

ReactJS

— for —

Jobseekers

The Only Guide You Need to Learn React and Crack Interviews



Qaifi Khan

bpb

ReactJS

— for —

Jobseekers

The Only Guide You Need to Learn React and Crack Interviews



Qaifi Khan

bpb

ReactJS

for

Jobseekers

*The Only Guide You Need to Learn
React and Crack Interviews*

Qaifi Khan



www.bpbonline.com

Copyright © 2023 BPB Online

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

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

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

First published: 2023

Published by BPB Online

WeWork

119 Marylebone Road

London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-5551-342-7

www.bpbonline.com

Dedicated to

My beloved parents

Khaleel Ahmad

Roobeena Begam

&

*My sisters **Summaiya** and **Laiba***

About the Author

Qaifi is a front-end developer, product designer, and tech evangelist. He has worked with numerous technologies over the years. Currently, he is working with HTML, CSS, JavaScript, and React. He absolutely loves sharing his knowledge with people interested in front-end development. He has more than 3000 hours of fresher training experience. He also has online courses with more than 1.7 million enrollments.

About the Reviewer

Sandeep Konakanchi, a Technocrat and Tech Enthusiast. He is working as a Senior Software Engineer and has multi-scripting language experience. Having graduated with a Masters's Degree in Computer and Information Systems department, he always brainstorms to become an entrepreneur by himself. He helps newcomers start their careers in the programming world. Besides working, he always does research and learns new technologies and trends in the programming world. He is also certified with Amazon Web Services for cloud computing.

Acknowledgement

Special thanks to the team at BPB Publications for being supportive enough to provide me with all the time it took to complete the book. They have been incredibly helpful with all the reviews and feedback to ensure we create this quality book that will help our readers.

I could not have undertaken this journey without my parents. Words cannot express my gratitude towards them for their continuous support and encouragement in writing the book — I could have never completed this book without their support.

Preface

You talk about frontend development, and the first thing that comes to any tech evangelist's mind is the latest frontend frameworks like ReactJS, VueJS, or even Angular. If you are in the tech industry or interested in frontend development, then I'm sure you must have heard about at least one of these frameworks. Based on its popularity and industry acceptance, ReactJS is leading by miles. It is one of the most popular front-end libraries for building single-page applications. It is used by some of the biggest names like Facebook, Netflix, Instagram, Airbnb, Nike, and so on.

In this book, we will learn everything you need to know about ReactJS to start working as a front-end developer. We will talk about the core concepts like components, state, props, lifecycle, and hooks, which will get you comfortable with the ReactJS ecosystem. We will discuss additional topics like routing, connecting to the backend, and handling state using Redux to give you a more holistic understanding of building production-level applications using ReactJS.

This book is structured into 3 sections. The first section is wholly focused on the basic and advanced concepts of React. The second section is focused on UI/UX concepts. The last section is focused on interview preparation. The details are listed as follows.

Chapter 1 will cover what is meant by Web Development. We will learn the difference between Frontend and Backend. We will look at different libraries and frameworks available for frontend and backend development. We will see how the front end is connected to the backend. We will learn the core features of ES6, which are recommended for writing code in React. These features are going to help you write code faster and cleaner.

Chapter 2 is where we make our first dive into React. We will understand what ReactJS is and why it is so popular. We will talk about a JavaScript syntax extension called JSX, which is heavily used to write components in ReactJS.

Chapter 3 will cover components and their types. We will dive deep into class-based and functional components. We will learn about handling

component data using state and how to pass data from one component to another using props. We will create our first component and pass custom values to it. We will learn about CSS modules and how they help us avoid all the challenges of global and inline styles. We will talk about both named and default exports. We will learn how to import methods across JavaScript files.

[**Chapter 4**](#) is all about understanding what happens when the render() method is called and how each component undergoes different stages of its lifecycle. We will learn how to utilize these lifecycle stages to implement functionalities in Class-based and Functional components.

[**Chapter 5**](#) covers how to connect our React app to the backend using packages like Axios. We will learn how to make different API requests and handle their responses.

[**Chapter 6**](#) is where we learn about the new Hooks feature introduced in React 16.8. We will talk about different hooks made available by the React library and we will implement the most commonly used ones.

[**Chapter 7**](#) will cover how to load different pages on different URLs of our web app. We will learn about React Router and how it can help us with handling different routes. We will see how to pass and read data from URLs. We will also learn how to handle exception pages like 401, 404, and 500.

[**Chapter 8**](#) will help us understand the difference between controlled and uncontrolled components. We will learn how to access DOM objects of elements using Ref. We will create our first form and also learn how to generate form elements dynamically.

[**Chapter 9**](#) is where we are introduced to Redux and how it works behind the scenes. We will understand how to install and set up Redux in a React app. We will create a global store and its reducer to manage state changes. We will learn how to dispatch actions and handle the state changes accordingly.

[**Chapter 10**](#) will cover different tools which work behind the scenes to run our React app. We will see how NPM is used to manage packages. We will learn about Webpack and how it bundles all our JavaScript and CSS files into chunks to help reduce load speed. We will learn how to create a production build of our app and how to host it.

Chapter 11 is where we learn how we introduce issues in our application that result in loss of performance. These issues could be due to bad code, usage of heavy images, unoptimized animations, slow APIs, not splitting the build files, and many more reasons. In this chapter, we will learn about application performance and how to optimize it. There are multiple ways it can be achieved but first, we need to learn how to measure and benchmark it.

Chapter 12 is our introduction to UI/UX. We will talk about different areas in UI/UX design. We will build a strong foundation here by understanding the principles of UI/UX design. We will learn how to use Figma for creating wireframes, designs, and prototypes.

Chapter 13 covers some of the trending design patterns like glassmorphism and neomorphism to add more depth to our page designs, and pastel backgrounds for a very clean and minimalistic look. We will also learn about the dark and light theming of a webpage.

Chapter 14 is our last chapter but an important one. This is where we will talk about the interview processes for the front-end developer role using ReactJS. We will also go through some commonly asked interview questions with more emphasis on the topics which are frequently asked in the interviews.

Code Bundle and Coloured Images

Please follow the link to download the *Code Bundle* and the *Coloured Images* of the book:

<https://rebrand.ly/2g00g5w>

The code bundle for the book is also hosted on GitHub at <https://github.com/bpbpublications/ReactJS-for-Jobseekers>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at <https://github.com/bpbpublications>. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePUB files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book

customer, you are entitled to a discount on the eBook copy. Get in touch with us at: business@bpbonline.com for more details.

At www.bpbonline.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

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

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit www.bpbonline.com. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit www.bpbonline.com.

Table of Contents

1. Introduction to Web Development

Structure

Objectives

Frontend versus backend

Modern JavaScript for ReactJS

Creating variables using let and const

Templates strings

Arrow functions

Rest property

Spread property

Destructuring

New array function: map()

New array function: reduce()

New array function: filter()

New array functions: find() and findIndex()

Classes, properties, and methods

Inheritance

Conclusion

Questions

Multiple choice questions with answers

Answers

References

2. Up and Running with React Ecosystem

Structure

Objectives

Introducing ReactJS

Component-based design

Single-page versus multi-page Web apps

Creating our first React project

Introducing JSX

Inline and external styles

[Rendering dynamic elements using objects and lists](#)

[Conclusion](#)

[Questions](#)

[Multiple choice questions](#)

[Answers](#)

3. Understanding Components, State, and Props

[Structure](#)

[Objectives](#)

[Introduction to components](#)

[Understanding props](#)

[Imports and exports](#)

[CSS modules](#)

[Responsive components](#)

[Stateful and stateless components](#)

[Class-based components](#)

[Passing props to components](#)

[Conclusion](#)

[Questions](#)

[Multiple choice questions with answers](#)

[Answers](#)

4. Lifecycle of Components

[Structure](#)

[Objectives](#)

[DOM versus virtual DOM](#)

[Component lifecycles](#)

[Mounting lifecycle methods](#)

[Updating lifecycle methods](#)

[Unmounting lifecycle method](#)

[Conclusion](#)

[Questions](#)

[Multiple choice questions with answers](#)

[Answers](#)

5. Connecting to Backend

[Structure](#)

Objectives

The purpose of backend

What is HTTP?

