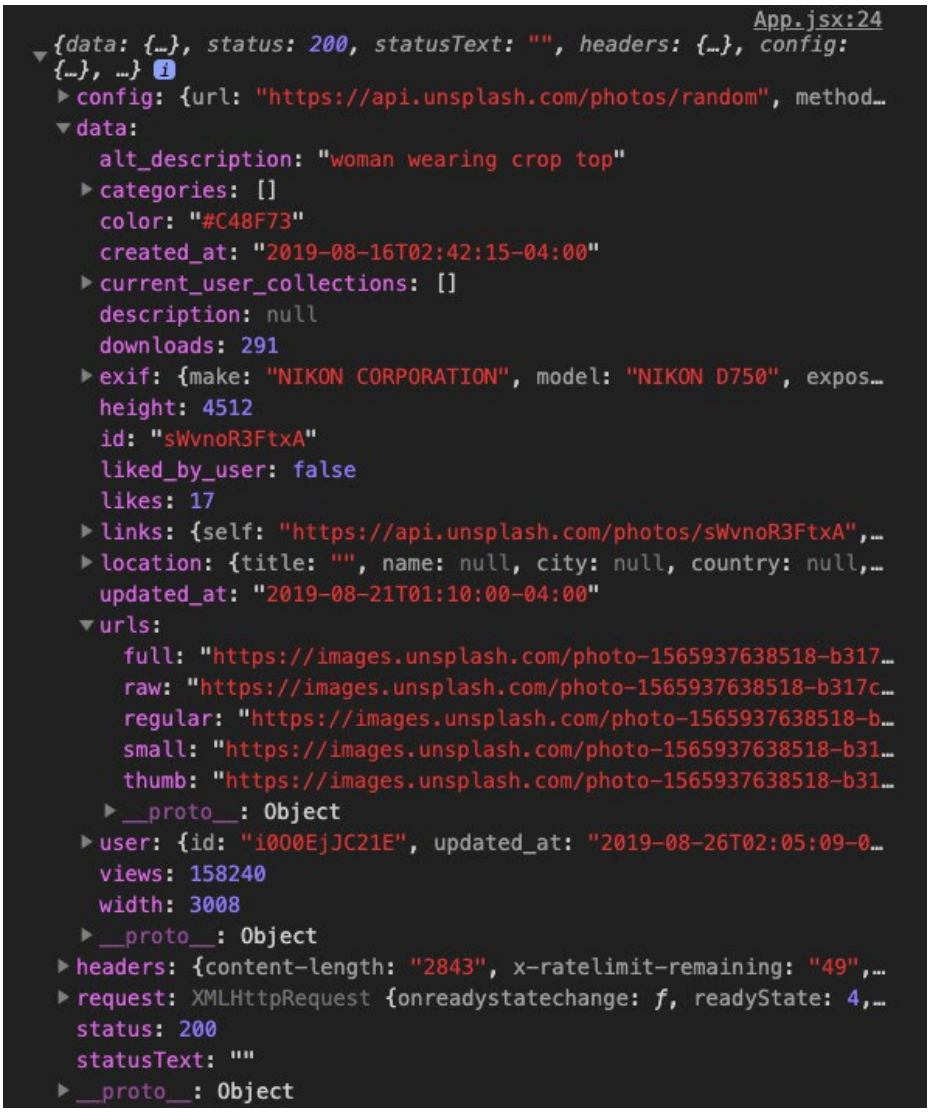

Figure 14.32: Receiving data from Unsplash

11. Now let's get the statistics with the data we just received. Go to the documentation page, https://unsplash.com/documentation#get-a-photos-statistics, and you will see that we need to provide the public ID of the photo. For the data we received in the previous step, the ID can be retrieved from **`data.id`**. For our data, the ID is **`sWvnoR3FtxA`**.

12. To make another API request, we are going to add the Axios code inside the **`then()`** method. First, we are going to define the **`photoId`**, and we need to include the ID in the endpoint. From the documentation page, the ID should go in between the photos and statistics in the endpoint. We also need to send the Authorization header to authenticate. Altogether, the Axios code should look like this inside the **`then()`** method:

```
onGetPhoto() {
  axios.get('https://api.unsplash.com/photos/random', {
  headers: {
    Authorization: 'Client-ID 12345'
  }
  }).then(res => {
  const photoId = res.data.id;
  axios.get(`https://api.unsplash.com/photos/${photoId}/statis
   tics`, {
      headers: {
          Authorization: 'Client-ID
   6463359ac22d145576915c2fd1d28838f53e80174b2e95fc0b86026b6c7d69
  55'
      }
  }).then(res => {
      console.log('stat', res);
  }).catch(err => {
      console.log(err);
  });
  }).catch(err => {
    console.log(err);
  });
}
```

13. Click the **`Random Photo`** button, and you should receive the data from the server as follows:

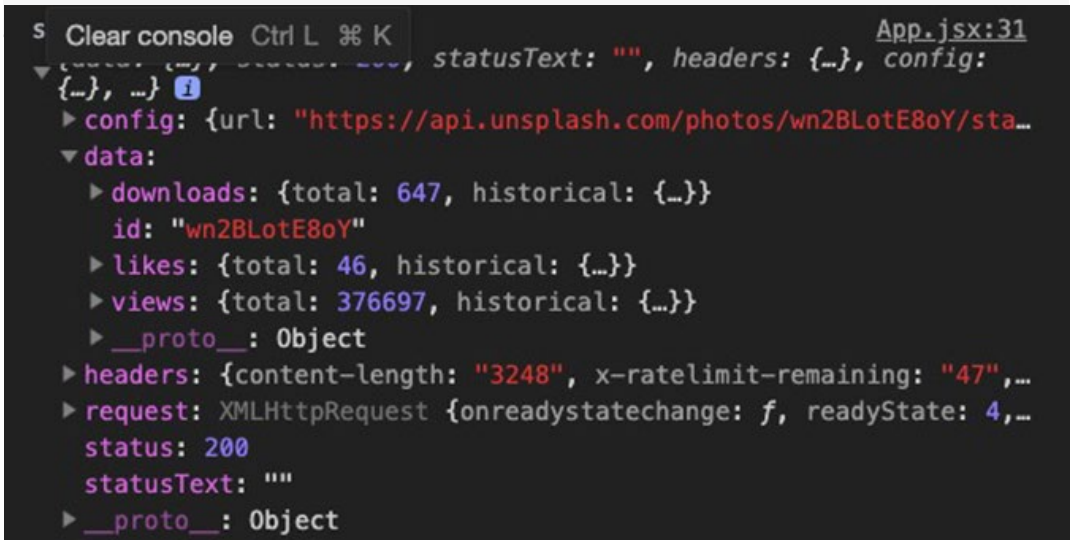

Figure 14.33: Receiving statistics of the random photo

(**Optional**) To display the photo covering the entire page and display the statistics, first let's create a CSS file. In the **`src`** folder, create a file called **`App.css`** and add the following CSS code:

`App.css`

```
1 body {
2  margin: 0;
3 }
4
5 h2 {
6  margin-top: 50px;
7 }
```

`The complete code can be found here` https://packt.live/3dJVujl

14. Import the CSS file:

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

15. At the top of the **<App>** component, add a constructor method and initialize the state with the **photoUrl**, **downloads**, views, and likes with the default values:

```
constructor(props) {
  super(props);

  this.state = {
    photoUrl: '',
    downloads: 0,
    views: 0,
    downloads: 0,
    likes: 0,
  };
}
```

16. Let's add markup to display the button and the statistics right underneath the **<button>**. For the statistics, we are going to get the value from the state with the **toLocalString()** method to add commas as thousand separators:

```
<h2>Statistics</h2>
<ul className="stat-list">
  <li className="stat-item">
  <h3>Downloads</h3>
  <p>{this.state.downloads.toLocaleString()}</p>
  </li>
  <li className="stat-item">
  <h3>Views</h3>
  <p>{this.state.views.toLocaleString()}</p>
  </li>
  <li className="stat-item">
  <h3>Likes</h3>
  <p>{this.state.likes.toLocaleString()}</p>
  </li>
</ul>
```

17. As we want to display a random photo covering the entire page, we are going to add the photo using *inline styles* as a **`background-image`** in the **`page`** element. To add an inline style, we are going to define a variable called style right above **`return()`** in the render() method and assign **`backgroundImage`** in an object. Once the variable has been defined, add the style to the **`style`** attribute in the **`page`** element:

```
const style = {
  backgroundImage: `url(${this.state.photoUrl})`
};

return(
  <div className="page" style={style}>
  ...
```

18. We need to update the state in two places, one for updating the photo's URL and the other for the statistics. Let's update the photo URL in the state first. Add the following code at the top of the **`then()`** method for requesting a random photo:

```
this.setState({
  photoUrl: res.data.urls.full
});
```

19. To update the state for the statistics, remove **`console.log('stat', res);`** and add the following code inside the then() method after requesting the statistics:

```
this.setState({
  downloads: res.data.downloads.total,
  views: res.data.views.total,
  likes: res.data.likes.total
});
```

20. Click the **`Random Photo`** button and check whether the photo appears, covering the entire page, and also that the statistics are updated.

21. The output should look like this:

Figure 14.34: Final app

> **NOTE**
>
> The image above, *Figure 14.30*, is susceptible to change since we are accessing a random photo.