API Endpoints

HTTP methods

HTTP response codes

Intro to Axios

Making our first API request

POST request using Axios

PUT request using Axios

DELETE requests using Axios

Global Axios setup

Intercept request and response

Scalable code architecture for network requests

Conclusion

Questions

Multiple choice questions with answers

Answers

6. React Hooks

Structure

Objectives

Introduction to Hooks

Converting a class-based component to functional with Hooks

Hooks provided by React

useState() Hook

useEffect() Hook

Creating custom hooks

Conclusion

Questions

Multiple choice questions with answers

Answers

7. Routing in React Apps

Structure

Objectives

What is routing?

[Introduction to React-Router](#)
[React-Router setup in React app](#)
[Handling dynamic URLs](#)
[Hooks provided by React Router](#)
[Conditional redirect](#)
[Handling query params in URLs](#)
[Handling 401, 404, and 500 pages](#)
[Conclusion](#)
[Questions](#)
[Multiple choice questions with answers](#)
[Answers](#)

8. Controlled and Uncontrolled Components

[Structure](#)
[Objectives](#)
[Controlled versus uncontrolled components](#)
 [Forms using uncontrolled components](#)
 [Forms using controlled components](#)
[Keeping the users logged in](#)
[Callback functions and callback hell](#)
[Introduction to promises](#)
 [Create a promise](#)
 [Chaining multiple promises](#)
[Create asynchronous functions using `async` and `await`](#)
 [Restructuring our existing code using `async` and `await`](#)
 [Refactor the project structure](#)
[Conclusion](#)
[Questions](#)
[Multiple choice questions with answers](#)
[Answers](#)

9. State Management Using Redux

[Structure](#)
[Objectives](#)
[Introduction to Redux](#)
 [Global store](#)
 [Actions](#)

[Reducer](#)

[Redux installation](#)

[Configure Redux in a React app](#)

[Create a global store](#)

[Fetch state from the global store in components](#)

[Update global store from components](#)

[Handle multiple reducers](#)

[Implementing Redux using Hooks](#)

[Action creators](#)

[Async actions using middleware](#)

[Conclusion](#)

[Questions](#)

[Multiple choice questions with answers](#)

[Answers](#)

10. Production Build and Hosting React Apps

[Structure](#)

[Objectives](#)

[Production build](#)

[Host React applications](#)

[Introduction to Webpack](#)

[Conclusion](#)

[Questions](#)

[Multiple choice questions with answers](#)

[Answers](#)

11. Performance Optimization

[Introduction](#)

[Structure](#)

[Objectives](#)

[Why measure performance?](#)

[React Profiler](#)

[useMemo\(\)](#)

[useCallback\(\)](#)

[PureComponents](#)

[shouldComponentUpdate\(\)](#)

[Code splitting](#)

[Conclusion](#)

[Questions](#)

[Multiple choice questions with answers](#)

[Answers](#)

[12. Starting with Tools and Concepts of UI/UX](#)

[Structure](#)

[Objectives](#)

[Introduction to UI/UX](#)

[UI/UX tools](#)

[Different stages involved in UI/UX projects](#)

[User research](#)

[Information architecture](#)

[Wireframe](#)

[Design](#)

[Prototype](#)

[Fundamental concepts](#)

[White space](#)

[Contrast](#)

[Scale](#)

[Alignment](#)

[Colors](#)

[Typography](#)

[Visual hierarchy](#)

[Conclusion](#)

[Questions](#)

[13. Trending UI Patterns](#)

[Structure](#)

[Objectives](#)

[Glassmorphism](#)

[Neomorphism](#)

[Trying out soft gradients](#)

[Work with geometric elements](#)

[Pastel backgrounds](#)

[Designing dark mode](#)

[Conclusion](#)

Questions

14. Prepping for React Interviews

Structure

Objectives

React Interview Process

Resume Template

Preparation Material

Easy Questions

Intermediate Questions

Hard Questions

Easy Questions [Solutions]

Intermediate Questions [Solutions]

Hard Questions [Solutions]

Index

CHAPTER 1

Introduction to Web Development

In this chapter, we are going to learn about the fundamental Web development concepts. So far, you must have used the ES5 version of JavaScript. When we start working with ReactJS, we will write a lot of modern JavaScript, which includes features from ES6 and newer versions. We will learn those concepts as well in this chapter. If you are strongly comfortable with the ES6 concepts then you can skip this chapter or just skim through it.

Structure

In this chapter, we will discuss the following topics:

- Introduction to Web development
- Frontend versus backend
- Libraries/Frameworks for frontend and backend development
- 3-tier Web app architecture
- Intro to ES6 for ReactJS

Objectives

After studying this chapter, you will be able to understand the difference between frontend and backend, the purpose of HTML, CSS, and JS in frontend development, how Web apps function behind the scenes, and basic ES6 features such as scoped variable creation, array functions, rest and spread operators, and destructuring.

Frontend versus backend

Let us start by understanding what you see versus what happens behind the scenes. The Web apps that you see loaded on your browser screen are called **frontend apps**. The Web apps which control what data to be loaded on the

frontend apps are called **backend apps**. The space where all your data is stored is called a **database**.

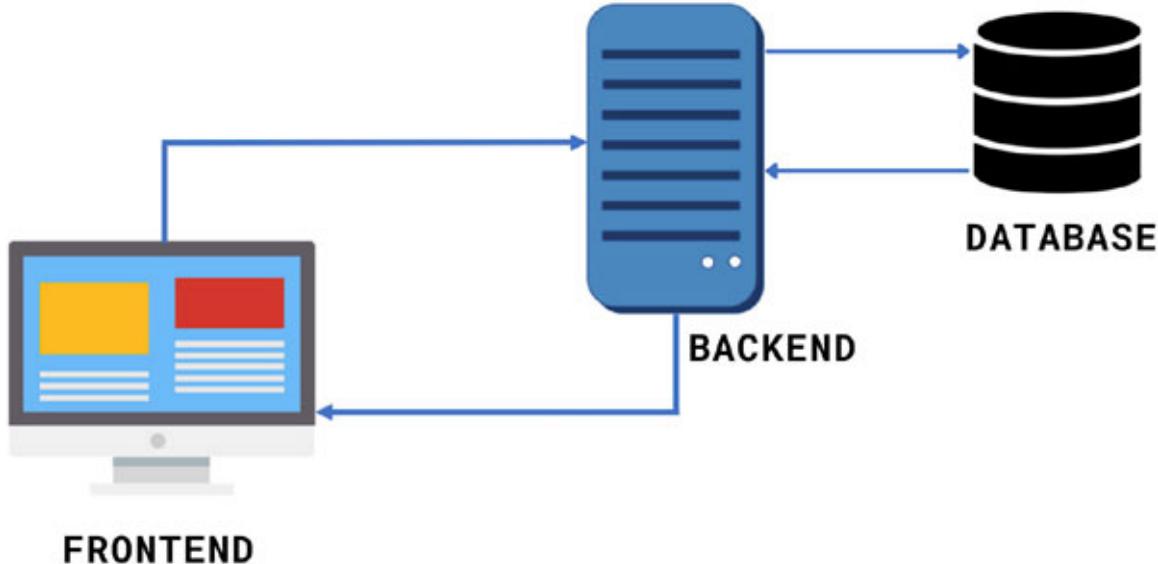


Figure 1.1: 3-tier architecture

Let us understand it better with an example. Look at YouTube; when you open the homepage, it shows you a grid of videos, as shown here:

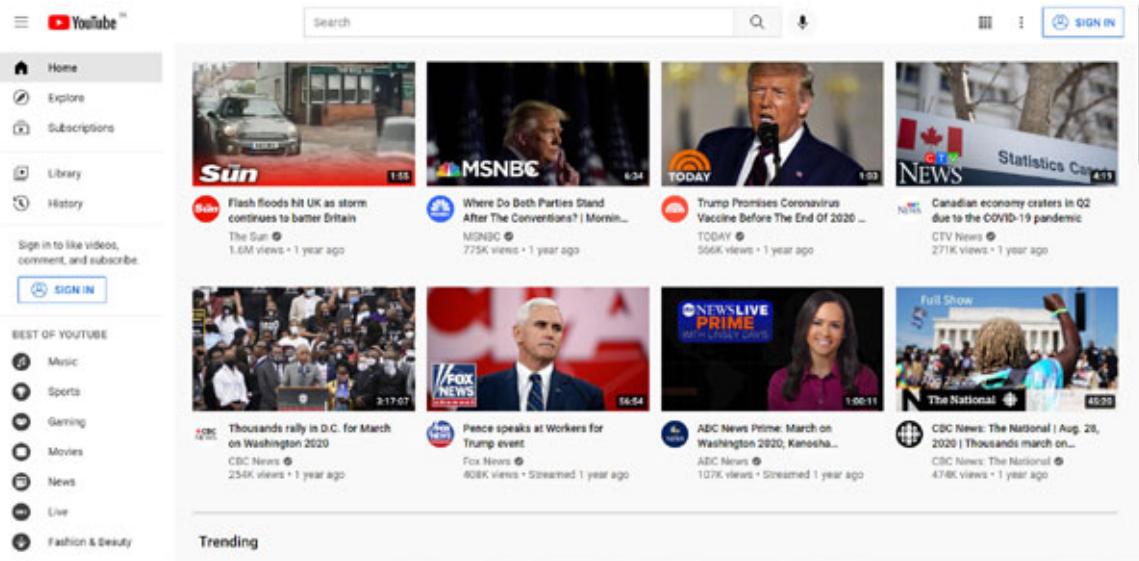


Figure 1.2: YouTube landing page

And every time you open the homepage, it shows different grids as follows:

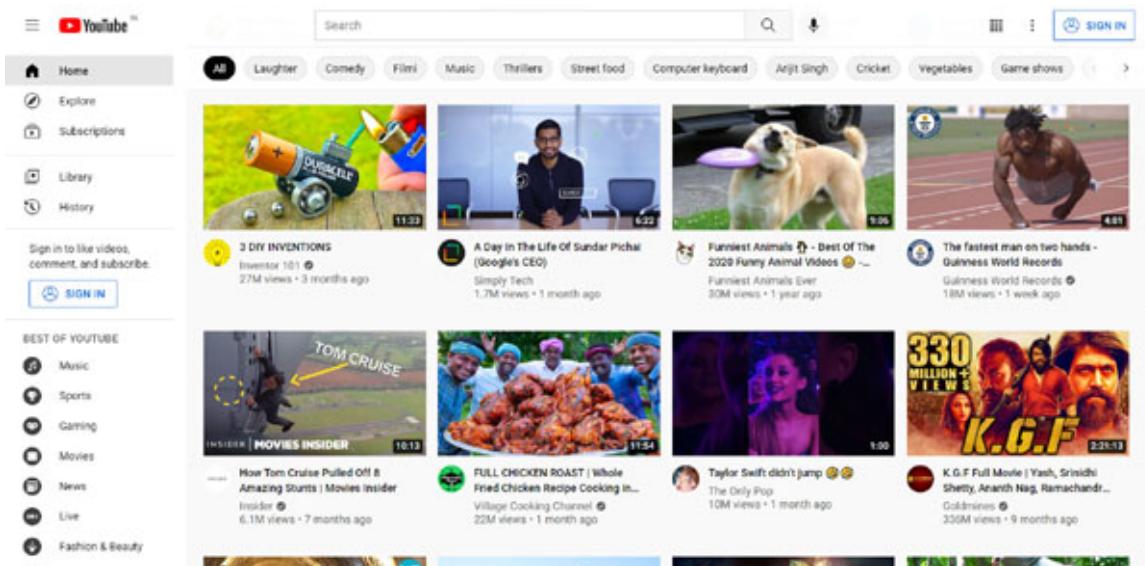


Figure 1.3: YouTube landing page—after refresh

The grid and the video cards, basically the entire page that you see, are the frontend part. The videos to be shown to you on every page load is decided by another application, which is the backend application. I hope that clears the difference between frontend and backend applications.

At the core, we have three Web development technologies—**HTML**, **CSS**, and **JavaScript**. Just by learning these three, you can build any frontend application.

HTML is used to create the structure of a Webpage. Structure refers to all the elements that you see on a Webpage. For example, images, headings, buttons, links, and so on.

```

<head>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link href="https://fonts.googleapis.com/css?family=Dr+Sugiyama" rel="stylesheet">
  <link href="https://fonts.googleapis.com/css?family=Lato:300,400" rel="stylesheet">
  <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.7.2/css/all.css" integrity="sha384-fnnOJmEhpdqtmkP0L2sZJLQ2/JhE9hsRZT2kZl8ZGqRfF5jw==" crossorigin="anonymous">
  <title>My First Site</title>
</head>

```

Figure 1.4: HTML layout with code

CSS is used for styling, formatting, and designing layouts on a Webpage. For example, the font size of the heading, background color of the page, aligning elements on the left and right side of the same row, all this, and much more can be achieved using CSS, as shown in the following figure:

```

body {
  margin: 0;
}

/** FIRST FOLD START **/ 

#header-section {
  background-color: #4eacdd;
  width: 100%;
  height: 100vh;
  padding: 0 12% 0 12%;
  box-sizing: border-box;
  position: relative;
  overflow: hidden;
}

#header-text-section {
  text-align: end;
  margin-top: 15vh;
}

```

Figure 1.5: CSS layout with code

JavaScript is used to add functionalities to our Web pages. It is used to:

- Render HTML dynamically
- Change CSS styles programmatically
- Design beautiful animations

For example, consider the following figure showing a login popup on clicking the login button:

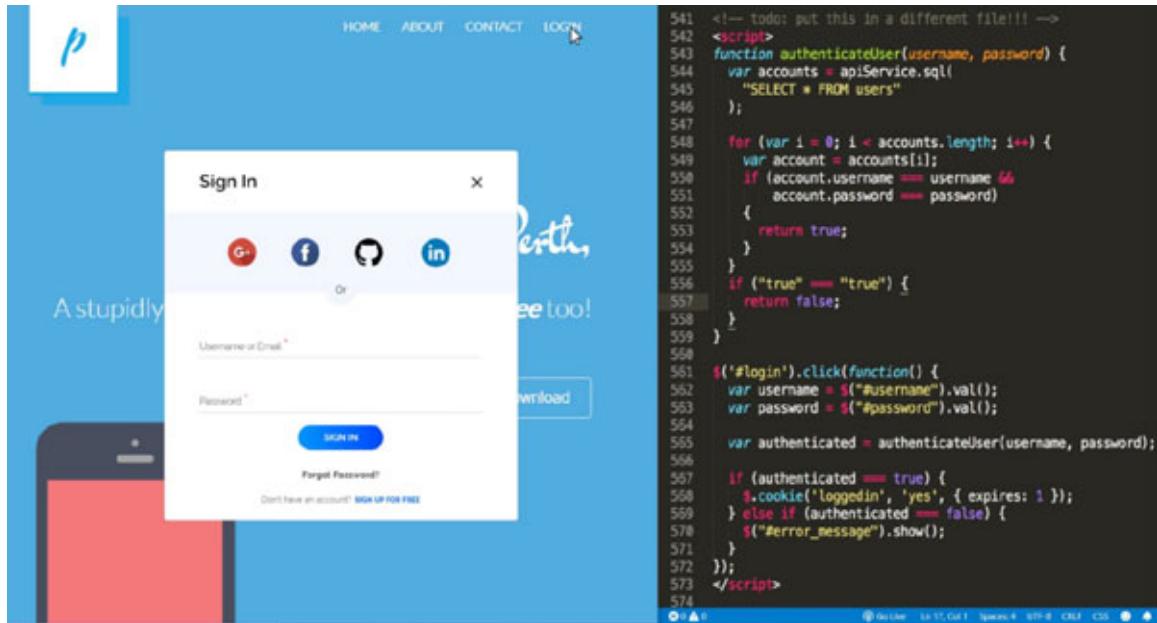


Figure 1.6: JavaScript layout with code

Now, the question arises, if we can create any Web app using these three, then why do we have frameworks and libraries such as ReactJS, AngularJS, VueJS, and so on. The simple or naive answer would be that these frameworks make building complex Web apps easy and fast. They save us time and effort by bringing a lot of core functionalities out of the box.