CHAPTER 15: PROMISE API AND ASYNC/AWAIT

ACTIVITY 15.01: CREATING A MOVIE APP

Solution

1. Install **`create-react-app`** for the project named **`search-movies`**.
2. In the **`src`** folder, remove all the existing files. Once they are removed, create a new file called index.js and add the following code. It will import the **`<App>`** component and render it in HTML:

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './components/App';
ReactDOM.render(<App />, document.querySelector('#root'));
```

3. Still in the **`src`** folder, create a folder called components and create a file called **`App.js`** inside the component folder. Once this is done, add some boilerplate for the **`<App>`** component:

```
import React from 'react';
const App = () => {
  return(
    <div className="page">
   <h1>5 Most Popular Movies Now</h1>
    </div>
  );
};
export default App;
```

4. In the component folder, create one more file called **`App.css`** and add the style code provided earlier. Also import the CSS file at the top of **`App.js`**:

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

5. Let's add a form and a button under <h1>. Add an **`onSubmit`** event and a reference to a function called **`getMovies`**:

```
<form onSubmit={getMovies}>
  <button className="button">Get Now</button>
</form>
```

6. Create the **getMovies** function above **return()**. For **getMovies**, receive an event and add **event.preventDefatut()** to prevent page reload when we submit the form by clicking the button:

```
const getMovies = e => {
  e.preventDefault();
}
```

7. We are going to fetch the popular movies. To do that, declare a variable called **API_KEY** and assign your **API** key to it. Add it outside the **App** component:

```
import React from 'react';
import './App.css';
const API_KEY = '12345';
const App = () => {
…
```

8. To fetch the popular movies, we are going to use the endpoint specified in the Discover Movie documentation page, https://developers.themoviedb.org/3/discover/movie-discover.

   To test it, go to the Try it out tab, add your **api** key, select **popularity.desc** for **sort_by**, and then click the **Send request** button. In a few seconds, you will receive a list of currently popular movies. Let's use the endpoint:

```
https://api.themoviedb.org/3/discover/movie?api_
key=12345&language=en-US&sort_by=popularity.desc&include_
adult=false&include_video=false&page=1
```

9. Inside the **getMovies** function, first construct the API endpoint:

```
const getMovies = e => {
  e.preventDefault();


  const moviesUrl =
  `https://api.themoviedb.org/3/discover/movie?api_key=${API_
KEY}&language=en-
  US&sort_by=popularity.desc&include_adult=false&include_
video=false&page=1`;
}
```

10. Next, we are going to use Axios to fetch the movie data, so let's install Axios. Once it's installed, import Axios at the top of **`App.js`**:

```
yarn add axios
import React from 'react';
import axios from 'axios';
import './App.css';


…
```

11. As we are going to use **`async`**/**`await`**, add an **`async`** expression in the function name and add await in front of Axios:

```
const getMovies = async (e) => {
…
  const res = await axios.get(moviesUrl);
}
```

12. Let's log the responses and see if you are receiving popular movie data when you click on the button:

```
const getMovies = async (e) => {
…


  const res = await axios.get(moviesUrl);
  console.log(res.data);
}
```

The output is as follows:

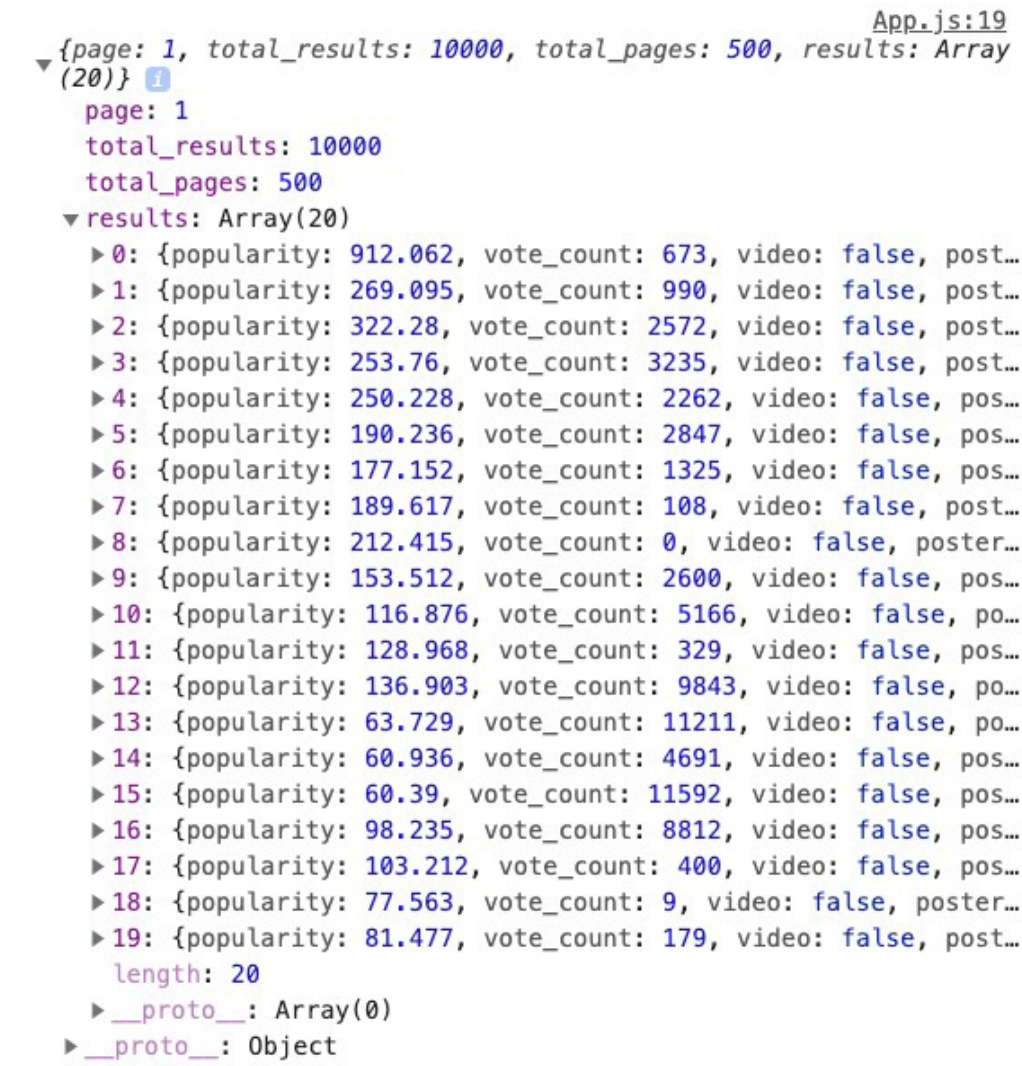

Figure 15.15: Receiving movie data

13. As you only want to display five popular movies, let's use the **`slice()`** method to get the first five movies from the response we received from the server:

```
const getMovies = async (e) => {
…

  const res = await axios.get(moviesUrl);
  const displayMovies = res.data.results.slice(0, 5);
}
```

14. To display the movies, we are going to store the popular movie data in a state by using **`useState`**. Let's import **`useState`**:

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

15. Once you have imported **`useState`**, let's declare movies at the top of the App component:

```
const App = () => {
  const [movies, setMovies] = useState([]);

  const getMovies = async (e) => {
…
```

16. Let's store the five popular movies in the state:

```
const getMovies = async (e) => {
…

  const res = await axios.get(moviesUrl);
  const displayMovies = res.data.results.slice(0, 5);
  setMovies(displayMovies);
}
```

17. Now let's display the movies. To display the movies, we are going to create another file called **`Movie.js`** and import the **`<Movie>`** component:

```
import React, { useState } from 'react';
import axios from 'axios';
import Movie from './Movie';
import './App.css';
```

18. Under the **`<form>`**, add **`<ul>`** and then loop the movie data using the **`map()`** method to display the **`Movie`** component wrapped by **`<li>`**. Additionally, we also need to add the unique key to <li>.

    To display the movie details, let's send the movie value through a prop called **`movie`**:

```
return (
  <div className="page">
    <h1>5 Most Popular Movies Now</h1>

    <form onSubmit={getMovies}>
```

```
    <button className="button">Get Now</button>
      </form>

       <ul>
    {movies.map((movie, index) => (
      <li key={index}>
        <Movie movie={movie} />
      </li>
    ))}
    </ul>
    </div>
  );
```

19. In the **`Movie`** component, let's receive the movie prop value:

```
import React from 'react';

const Movie = ({movie}) => {
  return <div>Movies</div>;
};
export default Movie;
```

20. In **`return()`**, inside the top-level **`<div>`**, add the poster image in the **`<img>`** tag that is wrapped with **`<div>`**. For the URL of the image, please refer to the Images page, https://developers.themoviedb.org/3/getting-started/images. We will get the image filename from **`movie.poster_path`** in the movie prop, and for the **`alt`** tag, we will use **`movie.title`**.

    Let's add a title too. For the title, we are going to use **`movie.title`** inside **`<h2>`**:

```
import React from 'react';

const Movie = ({movie}) => {
  return(
    <div>
    <div><img
   src={`https://image.tmdb.org/t/p/w200${movie.poster_path}`}
alt={movie.title} /></div>
      <h2>{movie.title}</h2>
    </div>
```