Some of the popular frontend frameworks and libraries include the following:

1. ReactJS
2. AngularJS
3. VueJS
4. Bootstrap
5. jQuery and many more

Some of the popular backend frameworks and libraries include the following:

1. ExpressJS

2. NextJS
3. MeteorJS
4. Django
5. ASP.NET, and many more

Another quick question, how do you decide which one to choose? Trust me this is one of the most debatable questions on any tech platform. I personally think that any frontend or backend application can be built using any of these applications, but not all will give you the same results. Every framework will have its pros and cons in terms of speed of project completion, learning curve, code maintenance, scalability, and so on. There is no one that fits all solution here. Based on the kind of application you are trying to build, you will have to choose the framework or library.

In this book, we are going to learn how to build fast and scalable single-page applications using a library called ReactJS.

Modern JavaScript for ReactJS

JavaScript has had many versions over the years, and ES6 is one of the JavaScript versions, as shown here:

ECMASCRIPT VERSIONS



ECMASCRIPT 1

First Edition



ECMASCRIPT 2

Editorial Changes



ECMASCRIPT 3

Added Regular Expressions, try/catch, switch, do-while.



ECMASCRIPT 5

Added strict mode, JSON support, Array iteration methods.



ECMASCRIPT 6

Added let and const, default parameter values, Array methods - find() and findIndex()



ECMASCRIPT 7

Added exponential operator, Array.includes () methods



ECMASCRIPT 8

Added Object.entries (), Object.values (), async functions and shared memory



ECMASCRIPT 9

Added rest/spread properties, asynchronous iteration, Promise.finally()

Figure 1.7: ECMAScript versions

ECMAScript is the standardized name for JavaScript. It is a major enhancement to the JavaScript language and adds many more features intended to make large-scale software development easier. We will learn some of the most commonly used features released in ES6 and in newer versions. As you can see, ES6 was released in 2015, and ES6 refers to version six of the ECMAScript programming language.

Some of these features are as follows:

1. Block-scoped variable creation using “let” and “const”
2. Template strings
3. Arrow functions
4. Rest and spread operator
5. Destructuring
6. Array functions: map(), filter(), reduce(), find(), and findIndex()
7. Template strings
8. Classes
9. Promises, and much more.

Even today, a lot of features of ES6 and newer versions are not supported by all browsers. Some of these features might not work in your browser directly.

So, what do we do?

We can use a preprocessor like Babel. Babel is mainly used to convert ES6+ code into a backward-compatible version of JavaScript that can run on older browsers.

Let us check it out.

When we write some ES6 code, Babel converts it to an older version of JS, which is supported by the browser. Do not worry if you do not understand this ES6 code. We will learn all about it in this chapter.

ES6 Code:

```
const getFullName = (fName, lName) => fName + " " + lName
```

The preceding code is written using arrow functions provided by ES6. The function takes two arguments, “**fName**” and “**lName**” and returns the concatenated value.

ES5 code:

```
"use strict";
var getFullName = function(fName, lName) {
    return (fName + " " + lName)
}
```

Okay, so this is what Babel does. But how do we use it in our code??

Well, for practice, the easiest way is Codepen:

Let us give it a try. Follow the given steps:

1. Go to Codepen and create a new pen, as shown in the following figure. Alternatively, you can open the following URL in your browser:
<https://codepen.io/pen/>

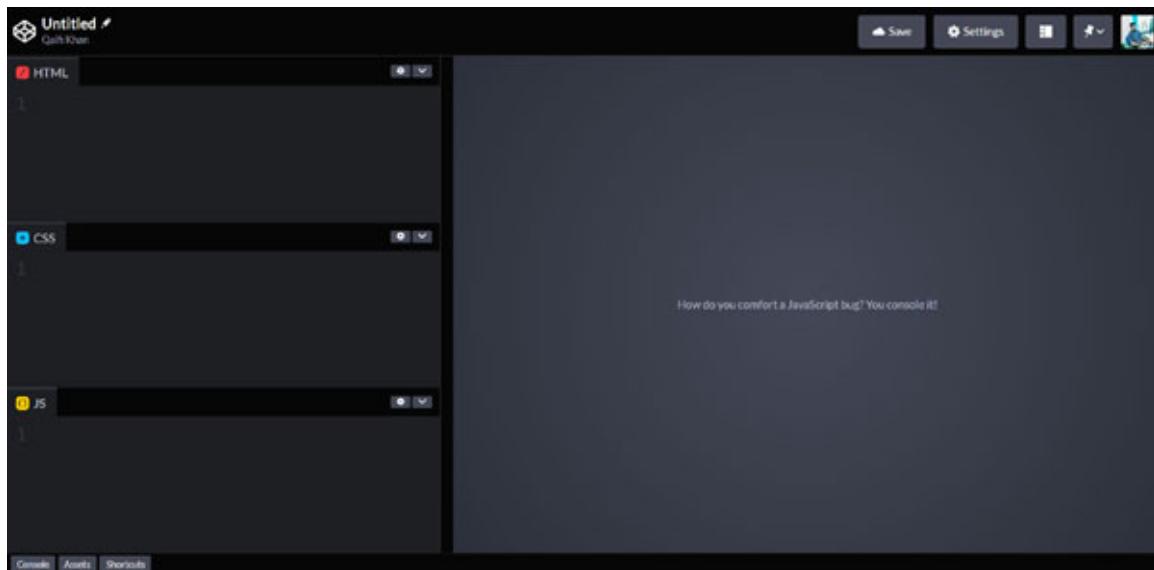


Figure 1.8: New pen on Codepen

2. Click on the **Settings** icon of the JavaScript section. It will open a popup where it gives options to configure your HTML, CSS, and JS code. Click on **JS** to open the JS config section, as shown here:

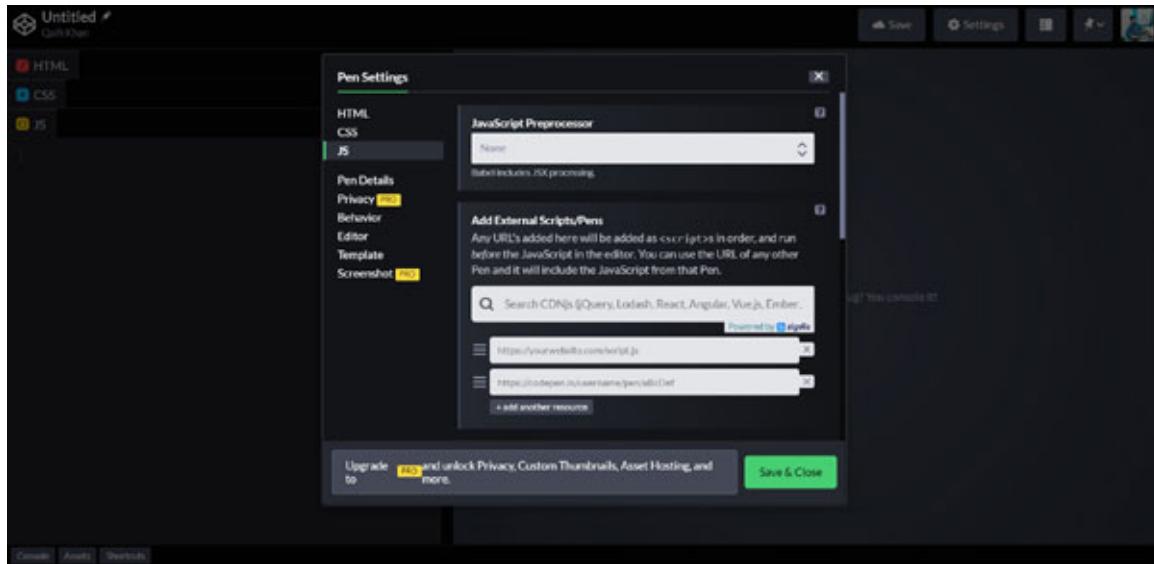


Figure 1.9: Pen settings on Codepen

3. Under JavaScript Preprocessor, there is a dropdown where you can select a preprocessor for your JS code. In that dropdown, you can select **Babel**. Click on “**Save & Close**”, as shown here:

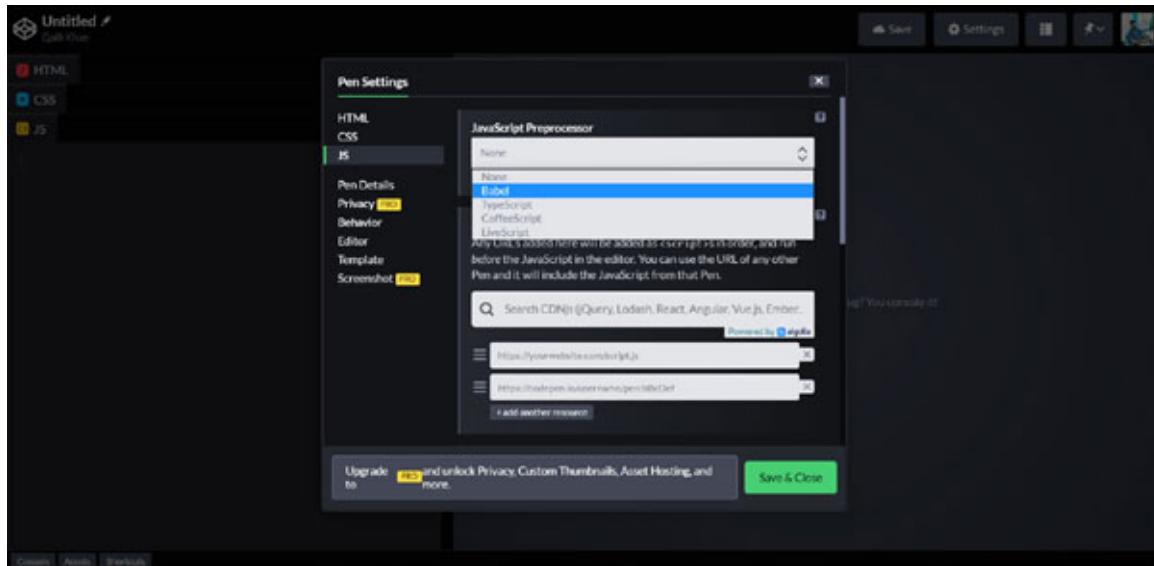


Figure 1.10: JavaScript setting on Codepen

4. Now, if you notice right next to JS, Babel is written in the brackets, which shows that Codepen is using Babel to compile your JavaScript code:

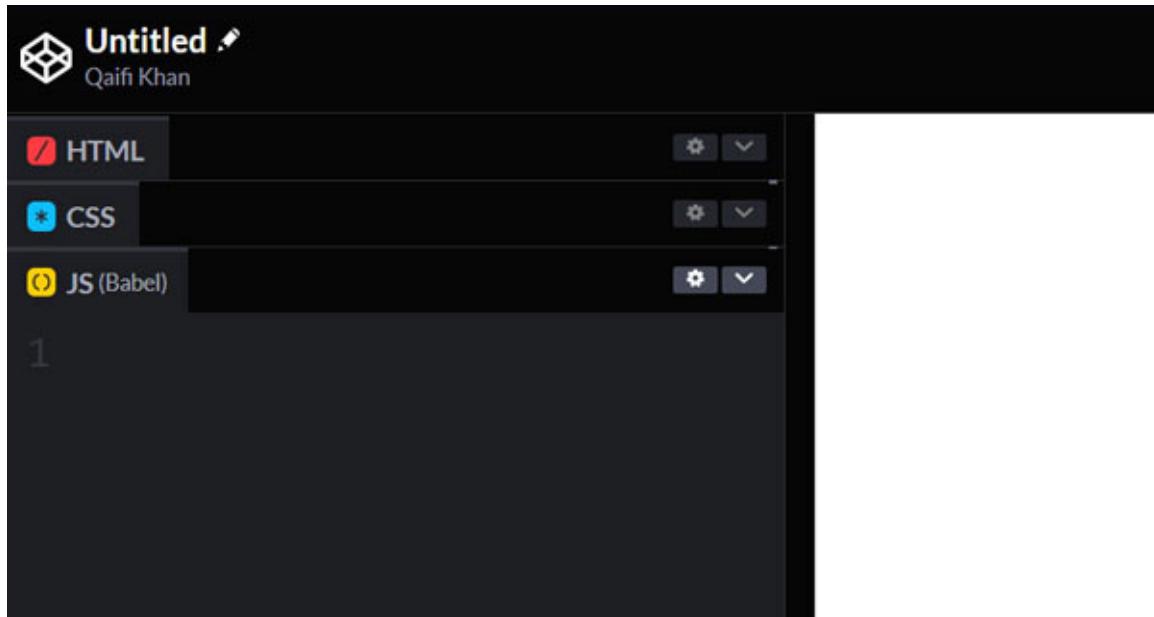


Figure 1.11: Verify preprocessor as Babel

Now, one thing to remember, sometimes Codepen's console misses errors. It would not show you any error even though the code throws an error. These errors are logged in your browser's console, so it is recommended to use the browser's console to check errors at least until we learn a way to use Babel in ReactJS applications.

In the next section, we will learn about all these features, along with some other features that we will use very frequently while building ReactJS applications.

Creating variables using let and const

ES6 introduced two new ways to create variables. The first one is “**let**”; this is kind of like a replacement for **var**. To create a variable, you can simply do something like this.

```
let mName = "John";  
let mSum = 100;
```

The second way is **const**. It is used to create constants, which means that the variables created using **const** cannot be updated later in the code. For example,

```
const API_URL = "https://api.test.com/v1/"
```

You can use **let** and **const** to create variables that hold all the data types provided by JavaScript. Let us give them a try:

```
let num1 = 10;
```

```
let num2 = "Hello Modern JavaScript"
let isOldEnough = false;
let marks = [9, 8, 9.5, 8.5, 10]
let blogData = {
  title: 'Title 1',
  description: 'Lorem ipsum dolor'
}
```

Similarly, let us try creating variables using constants.

```
const num1 = 10;
num1 = 20;
console.log(num1);
```

The preceding code will give us an error (as shown here) because we are trying to update the value of a constant:

A screenshot of a browser's developer tools console. On the left, there are tabs for HTML, CSS, and JS (Editor). The JS (Editor) tab is selected, showing the following code:

```
1 console.clear();
2
3 const num1 = 10;
4 num1 = 20;
5 console.log(num1);
```

On the right, the Console tab shows the output:

```
Uncaught TypeError: Assignment to constant variable.
at https://cdpn.io/cp/internal/boomboom/pen.js?key=pen.js-f44e23a0-e2a1-e56c-d8c1-887f5726a1a3:4
```

Figure 1.12: Updation error on const

So, what is the benefit of creating variables using `let` and `const`?

There are two benefits. First, the variables created using `let` and `const` are block-scoped. Block scoped means that the variables are scoped inside a code block. For example, a variable created inside an `if-else` will be scoped inside that block, as shown in the following figure:

The screenshot shows a browser's developer tools with the JS (Eval) tab selected. The code is as follows:

```
let num1 = 100;
let num2 = 200;
const total = num1 + num2;

if(total > 200) {
  const result = total * 0.9;
  console.log("Result inside if condition => ", result);
}

console.log("Result outside if condition => ", result);
```

The console output is:

```
"Result inside if condition => " 270
Uncaught ReferenceError: result is not defined
at https://cdpn.io/cp/internal/boomboom/pen.js?key=pen.js-d6132d6b-3592-3831-194a-7e6f99e60016:12
```

Figure 1.13: Block-scoped variables

As you can see in the preceding console, we could access and print the value of the variable “**result**” inside the if-condition. Outside the if-condition, it throws an error because it cannot access the variable “**result**”.

Second, they do not support hoisting. This saves us from a lot of unexpected issues, as shown here:

The screenshot shows a browser's developer tools with the JS (Eval) tab selected. The code is as follows:

```
num1 = 100;
console.log(num1);
let num1;
console.log(num1);
```

The console output is:

```
Uncaught ReferenceError: Cannot access 'num1' before initialization
at https://cdpn.io/cp/internal/boomboom/pen.js?key=pen.js-1c39a4bb-2d32-d94c-42bf-2324ac0b11a0:3
```

Figure 1.14: Hoisting error using let and const

Hope this gave you a better understanding of variable creation using **let** and **const**.

Templates strings

It is a string that allows embedding expressions inside it. To create a template string, you do not use single or double quotes; you use a backtick, and the JavaScript expressions are wrapped inside \${}. For example,

```
let name = "John";
const mGreetings = `Hello ${name}`; //Here name is a variable
console.log(mGreetings); //This will print "Hello John"
```

We can even call functions inside this template string. We can also write more complex expressions. Say we wanted to create this string **rgba(randomNum(), randomNum(), and randomNum())**. It is very cumbersome to create this using concatenation, but with Template Strings, it is a breeze.

Example without template strings:

```
let rgbStr = "rgb(" + Math.floor(Math.random() * 255) + ", " +
Math.floor(Math.random() * 255) + ", " + Math.floor(Math.random()
* 255) + ")";
console.log(rgbStr);
```

Example with template strings:

```
let rgbStr = `rgb(${Math.floor(Math.random() * 255)},
${Math.floor(Math.random() * 255)}, ${Math.floor(Math.random() *
255)})`;
console.log(rgbStr);
```

Open the following link to try a working example:

<https://codepen.io/qaifikhān/pen/XWgEoay?editors=0012>

Arrow functions

ES6 gives us a new syntax for defining functions using a fat arrow (=>). Arrow functions bring a lot of clarity and code reduction.

Using function keyword:

```
function greetings(name) {
  return(`Welcome ${name}`);
}
```

Using Arrow functions:

```
const greetings = (name) => {
  return(`Welcome ${name}`)
}
```

As you can see, we do not need the function keyword to define functions anymore. This is one way of writing arrow functions. Let us try other

variations as well.

Function without arguments:

```
const getLikeCount = () => {  
    //Function Body  
}
```

Function with multiple arguments:

```
const getSum = (num1, num2, num3) => {  
    return (num1 + num2 + num3)  
}
```

Function with a single argument:

```
const greetings = name => {  
    return(`Welcome ${name}`)  
}
```

Similarly, if your function just has one line of code, then you can even skip the curly brackets.

```
const greetings = name => `Welcome ${name}`
```

Rest property

So far, we do not know a way to pass a dynamic number of arguments to a function. We always have to specify the number of arguments required by a function. If we pass less, then they are set as undefined. If we pass more arguments, then those extra arguments are simply ignored.

ES9 solved this problem by introducing the Rest property. It is represented by three dots.

Syntax:

```
const mFunc = (...args) => {  
    //Function body  
}
```

To access different arguments, you can treat **args** as an array. The first argument will be accessed as **args[0]**, the second will be accessed as **args[1]**, the third will be accessed as **args[2]**, and so on.

For example,

The following function will return the sum of all the numbers you pass to it:

The screenshot shows a code editor interface with three tabs: HTML, CSS, and JS (Index). The JS tab contains the following code:

```
const getSum = (...args) => {
  let sum = 0;
  for (let i = 0; i < args.length; i++) {
    sum += args[i];
  }
  return sum;
};

console.log(`Sum of 1, 2, 3, 4, 5 is ${getSum(1, 2, 3, 4, 5)}`);
console.log(`Sum of 23, 13, 112 is ${getSum(23, 13, 112)}`);
```

To the right of the editor is a 'Console' window displaying two lines of output:

```
"Sum of 1, 2, 3, 4, 5 is 15"
"Sum of 23, 13, 112 is 148"
```

Figure 1.15: Dynamic arguments using the Rest

Now, just one thing to remember, this rest operator should be the last argument if your function has some named arguments as well. For example,

```
const getDetails = (firstName, lastName, age, ...marks) => {
  //Function Body
}
```

If you add the Rest operator at a position other than the last, then you will get unexpected results from your code.

Open the following link to try a working example:

<https://codepen.io/qaifikhan/pen/gORJrzX?editors=0012>

Spread property

The spread property allows arrays to be expanded into elements and objects into key-value pairs. This helps to create shallow copies of arrays and objects. It is also represented by “three dots”.

Let us try an example to understand it better. Say you have to make a copy of an array, as shown in the following figure:

The screenshot shows a browser's developer tools with the JS (Read) tab selected. On the left, the code is as follows:

```
1 console.clear();
2
3 let mArr = [1, 2, 3]
4 let mArrCopy = mArr
5
6 mArrCopy.push(4);
7
8 console.log(`Original Array ${mArr}`);
9 console.log(`Copied Array
${mArrCopy}`);
```

On the right, the Console tab shows the output:

```
"Original Array 1,2,3"
"Copied Array 1,2,3,4"
```

Figure 1.16: Array copy without Spread

The preceding method does not really copy the values from `mArr` to `mArrCopy`. It only copies the reference of `mArr`, which means if you make any changes in `mArrCopy`, then it will also make the same change in `mArr` because they are basically the same array.

Now, let us use the same example using the spread property. When you do `let mArrCopy = [...mArr]` it basically copies all the elements from the `mArr` and stores them in the new array you created for `mArrCopy`. Notice it is not copying the reference of `mArr` to `mArrCopy` instead, it is copying all the values. If you make any changes in `mArrCopy` it will not be reflected in `mArr` because both are different arrays with the same elements.

The screenshot shows a browser's developer tools with the JS (Read) tab selected. On the left, the code is as follows:

```
1 console.clear();
2
3 let mArr = [1, 2, 3]
4 let mArrCopy = [...mArr]
5
6 mArrCopy.push(4);
7
8 console.log(`Original Array ${mArr}`);
9 console.log(`Copied Array
${mArrCopy}`);
```

On the right, the Console tab shows the output:

```
"Original Array 1,2,3"
"Copied Array 1,2,3,4"
```

Figure 1.17: Array copy with spread

Open the following link to try a working example:

<https://codepen.io/qaifikhan/pen/RwgmpeP>

We can also use the spread property to concatenate multiple arrays. For example,

```
const mArr1 = [1, 2, 3];
const mArr2 = [4, 5, 6];
const concatenatedArr = [...mArr1, ...mArr2];
```

Spread property can also be used to add new elements at the start or end. For example, you want to add 1 at the start and 5 at the end of the following array:

```
let mArr1 = [2, 3, 4];
mArr1 = [1, ...mArr1, 5]
console.log(mArr1);
```

You can perform all these actions on objects as well. For example, to create a copy of an object, you can simply try the following code:

```
let mObj = {
  fName: "John",
  lName: "Lark"
}
let mObjCopy = {...mObj}
```

Destructuring

Destructuring allows us to “*unpack*” arrays or objects into a bunch of variables. It makes working with arrays and objects a bit more convenient.

Syntax:

```
let [a, b] = mArr;
let {name, age} = mObj;
```

Here, **a** and **b** are variables that hold the first and second values of the array **mArr**. The **name** and **age** are variables holding values from the keys **name** and **age** from the object **mObj**.

Let us give it a try to understand it better:

The screenshot shows a code editor with two sections. On the left, under the 'JS (Babel)' tab, there is a block of JavaScript code. The code demonstrates two ways to extract elements from an array. The first way, labeled 'Without Destructuring', uses traditional assignment and indexing: it splits the name into an array, then assigns the first element to fName and the second to sName, and logs the result. The second way, labeled 'With Destructuring', uses the [variable, variable] syntax to directly assign the first and second elements of the array to fName and surname respectively, and logs the result. Both methods produce the same output. On the right, the 'Console' tab shows the output of the code: three lines of text: "Without Destructuring -> John Maneul", "With Destructuring -> John Maneul", and "With Destructuring -> John Lark".

```
// Without Destructuring
const name = "John Maneul Lark"
let arr = name.split(' ')
let fName = arr[0]
let sName = arr[1]
console.log(`Without Destructuring -> ${fName}
${sName}`)

// With Destructuring
let [firstName, surname] = arr;
console.log(`With Destructuring -> ${firstName}
${surname}`)
let [first, , last] = arr;
console.log(`With Destructuring -> ${first} ${last}`)
```

Figure 1.18: Destructuring arrays

The name of the variables can be anything when working with arrays. In Line 11, we wrote `[firstName, surname]`. This `firstName` and `surname` are just variables. `firstName` is the first variable, so it will select the first item from `arr`, and `surname` is the second variable, so it will select the second element from `arr`. When we are working with arrays, the sequence matters. Say you want to select the first and third elements from the array, then we simply leave an empty hole in the destructuring array, as you can see in Line 13.