```
    );
  };

  export default Movie;
```

21. Now let's save and test whether we are getting the five poster images and titles of the popular movies:

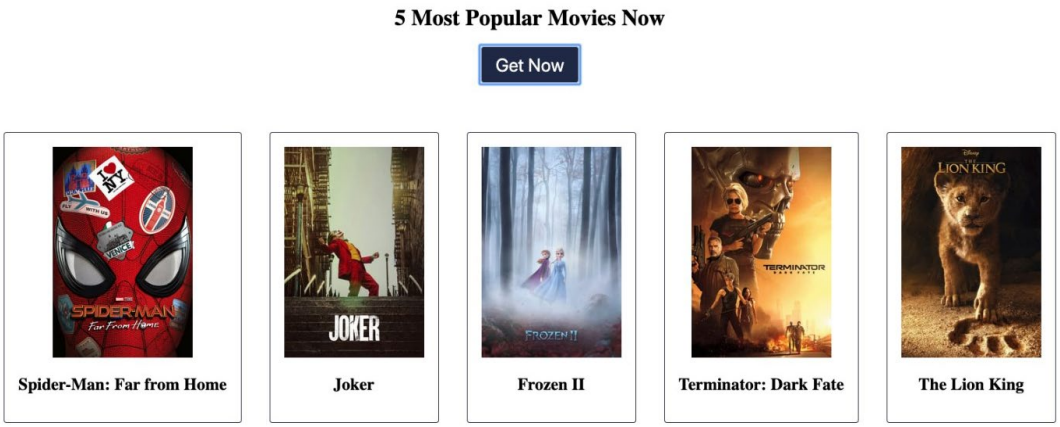

Figure 15.16: Receiving five popular movies with poster images and titles

22. It's time to get the first three cast members. But before we do it, let's go back to App.js and add the try/catch pattern to catch any possible future errors:

```
try {
  const res = await axios.get(moviesUrl);
  const displayMovies = res.data.results.slice(0, 5);
  setMovies(displayMovies);
} catch (error) {
  console.log(error);
}
```

23. To fetch the cast data, let's take a look at what API endpoint we need. Go to the Get Credits page, https://developers.themoviedb.org/3/movies/get-movie-credits, in the Movies section, and go to the Try it out tab. To get the credits, you need to add the movie ID. Let's get one from our five popular movies data, **`429617`**, as an example. When you successfully fetch the data, you will see that the credits data contains two lists, cast and crew. What we need is cast.

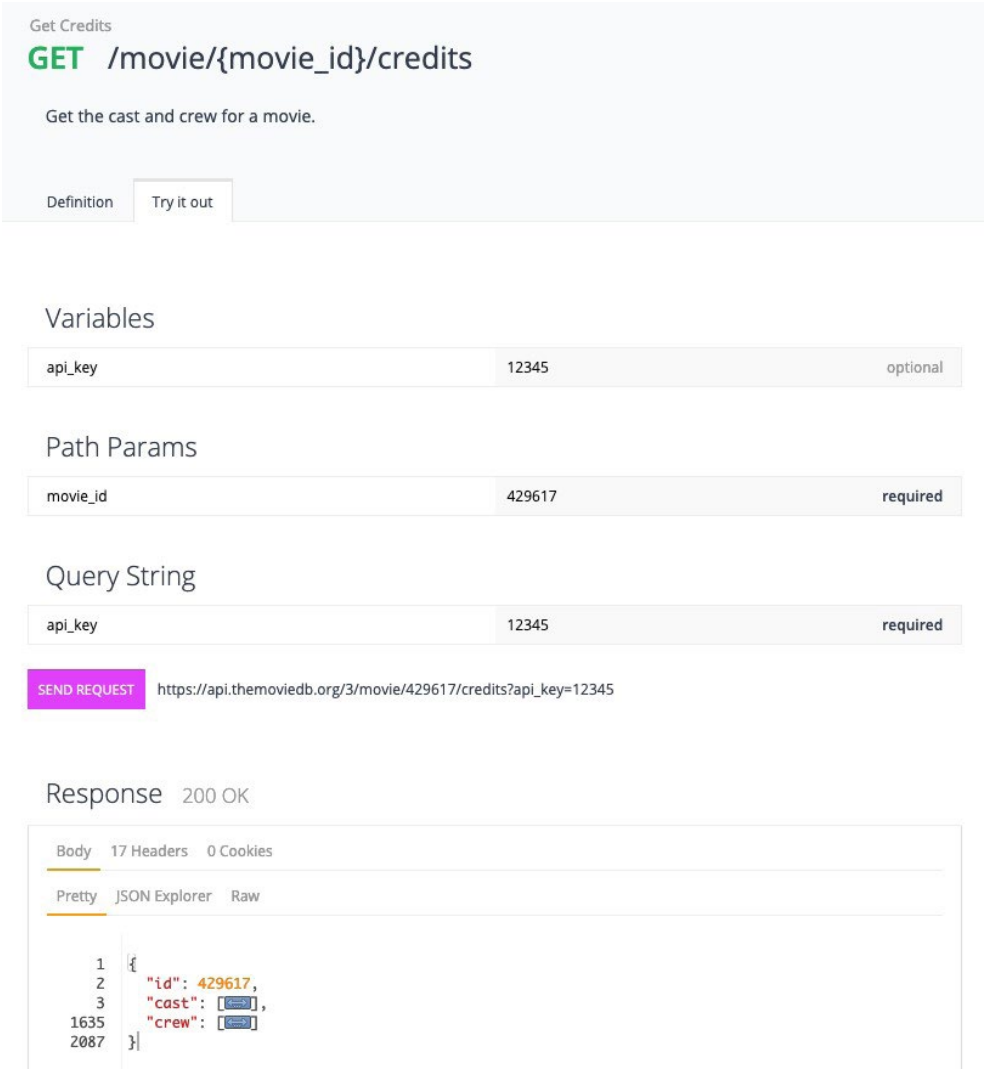

Figure 15.17: Receiving cast data

We are going to compare the movie id stored in the movie state with the credits state, which we will create, so we need to add both **`movieId`** and cast in the credits state.

24. To get the cast members, we need to provide each movie ID in the **`credits`** API endpoint. As we want to get the data in parallel so that we can display all the cast members for the five popular movies at once, let's use the **`map()`** method:

```
try {
  const res = await axios.get(moviesUrl);
  const displayMovies = res.data.results.slice(0, 5);
  setMovies(displayMovies);


  const creditPromise = await displayMovies.map(async movie =>
  {
  const creditUrl =
   `https://api.themoviedb.org/3/movie/${movie.id}/credits?api_ke
y=${API_KEY}`;
 const creditRes = await axios.get(creditUrl);


 return {movieId: movie.id, credits: creditRes.data.cast.slice(0,
3)};
  });
} catch (error) {
  console.log(error);
}
```

25. Once we have created a new array in the **`creditPromise`**, let's return a single promise using **`Promise.all()`**:

```
try {
…


  const creditPromise = await displayMovies.map(async movie => {
  const creditUrl =
`https://api.themoviedb.org/3/movie/${movie.id}/credits?api_ke
  y=${API_KEY}`;
  const creditRes = await axios.get(creditUrl);


  return {movieId: movie.id, credits:
        creditRes.data.cast.slice(0, 3)};
  });
  const creditArray = await Promise.all(creditPromise);
} catch (error) {
  console.log(error);
}
```

Let's log what we are getting in **`creditArray`**:

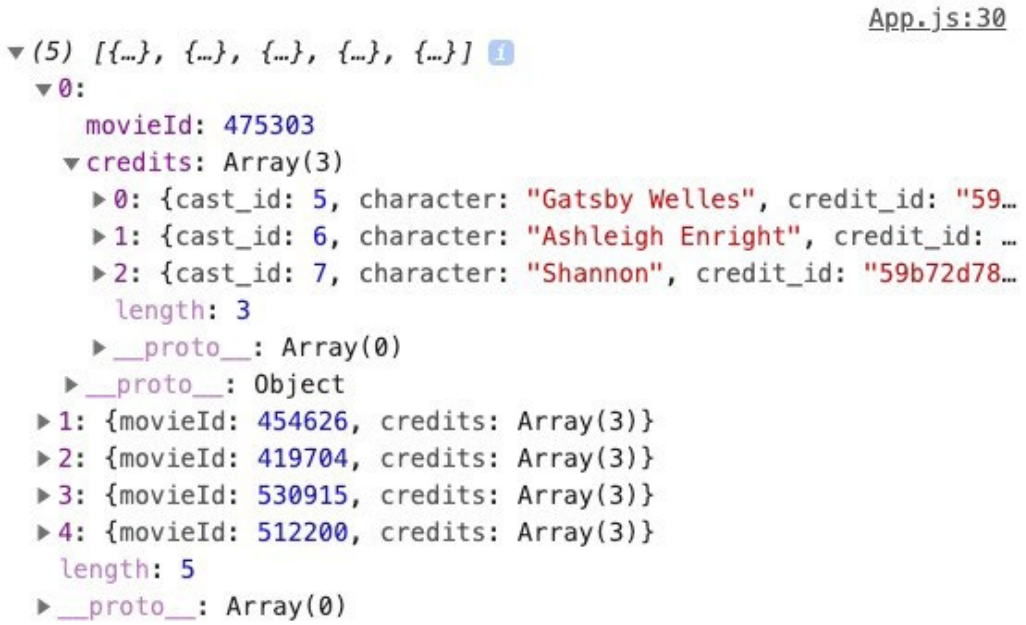

Figure 15.18: Receiving three casts for each movie along with movieId

26. Let's save the **`creditArray`** value to the credits state:

```
const App = () => {
  const [movies, setMovies] = useState([]);
  const [credits, setCredits] = useState([]);

  const getMovies = async (e) => {
…

 try {
…

   const creditPromise = await displayMovies.map(async movie
   => {
    const creditUrl =
     `https://api.themoviedb.org/3/movie/${movie.id}/credits?api_ke
y=${API_KEY}`;
    const creditRes = await axios.get(creditUrl);

    return {movieId: movie.id, credits:
    creditRes.data.cast.slice(0, 3)};
```

```
    });

    const creditArray = await Promise.all(creditPromise);
    setCredits(creditArray);
  } catch (error) {
    console.log(error);
  }
    }
```

27. Let's send the **`credits`** value to the **`Movie`** component:

```
  return(
    <div className="page">
  …

    <ul>
    {movies.map((movie, index) => (
      <li key={index}>
        <Movie movie={movie} credits={credits} />
      </li>
    ))}
    </ul>
    </div>
  );
```

28. In the **`Movie`** component, we are going to get first three cast members for each movie and send it to the new component called **`Credit`**. Let's create **`Credit.js`** in the **`component`** folder and import it from **`Movie.js`**.

    Add the **`Credit`** component under **`<h2>`** wrapped with **`<ul>`** and add a prop called casts and a reference to a function called **`getCredits`** with the movie ID sent to the function:

```
  import React from 'react';
  import Credit from './Credit';

  const Movie = ({movie, credits}) => {
    return(
    <div>
    <div><img src={`https://image.tmdb.org/t/p/w200${movie.poster_
  path}`}
   alt={movie.title} /></div>
```

```
      <h2>{movie.title}</h2>
      <ul><Credit casts={getCredits(movie.id)} /></ul>
    </div>
    );
};

export default Movie;
```

29. Create the **`getCredits`** function at the top of the **`Movie`** component. The **`getCredits`** function will get the array matching the current movie id and return the cast value. In that way, we can send the first three cast members to the **`Credit`** component.

    Also make sure to return early if there are no credits available when fetching the movie data:

```
const getCredits = (movieId) => {
  const value = credits.find(o => o.movieId === movieId);

  if (value === undefined) return;

  return value.credits;
};
```

30. Let's add a **`console.log`** before return **`value.credits`** to see what we are getting:

```
const getCredits = (movieId) => {
  const value = credits.find(o => o.movieId === movieId);

  if (value === undefined) return;
  console.log(value);
  return value.credits;
};
```

The console output is as follows:

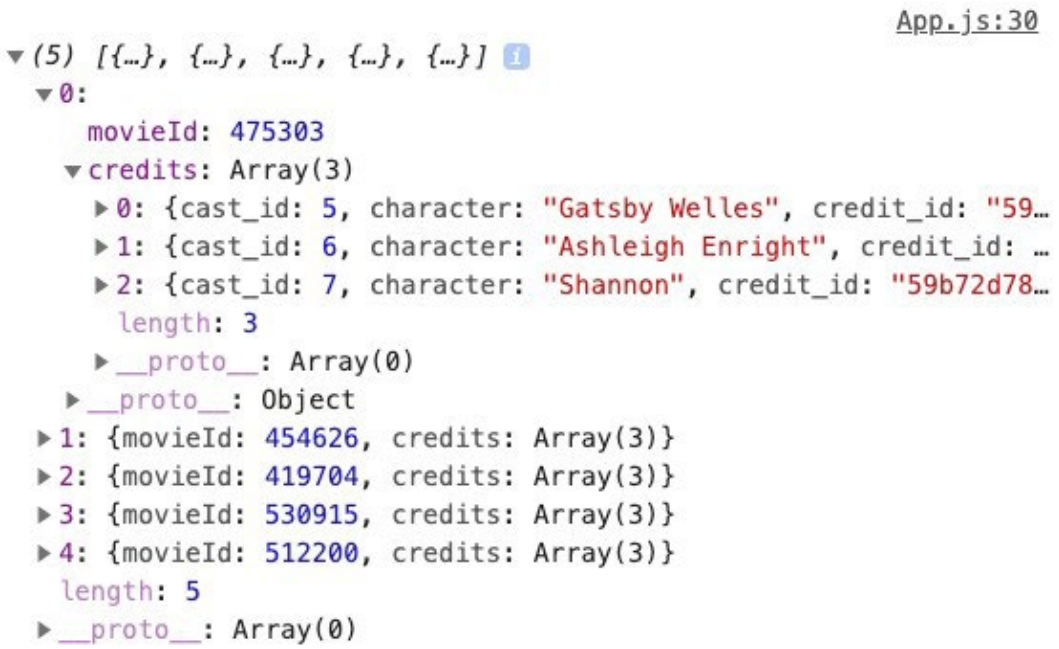

Figure 15.19: Receiving three cast members separately for each movie

31. In the **`Credit`** component, let's prepare for the case when no cast members are returned, which will throw an error on the initial load because no data has been fetched:

```
import React, { Fragment } from 'react';
const Credit = ({casts}) => {
  if (casts == null) return <Fragment/>;
  return(
  <Fragment>Casts</Fragment>
  );
};
export default Credit;
```

32. In **`return()`**, loop the **`cast`** array and display the name and the image:

```
import React, { Fragment } from 'react';

const Credit = ({casts}) => {
  if (casts == null) return <Fragment/>;
```

```
    return(
<Fragment>
  {casts.map((cast, index) => (
    <li key={index}>
      <div>{cast.name}</div>
      <div><img width="70"
…
src={`https://image.tmdb.org/t/p/original${cast.profile_path}`
     } alt={cast.name} /></div>
    </li>
  ))}
</Fragment>
  );
};

export default Credit;
```

33. Let's test whether the app is working correctly. Clicking on the button should display the results as shown in the screenshot.

    The output is as follows:

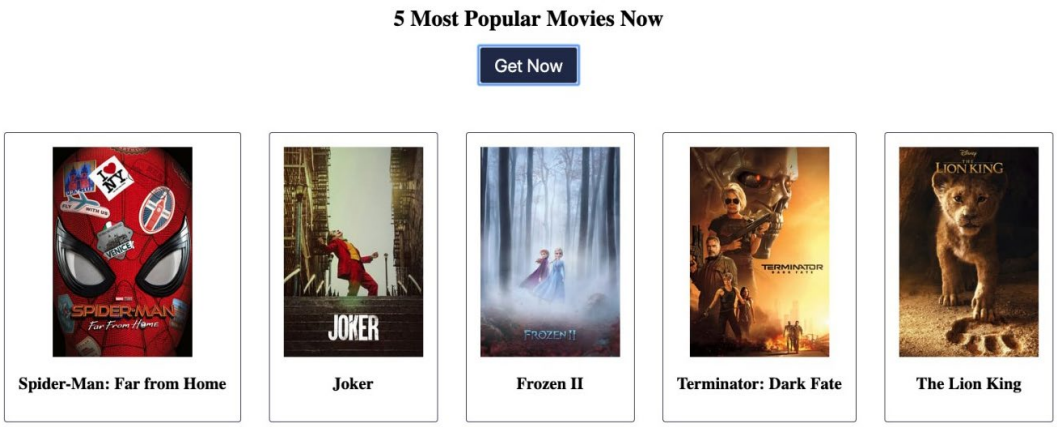

Figure 15.20: Final output

CHAPTER 16: FETCHING DATA ON INITIAL RENDER AND REFACTORING WITH HOOKS

ACTIVITY 16.01: CREATING AN APP USING POTTER API

Solution:

1. Start a new React application for the project named **`harry-potter-api`**:

```
npx create-react-app harry-potter-api
```

2. In the **`src`** folder, remove all the existing files. Once they are removed, create a new file called **`index.js`** and add the following code. This will import the **`<App>`** component and render it in HTML:

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './components/App';

ReactDOM.render(<App />, document.querySelector('#root'));
```

3. Still in the **`src`** folder, create a folder called **`components`** and then create a file called **`App.js`** inside the **`component`** folder. Once that's done, add some boilerplate for the **`<App>`** component:

```
import React from 'react';
const App = () => {
  return(
    <div className="page">
    <h1>Characters</h1>
    </div>
  );
};

export default App;
```

4. In the **`component`** folder, create one more file called **`App.css`** and add the style code provided previously. Import the CSS file at the top of **`App.js`**:

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

We are going to add four buttons, one labeled for each house (**`Gryffindor`**, **`Slytherin`**, **`Hufflepuff`**, and **`Ravenclaw`**).