Instead of two variables, the values can also be stored in object properties:

```
let mObj = {
  firstName: '',
  lastName: ''
}
[mObj.firstName, mObj.lastName] = arr
```

Open the following link to try a working example:

<https://codepen.io/qaifikhan/pen/mdwYmmW?editors=0012>

Now, the same destructuring concept can be used for objects. Just one thing to remember, the variable names need to be the same as that of key names, as shown in the following figure. It automatically maps the variables with the keys. This is why the sequence does not matter for objects.

```
HTML
CSS
JS (Label)
1 console.clear();
2
3 const mObj = {
4   fName: 'John',
5   lName: 'Lark',
6   age: 28
7 }
8 let {fName, age} = mObj;
9 console.log(`Name: ${fName} and Age: ${age}`)
```

Console
"Name: John and Age: 28"

Figure 1.19: Destructuring objects

Even if you write the code in Line 8 as shown as follows, then also it will generate the same result because it matches the key names from the object.

```
let {age, fName} = mObj;
```

Similarly, you can also create new objects using destructuring. For example,

```
let name="John";
let age = 28;
const mObj = {name: name, age: age}
```

Instead of writing the preceding code, you can write the following code. Here, the object keys will be the same as variable names.

```
const mObj = {name, age}
```

I hope this gave you clarity on how you can use destructuring to easily access values from objects and arrays.

New array function: map()

Say you have an array of numbers and want to write code to double the values of array elements. You would do something like the following code:

```
const mArr = [1, 2, 3, 4, 5]
const result = [];
for(let i=0; i<mArr.length; i++) {
  result.push(mArr[i] * 2);
}
```

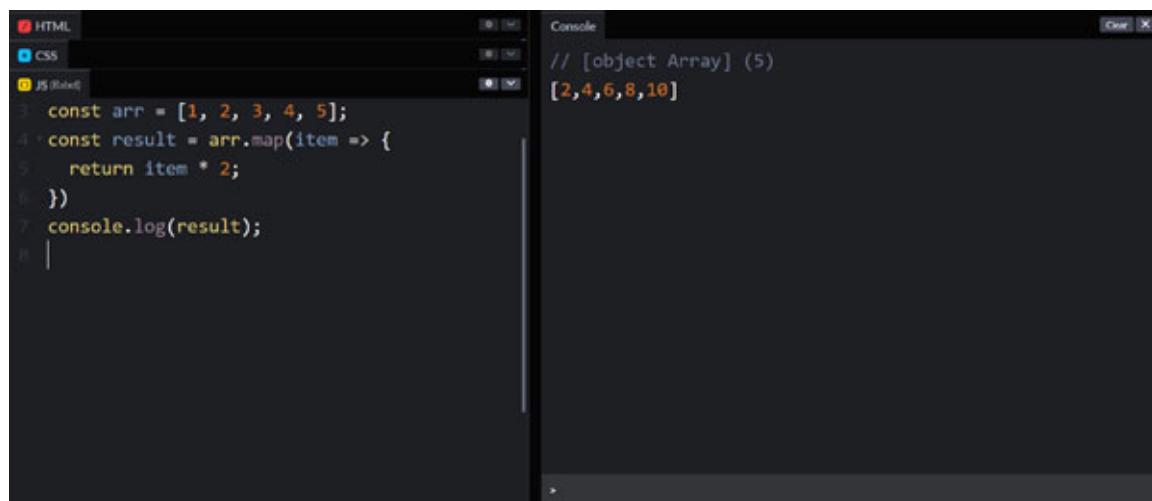
If you analyze the preceding code, you are doing two things—iterating through all the array elements and performing the same operation on all the elements.

Something similar can be achieved using the `map()` function. When we use the `map` function on an array, it iterates the array. We can pass a callback function to perform the same operation on each array item, and it returns a new array for the result.

Syntax of `map()`:

```
const result = arr.map(item => {
  //Some code to execute for each iteration
  return val;
})
```

If you look at the preceding code, we are using the `map()` method on the array stored in variable `arr`. We have passed a callback function as an argument to the `map` method. The callback method receives the current item for each iteration. The function body is the operation you perform for each iteration. You have to return a value at the end of each iteration which will be stored in the result array. Let us try the previous example of doubling array values using `map()`.

A screenshot of a browser's developer tools showing the JavaScript console. On the left, the code is written:

```
const arr = [1, 2, 3, 4, 5];
const result = arr.map(item => {
  return item * 2;
})
console.log(result);
```

On the right, the output is displayed in the 'Console' tab:

```
// [object Array] (5)
[2,4,6,8,10]
```

The 'JS (Eval)' tab is selected in the top-left panel.

Figure 1.20: Double array values using `map()`

Let us try another example to understand this better. Say we have an array of objects, each object holds details about pilots such as full name, age, experience, and so on, and we wanted to create a new array with only the full names of these pilots.

The screenshot shows a browser's developer tools with the 'Console' tab selected. On the left, there is a code editor window with the following JavaScript code:

```
const pilotData = [
  {name: "John Doe", age: 32, experience: "10 years"},
  {name: "James May", age: 32, experience: "3 years"},
  {name: "Clara Cook", age: 32, experience: "12 years"},
  {name: "Ryan Jay", age: 32, experience: "9 years"},
]

const pilotNames = pilotData.map((item, pos) =>
  item.name);
console.log(pilotNames);
```

On the right, the 'Console' output window displays the result of the `console.log` statement:

```
// [object Array] (4)
["John Doe", "James May", "Clara
Cook", "Ryan Jay"]
```

Figure 1.21: Get pilot names using the map()

The map function also gives us access to the position of each item in the array. It is passed as a second argument to the callback function, as you can see in [figure 1.22](#).

New array function: reduce()

Just like `map()`, the `reduce()` method also iterates through the entire array and accepts a callback function to perform some action on the array elements. The difference here is that `reduce` passes the result of this callback from one iteration to the next one. This result of the callback is called an **accumulator**. The accumulator can be pretty much anything—an integer, string, object, or even an array. This accumulator must be instantiated and passed when calling `reduce()`.

The following is the syntax of `reduce()`:

```
arr.reduce((accumulator, item) => {
  //Function Body
}, accumulatorDefaultValue)
```

Let us try to understand the behavior of the accumulator:

The screenshot shows a browser's developer tools interface with two panels: 'JS (state)' on the left and 'Console' on the right. In the 'JS (state)' panel, there is some JavaScript code. In the 'Console' panel, the output is as follows:

```
0
undefined
undefined
undefined
undefined
```

Figure 1.22: Accumulator value without return

The first time it is 0, and for the remaining times, it is undefined. This is because we are not returning any value in the first iteration, which is why for the next iteration, the accumulator gets a value of undefined. So, if we return the `acc`, it should be passed to the next iteration.

The screenshot shows a browser's developer tools interface with two panels: 'JS (state)' on the left and 'Console' on the right. In the 'JS (state)' panel, there is some JavaScript code. In the 'Console' panel, the output is as follows:

```
0
0
0
0
0
```

Figure 1.23: Accumulator value with return

Now, as you can see, we are getting four 0s printed in the console because it is being returned in every iteration, which is passed to the next iteration.

Let us try incrementing this accumulator, as shown here. Now, `acc`'s value will be incremented by 1 and will be passed to the next iteration.