To get the house details, fetch the data using the **`/houses`** endpoint by means of the **`GET`** method:

```
const res = await axios.get(
  `https://www.potterapi.com/v1/houses?key=${API_KEY}`
);
```

5. We need to include the API key with the endpoint. Include our API key at the very top of the page, right after importing the components:

```
import React, { useEffect } from 'react';
import axios from 'axios';
import './App.css';

const API_KEY = '12345';
```

6. Add the **`axios`** code to the **`useEffect`** hook. For the second argument of the **`useEffect`** hook, as we only want to fetch the houses' data upon mounting the component, add an empty array as the second argument:

```
const App = () => {
  useEffect(() => {
    const res = await axios.get(
      `https://www.potterapi.com/v1/houses?key=${API_KEY}`
      );
    },[]};
}
```

7. Inside **`useEffect()`**, create a new function called **`getData`** and fetch data with **`async`**/**`await`** using **`axios`**. Then, use **`console.log`** to see what data we are getting with **`res.data`**:

```
import React, { useEffect } from 'react';
import axios from 'axios';

import './App.css';

const API_KEY = '12345';

const App = () => {
 useEffect(() => {
   const getData = async () => {
     const res = await axios.get(
```

```
          `https://www.potterapi.com/v1/houses?key=${API_KEY}`
        );

        console.log(res);
      };

      getData();
    }, []);

    return (
      <div className="container">
        <h1>Characters</h1>
      </div>
    );
  };
  export default App;
```

We should receive four results in an array with details of each house in the console:

```
▼data: Array(4)
  ▶0: {_id: "5a05e2b252f721a3cf2ea33f", name: "Gryffindor", mascot: "lion", h…
  ▶1: {_id: "5a05da69d45bd0a11bd5e06f", name: "Ravenclaw", mascot: "eagle", h…
  ▶2: {_id: "5a05dc8cd45bd0a11bd5e071", name: "Slytherin", mascot: "serpent",…
  ▶3: {_id: "5a05dc58d45bd0a11bd5e070", name: "Hufflepuff", mascot: "badger",…
   length: 4
```

Figure 16.13: List of houses

8. Now, let's save the data in a state called **houses**. As we are going to use the **useState** hook, let's import it first. At the top of the **App** component, define the house state, and then add **setHouses** to update the **houses** value inside the **useEffect**:

```
const [houses, setHouses] = useState([]);
…
useEffect(() => {
  const getData = async () => {
    const res = await axios.get(
      `https://www.potterapi.com/v1/houses?key=${API_KEY}`
    );
```

```
      console.log(res);
      setHouses(res.data);
    };

    getData();
  }, []);
```

9. Display four buttons, one with each house name. In **`return()`**, remove **`<h1>Characters</h1>`** and add a new **`div`** element with a class name, **`house`**. Inside the **`div`** element, loop the **`house`** array we fetched using the **`map()`** method. For each house, add a **`<button>`** element with the class name **`house__button`**. Then, add the house name to the button. Don't forget to add the key. We can use **`_id`** from the data fetch:

```
return (
  <div className="container">
    <div className="house">
      {houses.map(house => (
        <button key={house._id} className="house__button">
          {house.name}
        </button>
      ))}
    </div>
  </div>
);
```

Now, we should see **`4`** buttons, one with each house name, as per the following screenshot:

Figure 16.14: Four buttons displayed

10. Update the background and text color for each house. Check **`console.log`** again, and you will see the **`colors`** values from the data we received. Use the first value as the background and the second value for the text color:

```
▼data: Array(4)
  ▼0:
    ▼colors: Array(2)
       0: "scarlet"
       1: "gold"
       length: 2
      ▶__proto__: Array(0)
     founder: "Goderic Gryffindor"
     headOfHouse: "Minerva McGonagall"
     houseGhost: "Nearly Headless Nick"
     mascot: "lion"
    ▶members: (40) ["5a0fa648ae5bc100213c2332", "5a0fa67dae5bc100213c2333", "…
     name: "Gryffindor"
     school: "Hogwarts School of Witchcraft and Wizardry"
    ▶values: (4) ["courage", "bravery", "nerve", "chivalry"]
     __v: 0
     _id: "5a05e2b252f721a3cf2ea33f"
    ▶__proto__: Object
  ▶1: {_id: "5a05da69d45bd0a11bd5e06f", name: "Ravenclaw", mascot: "eagle", h…
  ▶2: {_id: "5a05dc8cd45bd0a11bd5e071", name: "Slytherin", mascot: "serpent",…
  ▶3: {_id: "5a05dc58d45bd0a11bd5e070", name: "Hufflepuff", mascot: "badger",…
   length: 4
```

Figure 16.15: Getting the colors for each house

11. Apply the **`color`** values directly to the inline style:

```
return (
  <div className="container">
    <div className="house">
      {houses.map(house => (
        <button
          key={house._id}
          className="house__button"
          style={{
            backgroundColor: house.colors[0],
            color: house.colors[1]
          }}
        >
          {house.name}
```

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

Now, you should see that the button colors are updated. However, there are two colors missing, **`scarlet`** (the background color of **`Gryffindor`**) and **`bronze`** (the text color of **`Ravenclaw`**). This is because those two colors are not recognized by the CSS, so we need to convert it to hex values. Also, when you look at the **`bronze`** value, there is an extra space in front of the word. We need to remove the space to compare the value.

12. Now, let's create a new function called **`colorConverter`** and check whether the color value is either **`scarlet`** or **`bronze`** (without the space) and, if so, return it with **`#ff2400`** for scarlet or **`#cd7f32`** for **`bronze`**, otherwise use the color value received from the server. As this is a utility function, create a new file called **`utils.js`** and add the **`colorConverter`** function. Also, export the **`colorConverter`** in **`App.js`** as a named export:

```
export const colorConverter = color => {
  const colorUpdated = color.replace(' ', '');

  switch (colorUpdated) {
    case 'scarlet':
      return '#ff2400';
    case 'bronze':
      return '#cd7f32';
    default:
      return colorUpdated;
  }
};
```

In **`App.js`**, add the following code:

```
import { colorConverter } from './utils';

...

  return (
    <div className="container">
      <div className="house">
        {houses.map(house => (
```

```
            <button
              key={house._id}
              className="house__button"
              style={{
                backgroundColor:
                colorConverter(house.colors[0]),
                color: colorConverter(house.colors[1])
              }}
            >
              {house.name}
            </button>
          ))}
        </div>
      </div>
    );
```

You should see that each button gets the background and text color received from the data:

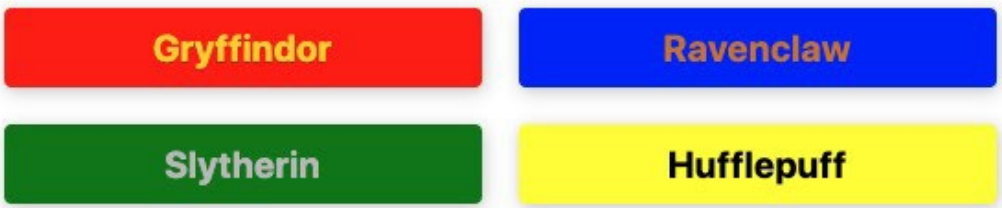

Figure 16.16: Four buttons with different colors

Now, let's display all characters upon initial rendering. As we are going to fetch data for all characters, we can do something similar to what we did to get the houses. However, as fetching those two different sets of data is quite similar, we are going to create a custom hook called **`useFetch`** and share it in order to fetch data for houses and characters.

13. First, create a new file called **`useFetch.js`** in the components folder and move **`API_KEY`**, **`useState`** for **`houses`**, and **`useEffect`** from **`App.js`** to **`useFetch.js`**. So, in the **`App`** component, we will only have **`return()`**.

In **useFetch.js**, create a new function called **useFetch** and receive a request as an argument. Paste the code we copied from **App.js** inside the **useFetch** function, but move **API_KEY** above the **useFetch** function:

```
import React, { useState, useEffect } from 'react';
import axios from 'axios';
import { colorConverter } from './utils';
import './App.css';

const App = () => {
  return (
    <div className="container">
      <div className="house">
        {houses.map(house => (
          <button
            key={house._id}
            className="house__button"
            style={{
              backgroundColor:
            colorConverter(house.colors[0]),
              color: colorConverter(house.colors[1])
            }}
          >
            {house.name}
          </button>
        ))}
      </div>
    </div>
  );
};

export default App;
```

14. In **useFetch.js**, add the following code:

```
const API_KEY = '12345';

const useFetch = request => {
  const [houses, setHouses] = useState([]);

  useEffect(() => {
    const getData = async () => {
```

```
        const res = await axios.get(
          `https://www.potterapi.com/v1/houses?key=${API_KEY}`
        );

        console.log(res.data);
        setHouses(res.data);
      };

      getData();
    }, []);
};

export default useFetch;
```

15. In **`useFetch`**, import **`useState`**, **`useEffect`**, and **`axios`**. And, since **`useFetch`** will be used by other components too, update the state value's more generic name to **`response`**. Also, update the endpoint by adding the request we received from the other component so that we can construct the different endpoint dynamically with the new **`value, ${request}`**. You will need to update the second argument to **`request`**. Finally, return the **`response`**:

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

const API_KEY = '12345';

const useFetch = request => {
  const [response, setResponse] = useState([]);

  useEffect(() => {
    const getData = async () => {
      const res = await axios.get(
        `https://www.potterapi.com/v1/${request}?key=${API_KEY
        }`
      );

      setResponse(res.data);
    };

    getData();
  }, [request]);
```

```
  return response;
};
export default useFetch;
```

16. Go back to **App.js** and use **useFetch** to get the house names again. As you no longer need **useState**, **useEffect**, and **axios** for now, let's remove the import statement for those dependencies. Instead, import **useEffect** and, above **return()** inside the **App** component, send **'houses'** as an argument to **useFetch**:

```
import React from 'react';
import useFetch from './useFetch';
import { colorConverter } from './utils';
import './App.css';

const App = () => {
  const houses = useFetch('houses');
};

export default App;
```

You should still see four buttons, with each house name and color.

Now, let's get all the characters from the initial rendering.

17. To display the characters, we are going to create a new component called **Characters**. In the **Characters** component, we are going to receive the endpoint as a prop called **searchCharacters** so that we can update the characters when clicking on one of the house buttons later. Let's use **useFetch** and send the **searchCharacters** value as an argument:

```
import React from 'react';
import useFetch from './useFetch';

const SearchResults = ({ searchCharacters }) => {
  const characters = useFetch(searchCharacters);

  console.log(characters);

  return <div className="character">Characters</div>;
};

export default SearchResults;
```

18. In the **App** component, let's define a state called **searchCharacters** and add **Characters** as an initial value. Later, we will update it when clicking on the house button. In **return()**, under the house div element, let's add the **Characters** component and send **searchCharacters** as a prop. And, of course, we need to import **useState** and the **Characters** component. Let's print the response in the **Characters** component:

```
import React, { useState } from 'react';
import useFetch from './useFetch';
import { colorConverter } from './utils';
import Characters from './Characters';
import './App.css';
const App = () => {
  const [searchCharacters, setSearchCharacters] =
  useState('characters');
  const houses = useFetch('houses');

  return (
    <div className="container">
      <div className="house">
        {houses.map(house => (
          ...
        ))}
      </div>
      <Characters searchCharacters={searchCharacters} />
    </div>
  );
};

export default App;
```

We should receive the data about the characters:

Figure 16.17: The data about the characters

19. Display the name of the characters. In **`return()`**, loop the characters and add a button with the name inside:

```
return (
  <div className="character">
    {characters.map(character => (
      <button className="character__box"
     key={character._id}>
        <div
       className="character__name">{character.name}</div>
      </button>
    ))}
  </div>
);
```

You should see the names appear under the buttons:

Figure 16.18: Displaying buttons and names of characters

20. Update the list of characters when clicking on one of the house buttons. Add **`onClick`** to the house button in the **`App`** component and reference to a function called **`updateHouse`**. Send the house **`ID`** to the function so that you can re-fetch the list of characters for the specific house. In **`updateHouse`**, update **`searchCharacters`** with **`houses/${id}`**:

```
const App = () => {
  const [searchCharacters, setSearchCharacters] =
  useState('characters');
  const houses = useFetch('houses');

  const updateHouse = id => {
    setSearchCharacters(`houses/${id}`);
  };

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

```
        <div className="house">
          {houses.map(house => (
            <button
              key={house._id}
              className="house__button"
              style={{
                backgroundColor:
                colorConverter(house.colors[0]),
                color: colorConverter(house.colors[1])
              }}
              onClick={() => updateHouse(house._id)}
            >
              {house.name}
            </button>
          ))}
        </div>
        <Characters searchCharacters={searchCharacters} />
      </div>
    );
  };

  export default App;
```

21. Now, when you click on one of the house buttons, you will receive the house name instead of the list of characters. This is because the data structures in the **houses** and **characters** endpoints are different. Therefore, we need to check whether\ we are getting the data from the house and, if so, retrieve the data from the **members** value; otherwise, display the data received from the character's endpoint directly:

```
import React from 'react';
import useFetch from './useFetch';

const SearchResults = ({ searchCharacters }) => {
  const characters = useFetch(searchCharacters);

  const displayChar = characters[0].members
    ? characters[0].members
    : characters;

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

```
        {displayChar.map(character => (
          <button className="character__box"
          key={character._id}>
      <div className="character__name">{character.name}</div>
    </button>
    ))}
    </div>
    );
};

export default SearchResults;
```

22. However, we will receive an error saying **`Uncaught TypeError: Cannot read property 'members' of undefined`**. This is because, upon initial rendering, we have no character value fetched yet, so there is nothing to get for the **`members`** value. Hence, we need to add a safeguard, so if there is no character data fetched yet, we will return an empty **`div`** element:

```
import React from 'react';
import useFetch from './useFetch';

const SearchResults = ({ searchCharacters }) => {
  const characters = useFetch(searchCharacters);

  if (!characters.length) return <div></div>;

  const displayChar = characters[0].members
    ? characters[0].members
    : characters;

  return (
    ...
  );
};

export default SearchResults;
```

23. Click on the button to update the list of characters based on the house. Finally, let's get the character's data, such as their role and house, when clicking on the character's name. For this, create another new component called **`CharacterDetails.js`**. In the **`CharacterDetails`** component, you will receive a prop called **`characterId`** and fetch the character's details using the **`useFetch`** hook. In **`return()`**, we will display the details using the list element:

```
import React from 'react';
import useFetch from './useFetch';

const CharacterDetails = ({ characterId }) => {
  const details = useFetch(`characters/${characterId}`);

  return (
    <ul>
      {details.role && <li>Role: {details.role}</li>}
      {details.house && <li>House: {details.house}</li>}
    </ul>
  );
};

export default CharacterDetails;
```

24. In the **`Characters`** component, import the **`CharacterDetails`** component. Define a state called **`characterId`** with an empty array as an initial value. Click on the character's name, add the ID to the **`characterId`** state, and loop the **`characterId`** state to display the character's details, which are only available in the **`characterId`** state. In the button, add **`onClick`** and reference a function called **`getDetails`** with **`characterId`** as an argument. In the **`getDetails`** function, we are going to add the ID to the **`characterId`** state:

`Characters.js`

```
1  import React, { useState } from 'react';
2  import useFetch from './useFetch';
3  import CharacterDetails from './CharacterDetails';
4
5  const SearchResults = ({ searchCharacters }) => {
6  const [characterId, setCharacterId] = useState([]);
7  const characters = useFetch(searchCharacters);
8
9  if (!characters.length) return <div></div>;
```

The complete code can be found here: https://packt.live/2T13SmI

25. Click on the name to get the details:

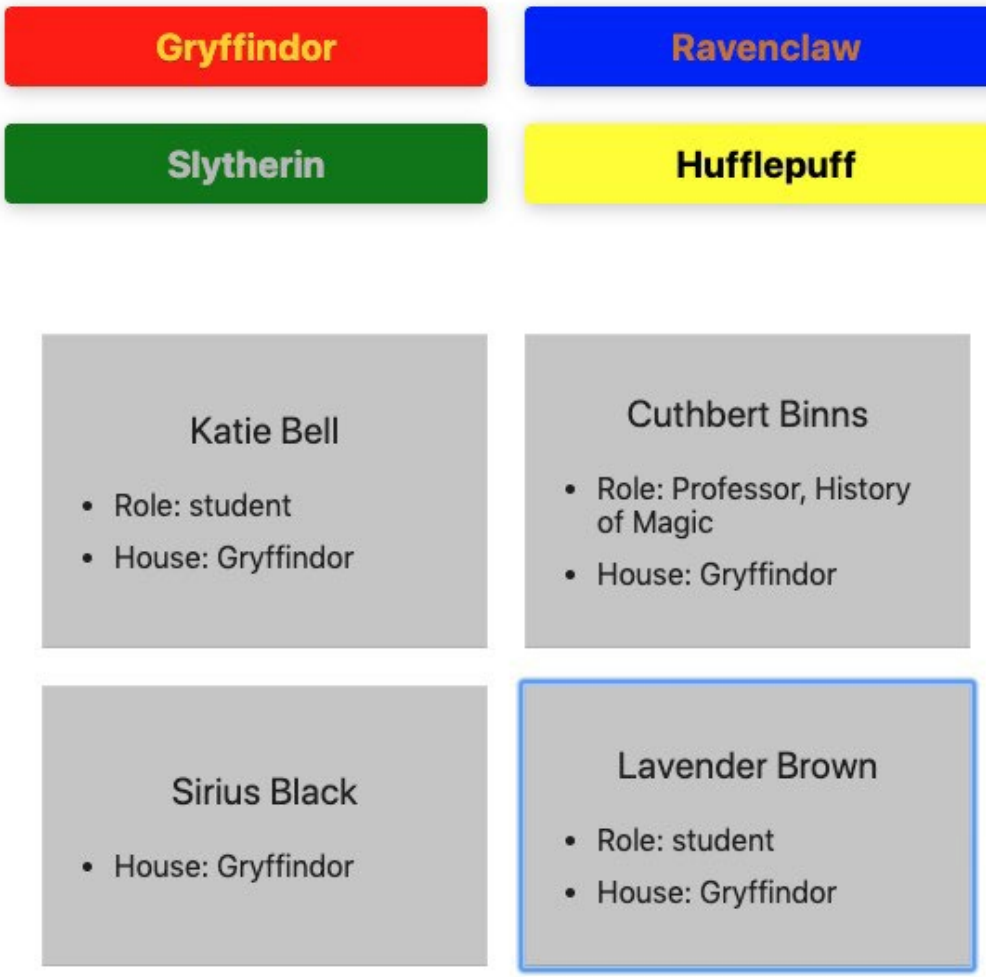

Figure 16.19: The final output

CHAPTER 17: REFS IN REACT

ACTIVITY 17.01: CREATING A FORM WITH AN AUTOFOCUS INPUT ELEMENT

Solution:

1. Create a new React app using the create-react-app CLI:

```
$ npx create-react-app autofocus-form
```

2. Move into the new React applications folder and start the **`create-react-app`** development server:

```
$ cd autofocus-form/ && npm start
```

3. Inside the **`App.js`** file, change the **`App`** component to only return a plain **`form`** component:

```
import React from "react";
class App extends React.Component {
  render() {
    return <form />;
  }
}
export default App;
```

4. Inside the **`form`** component, add three input fields – one for **`firstname`**, one for **`lastname`**, and a third one for **`email`**:

```
class App extends React.Component {
  render() {
    return (
      <form>
        <input type="text" placeholder="firstname" />
        <input type="text" placeholder="lastname" />
        <input type="text" placeholder="email" />
      </form>
    );
  }
}
```

5. Create a reference for an input field as a class field:

```
class App extends React.Component {
  inputRef = React.createRef();

  render() {
    return (
      <form>
        <input type="text" placeholder="firstname" />
        <input type="text" placeholder="lastname" />
        <input type="text" placeholder="email" />
      </form>
    );
  }
}
```

6. Pass the Ref to the first input field:

```
class App extends React.Component {
  inputRef = React.createRef();
  render() {
    return (
      <form>
        <input
          ref={this.inputRef}
          type="text"
          placeholder="firstname"
        />
        <input type="text" placeholder="lastname" />
        <input type="text" placeholder="email" />
      </form>
    );
  }
}
```

7. On **`componentDidMount`**, trigger the **`focus`** function of the current Ref:

```
class App extends React.Component {
  inputRef = React.createRef();

  componentDidMount() {
    this.inputRef.current.focus()
  }

  render() {
    return (
      <form>
        <input
          ref={this.inputRef}
          type="text"
          placeholder="firstname"
        />
        <input type="text" placeholder="lastname" />
        <input type="text" placeholder="email" />
      </form>
    );
  }
}
```

8. Create a new function component called **`FocusableInput`**, which can forward a reference. Make this component simply return null:

```
const FocusableInput = React.forwardRef(
    (props, forwardedRef) => null
)

class App extends React.Component {
  inputRef = React.createRef();

  componentDidMount() {
    this.inputRef.current.focus()
  }

  render() {
    return (
      <form>
```

```
          <input
            type="text"
            placeholder="firstname"
          />
          <input type="text" placeholder="lastname" />
          <input type="text" placeholder="email" />
        </form>
      );
    }
  }
```

9. Now, instead of **`null`**, the **`FocusableInput`** component should now return an **`input`** component that is of the **`text`** type. This input takes the **`placeholder`** prop passed to **`FocusableInput`**:

```
const FocusableInput = React.forwardRef(
    (props, forwardedRef) => {
      return (
        <input
          type="text"
        />
      )
    }
  )
```

10. This input field inside the FocusableInput component gets passed the **`placeholder`** prop:

```
const FocusableInput = React.forwardRef(
    (props, forwardedRef) => {
      return (
        <input
          type="text"
          placeholder={props.placeholder}
        />
      )
    }
  )
```

11. The input also gets passed the **`forwardedRef`** parameter as a Ref prop:

```
const FocusableInput = React.forwardRef(
    (props, forwardedRef) => {
      return (
        <input
          ref={forwardedRef}
          type="text"
          placeholder={props.placeholder}
        />
      )
    }
  )
```

12. The first input field within the App component should now be replaced with an instance of the **`FocusableInput`** component. This **`FocusableInput`** component should get passed both the reference and the placeholder that were passed to the original input field:

```
class App extends React.Component {
  inputRef = React.createRef();

  componentDidMount() {
    this.inputRef.current.focus()
  }
  render() {
    return (
      <form>
        <FocusableInput
          ref={this.inputRef}
          placeholder="firstname"
        />
        <input type="text" placeholder="lastname" />
        <input type="text" placeholder="email" />
      </form>
    );
  }
}
```

13. You could replace all native input fields in the **App** component with the **FocusableInput** component. Just make sure not to pass the same Ref to all **FocusableInput** components. By the way, the Ref prop on the **FocusableInput** component is optional; you don't have to pass it:

```
class App extends React.Component {
  inputRef = React.createRef();

  componentDidMount() {
    this.inputRef.current.focus();
  }

  render() {
    return (
      <form>
        <FocusableInput
          ref={this.inputRef}
          placeholder="firstname"
        />
        <FocusableInput placeholder="lastname" />
        <FocusableInput placeholder="email" />
      </form>
    );
  }
}
```

By now, your code should look similar to this:

```
import React from "react";
const FocusableInput = React.forwardRef(
(props, forwardedRef) => {
  return (
    <input
      ref={forwardedRef} type="text"
      placeholder={props.placeholder}
    />
  );
});

class App extends React.Component {
```

```
    inputRef = React.createRef();

    componentDidMount() {
      this.inputRef.current.focus();
    }
```

14. The **`render`** function should look like this:

```
    render() {
      return (
        <form>
          <FocusableInput
            ref={this.inputRef}
            placeholder="firstname"
          />
          <FocusableInput placeholder="lastname" />
          <FocusableInput placeholder="email" />
        </form>
      );
    }
  }

  export default App;
```

Simply running this code in your browser will display the following **`DOMRect`** object in the console:

Figure 17. 5: Component returning a text type placeholder

This last thing needed is to hide the ref and focus logic from the App component and encapsulate it in a proper component itself. Depending on where you pass the reference, you can now focus on any of the three input fields.

15. To do so, create a new functional component called **`FocusableForm`**. This component takes over all of the logic from the **`App`** component's **`render`** function:

```
const FocusableForm = props => {
  return (
    <form>
      <FocusableInput
        ref={inputRef}
        placeholder="firstname"
      />
      <FocusableInput placeholder="lastname" />
      <FocusableInput placeholder="email" />
    </form>
  );
}
```

The **`App`** component now only returns the **`FocusableForm`** component. Also, the **`componentDidMount`** function and the **`inputRef`** class field are no longer needed in the **`App`** component.

16. Update the **`App`** component:

```
class App extends React.Component {
  render() {
    return <FocusableForm />
  }
}
```

17. In order to access the input field in the **`FocusableForm`** component, you have to create a new reference and pass it to the first input field:

```
const FocusableForm = props => {
  const inputRef = React.useRef();

  return (
    <form>
      <FocusableInput
        ref={inputRef}
```

```
          placeholder="firstname"
        />
        <FocusableInput placeholder="lastname" />
        <FocusableInput placeholder="email" />
      </form>
    );
  }
```

18. Now, use the **`useEffect`** hook to trigger a focus on the current **`inputRef`**:

```
const FocusableForm = props => {
  const inputRef = React.useRef();

  React.useEffect(() => {
      inputRef.current.focus()
  })

  return (
    <form>
      <FocusableInput
        ref={inputRef}
        placeholder="firstname"
      />
      <FocusableInput placeholder="lastname" />
      <FocusableInput placeholder="email" />
    </form>
  );
}
```

19. Last but not least, make the **`FocusableForm`** component controllable by passing an **`autoFocus`** prop from the App component to **`FocusableForm`** and using the prop in the **`useEffect`** hook to decide whether to focus the reference:

```
const FocusableForm = props => {
  const inputRef = React.useRef();

  React.useEffect(() => {
    if (props.autoFocus) {
      inputRef.current.focus()
    }
  })
```

```
    return (
      <form>
        <FocusableInput
          ref={inputRef}
          placeholder="firstname"
        />
        <FocusableInput placeholder="lastname" />
        <FocusableInput placeholder="email" />
      </form>
    );
  }

  class App extends React.Component {
    render() {
      return <FocusableForm autoFocus={true} />
    }
  }
```

20. You can now control the auto-focus functionality in the parent **App** component, and the **FocusableForm** component encapsulates and hides the logic to actually trigger the focus on the reference:

App.js

```
3  export const FocusableInput = React.forwardRef(
    (props, forwardedRef) => {
4    return (
5    <input ref={forwardedRef} type="text"
   placeholder={props.placeholder}/>
6  );
7  });
8
9  const FocusableForm = props => {
10  const inputRef = React.useRef();
11
12  React.useEffect(() => {
13    if (props.autoFocus) {
14      inputRef.current.focus()
15    }
16  })
```

The complete code can be found here: https://packt.live/2LFlcd7

Simply running this code in your browser will display the following **`DOMRect`** object to the console:

Figure 17. 6: Final output UI

CHAPTER 18: PRACTICAL USE CASES OF REFS

ACTIVITY 18.01: PORTABLE MODALS USING REFS

Solution:

1. Add the **`App`** base component:

```
import React from "react";
import ReactDOM from "react-dom";
import "./App.css";
//omitting the styles for brevity
class App extends React.Component {
  viewportRef = React.createRef();
  state = {
    showModal: false
  };
  render() {
    return null
  }
}
export default App;
```

2. Add the **`ModalOverlay`** and Modal components:

```
import React from "react";
import ReactDOM from "react-dom";
import "./App.css";
const Modal = props => {
  return <div className="Modal">{props.children}</div>;
};
class ModalOverlay extends React.Component {
  render() {
    const { mountingPoint, showModal, onCloseHandler } = this.props;

    return null;
  }
}
class App extends React.Component {
  viewportRef = React.createRef();
  state = {
    showModal: false
```

```
    };
    render() {
      return null;
    }
  }
  export default App;
```

3. Extend the **`Modal`** component already given in *step 2*.
4. Create a **`PortalModal`** component that clones its children and passes an **`onClick`** prop to each of them:

```
  // omitted for brevity
  // inside the ModalOverlay render method
        const ModalPortal = (
          <div className="ModalOverlay">
           <Modal>
            {React.Children.map(this.props.children, child =>
              React.cloneElement(child, {
                onClick: onCloseHandler
                })
              )}
            </Modal>
          </div>
        );
        return ReactDOM.createPortal(ModalPortal,
        mountingPoint);
  // omitted for brevity
```

5. Only return the portal if the **`Modal`** component is supposed to be shown. Otherwise, render nothing:

`App.js`

```
1  import React from "react";
2  import ReactDOM from "react-dom";
3
4  const Modal = props => {
5  return <div className="Modal">{props.children}</div>;
6  };
7  export class ModalOverlay extends React.Component {
8  render() {
9    const { showModal, onCloseHandler, mountingPoint } = this.props
```

The complete code can be found here: https://packt.live/3dOSoe9

6. Extend the **`App`** component with the **`toggleModal`** method. Also, use the skeleton to include **`ModalOverlay`** and pass all relevant props to it:

```
// omitted for brevity
class App extends React.Component {
  viewportRef = React.createRef();
  state = {
    showModal: false
  };
  toggleModal = () => this.setState({
    showModal: !this.state.showModal
  });
  render() {}
}
// omitted for brevity
```

7. Add the **`render`** method:

```
// omitted for brevity
class App extends React.Component {
  viewportRef = React.createRef();
  state = {
    showModal: false
  };
  toggleModal = () => this.setState({
     showModal: !this.state.showModal
  });
render() {
  return (
    <div style={appStyles}>
      <button /> { /* open button */ }

      <div className="Viewport" />

      <ModalOverlay>
        { /* close button */ }
      </ModalOverlay>
    </div>
  );
}
// omitted for brevity
```

The **`App.css`** class is on the outermost **`div`** element.

8. Create an event handler for the button:

```
// omitted for brevity
class App extends React.Component {
  viewportRef = React.createRef();
  state = {
    showModal: false
  };
  toggleModal = () => this.setState({
  showModal: !this.state.showModal
  });
render() {
  return (
    <div className="App">
      <button onClick={this.toggleModal}>Show modal!</button>
      <div className="Viewport"} />
      <ModalOverlay>
        { /* close button */ }
      </ModalOverlay>
    </div>
  );
}
// omitted for brevity
```

9. Add the viewport container and pass it the **`viewportRef`** object:

```
// omitted for brevity
class App extends React.Component {
  viewportRef = React.createRef();
  state = {
    showModal: false
  };
  toggleModal = () => this.setState({
  showModal: !this.state.showModal
  });
render() {
  return (
    <div className="App">
      <button onClick={this.toggleModal}>Show modal!</button>
      <div ref={this.viewportRef} className="Viewport">
```

```
              Lorem Ipsum
           </div>
          <ModalOverlay>
            { /* close button */ }
          </ModalOverlay>
        </div>
      );
    }
    // omitted for brevity
```

10. Pass the relevant props and content to the **`ModalOverlay`** component:

`App.js`

```
// omitted for brevity
31  class App extends React.Component {
32  viewportRef = React.createRef();
33  state = {
34    showModal: false
35  };
36  toggleModal = () => this.setState({
     showModal: !this.state.showModal
37    });
37  render() {
38  return (
39    <div className="App">
40      <button onClick={this.toggleModal}>Show
        modal!</button>
```

The complete code can be found here https://packt.live/3bxfzb2

11. Disable the scroll functionality when the modal is shown (and enable it as soon as the modal is hidden):

`App.js`

```
// omitted for brevity
7  export class ModalOverlay extends React.Component {
8  render() {
9    const { showModal, onCloseHandler, mountingPoint } =
     this.props;
…
12    if (showModal) {
13      mountingPoint.style.overflowY = "hidden";
14      const ModalPortal = (
15        <div className="ModalOverlay">
16          <Modal>
17            {React.Children.map(this.props.children, child=>
18              React.cloneElement(child, {
```

The complete code can be found here https://packt.live/2Wv0eU9

12. Create a fallback mounting point for the **`createPortal`** function inside **`ModalOverlay`**:

```
// omitted for brevity
class ModalOverlay extends React.Component {
  render() {
    const { showModal, onCloseHandler } = this.props;
    const fallbackSelector = document.querySelector("body");
    const mountingPoint = this.props.mountingPoint ||
    fallbackSelector;
    if (showModal) {
      mountingPoint.style.overflowY = "hidden";

      const ModalPortal = (
        <div className="ModalOverlay">
          <Modal>
            {React.Children.map(this.props.children, child =>
              React.cloneElement(child, {
                onClick: onCloseHandler
              })
            )}
          </Modal>
        </div>
      );
      return ReactDOM.createPortal(ModalPortal, mountingPoint);
    }
    mountingPoint.style.overflowY = "scroll";
    return null;
  }
}
```

The final code should be as follows:

App.js

```
1  import React from "react";
2  import ReactDOM from "react-dom";
4  const Modal = props => {
5  return <div className="Modal">{props.children}</div>;
6  };
7  export class ModalOverlay extends React.Component {
8  render() {
9    const { showModal, onCloseHandler } = this.props
10    const fallbackSelector = document.querySelector("body");
11    const mountingPoint = this.props.mountingPoint ||
```

The complete code can be found here: https://packt.live/3dOSoe9

When you run this code and switch between passing a **mountingPoint** prop and not doing so, you will get two different outcomes. If you choose to pass **viewportRef** as a mounting point, the overlay should only cover the yellow **div** and still permit scrolling on the body:

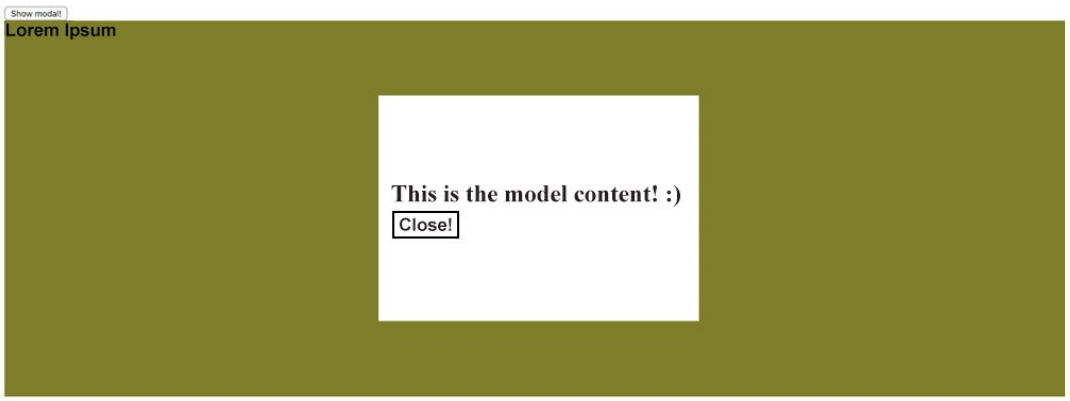

Figure 18.7: Overlay covering the yellow div element

However, in the case of a **`null`** or missing **`mountingPoint`** prop, our application will fall back to a global DOM body overlay and cover the entire application, which, in turn, would also disable scrolling on the entire page, and not just on the yellow **`div`**:

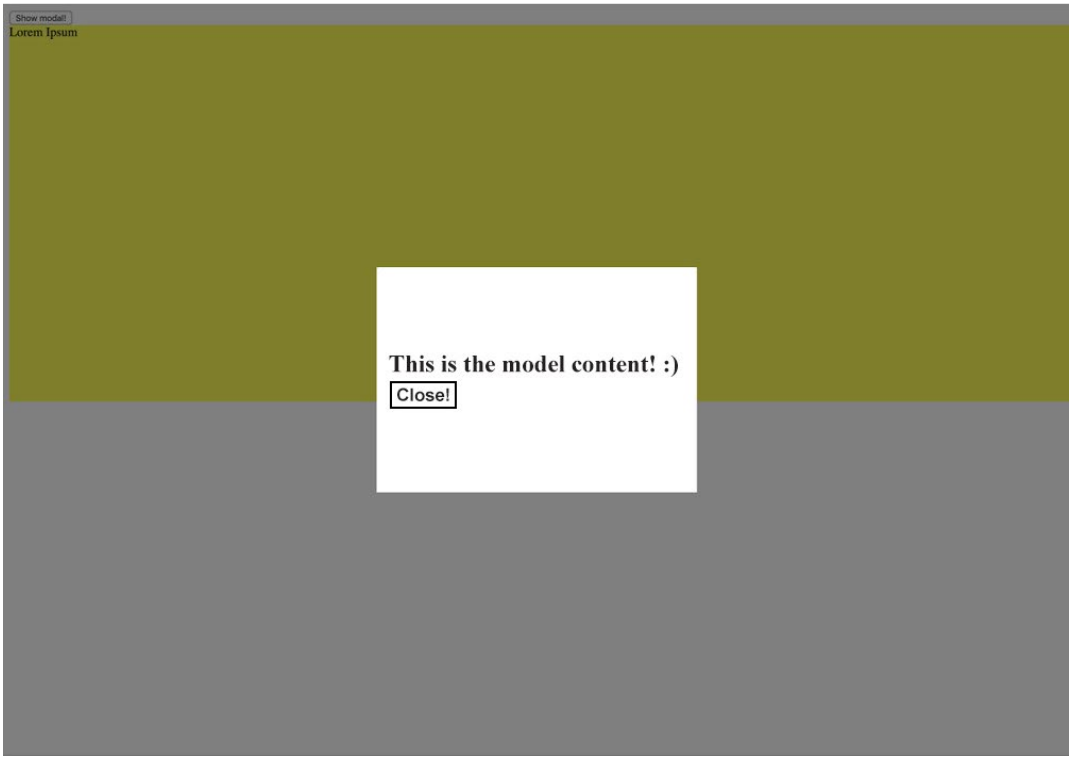

Figure 18.8: Overlay covering the entire application

INDEX

N

O

P

R

S

T

U

V

W

Y