A screenshot of a code editor interface. On the left, there are tabs for HTML, CSS, and JS (Babel). The JS (Babel) tab is active and contains the following code:

```
const mArr = [1, 2, 3, 4, 5];
const result = mArr.reduce((acc, item, pos) => {
  console.log(acc);
  acc++;
  return acc;
}, 0)
```

On the right, there is a 'Console' tab showing the output of the code. It displays the numbers 0, 1, 2, 3, and 4, each on a new line, representing the state of the accumulator at each iteration.

Figure 1.24: Accumulator value with an increment

This is all about the concept of **reduce()** and **accumulator**.

Let us try another example to understand it better. Say we wanted to calculate the total experience of all the pilots, as shown here:

A screenshot of a code editor interface. On the left, there are tabs for HTML, CSS, and JS (Babel). The JS (Babel) tab is active and contains the following code:

```
const pilotData = [
  {name: "John Doe", age: 32, experience: "10 years"},
  {name: "James May", age: 32, experience: "3 years"},
  {name: "Clara Cook", age: 32, experience: "12 years"},
  {name: "Ryan Jay", age: 32, experience: "9 years"},
]
const totalExp = pilotData.reduce((accumulator, item, pos) => {
  const currentExp = parseInt(item.experience.split(" ")[0], 10);
  const cumulativeExp = accumulator + currentExp;
  console.log(`Cumulative Experience after ${pos + 1} iterations => ${cumulativeExp}`);
  return cumulativeExp;
}, 0);
console.log(totalExp);
```

On the right, there is a 'Console' tab showing the output of the code. It displays the following text:

```
"Cumulative Experience after 1 iterations => 10"
"Cumulative Experience after 2 iterations => 13"
"Cumulative Experience after 3 iterations => 25"
"Cumulative Experience after 4 iterations => 34"
34
```

Figure 1.25: Total experience using reduce() method

We do have access to the pilot object inside this callback function; to access the experience, we can simply do **item.experience**. We split based on space to get the numerical value of experience that needs to be converted to a number. Once we have the experience for one iteration, we can add it to the accumulator and return it. I hope this gave you clarity on how **reduce()** and **accumulator** work.

Open the following link to try a working example:

<https://codepen.io/qaifikhān/pen/YzQmEMY?editors=0012>

New array function: filter()

As the name suggests, the **filter()** method is used to filter elements from an array. You can decide which elements should be added to the new array based on some conditions, and the result is stored in a new array.

Syntax for **filter()** function:

```
arr.filter(item => {
    //Return true/false to add/skip element
})
```

Let us try it with the following example to understand it better. Say we need all the pilots with 10 or more years of experience.

The screenshot shows a browser's developer tools console window. On the left, the code is written in JavaScript:

```
const pilotData = [
  {name: "John Doe", age: 32, experience: "10 years"},
  {name: "James May", age: 32, experience: "3 years"},
  {name: "Clara Cook", age: 32, experience: "12 years"},
  {name: "Ryan Jay", age: 32, experience: "9 years"},
]

const result = pilotData.filter((item, pos) => {
  const currentExp = parseInt(item.experience.split(" ")[0], 10);
  return currentExp >= 10;
})
console.log('Pilots with 10 or more years of experience',
result);
```

On the right, the console output is displayed:

```
"Pilots with 10 or more years
of experience" // [object
Array] (2)
[// [object Object]
{
  "name": "John Doe",
  "age": 32,
  "experience": "10 years"
}, // [object Object]
{
  "name": "Clara Cook",
  "age": 32,
  "experience": "12 years"
}]
```

Figure 1.26: filter() method example

New array functions: find() and findIndex()

ES6 introduced two new methods to search for elements inside an array. The **find()** method is used to search for an element in the array that matches a condition. It returns the first occurrence of the matched element.

The **findIndex()** method is quite similar to the **find()** method. The difference is that the **findIndex()** method returns the index of the element instead of the element itself.

Let us try an example to understand these better. Say we have an array and want to search if there is an even number. If an even number exists, then also print that number.

A screenshot of the JS Bin web application. On the left, the code editor shows a snippet of JavaScript:console.clear();
const numArr = [-1, -2, -3, -4, 1, 2, 3, 4];
const result = numArr.find((item) => {
 console.log(item);
})
console.log(result)On the right, the 'Console' tab displays the output of the code:-1
-2
-3
-4
1
2
3
4
undefined

Figure 1.27: *find()* method iterates the array

In Line 7, it is printing undefined because no value is returned from the find function. So, if we simply return true. Then we will get the first element, as you can see in [figure 1.28](#). Also, notice as soon as we return true, it stops iterating.

A screenshot of the JS Bin web application. On the left, the code editor shows a modified version of the previous code:console.clear();
const numArr = [-1, -2, -3, -4, 1, 2, 3, 4];
const result = numArr.find((item) => {
 console.log(`Inside loop \${item}`);
 return true;
})
console.log(`Found the following item: \${result}`);On the right, the 'Console' tab displays the output:"Inside loop -1"
"Found the following item: -1"

Figure 1.28: *find()* method returns the first element

Now, let us add a condition such that we get the first positive even number.

The screenshot shows a browser's developer tools with the JS (Reactive) tab selected. On the left, the code is displayed:

```
1 console.clear();
2
3 const numArr = [-1, -2, -3, -4, 1, 2, 3, 4]
4 const result = numArr.find((item) => {
5     console.log(`Inside loop ${item}`);
6     if(item > 0 && item % 2 === 0) {
7         return true;
8     }
9 })
10 console.log(`Found the following item: ${result}`);
```

On the right, the Console tab shows the output:

```
"Inside loop -1"
"Inside loop -2"
"Inside loop -3"
"Inside loop -4"
"Inside loop 1"
"Inside loop 2"
"Found the following item: 2"
```

Figure 1.29: find() method returns the first positive even number

Similarly, if you want the position of the selected element, then you can use the `findIndex()` method. It will return `-1` if it cannot find any matching element.

Classes, properties, and methods

We are not going to learn object-oriented programming here. I hope you know that JavaScript follows prototypal inheritance. It uses prototype methods to create object factories and implement object-oriented programming.

In ES6, classes were introduced to create object factories. Let us see how it works.

Syntax:

```
class ClassName {
    constructor() {
        // Initialize the class properties here
    }
    //Methods are added outside constructor
    methodName = () => {}
}
```

Now, remember that this is just syntax sugar. Behind the scenes, everything still works the same. Let us try an example to understand it better.

The screenshot shows a code editor with two tabs: 'HTML' and 'JS (file)'. The 'JS (file)' tab contains the following code:

```
3 class Person {
4     constructor(fName, lName, age) {
5         this.firstName = fName;
6         this.lastName = lName;
7         this.age = age;
8     }
9     getDetails = () => {
10        return `${firstName} ${lastName} with
11        age ${age} years`;
12    }
13 const john = new Person("John", "Doe",
14 "40");
15 console.log(john);
```

The 'Console' tab shows the output of the code execution:

```
// [object Object]
{
  "firstName": "John",
  "lastName": "Doe",
  "age": "40"
}
```

Figure 1.30: Classes, properties, and methods

Inheritance

Inheritance is also possible in ES6. It has provided us with a keyword called `extends` to inherit classes.

Syntax:

```
class ParentClass {
    // Class body
}
class ChildClass extends ParentClass {
    // Class body
}
```

Say you have these two classes—**Person** and **Pilot**. The **Person** class is the parent, and the **Pilot** is the child class. We already have this **Person** class we created in the previous example. Let us try writing some code to create a **Pilot** class and inherit the **Person** class, as shown in the following figure:

```
14 class Pilot extends Person {
15     constructor(fName, lName, age, exp, airlines) {
16         super(fName, lName, age);
17         this.experience = exp;
18         this.airlines = airlines;
19     }
20     printFullDetails = () => {
21         console.log(`${this.getDetails()} flies with ${this.airlines}
and has experience of ${this.experience} years`)
22     }
23 }
24
25 const john = new Pilot("John", "Doe", "40", "20", "AirEarth");
26 john.printFullDetails();
```

Figure 1.31: Inheritance

The parent class will also need the data to initialize its properties. For that, ES6 provides us with a method called **super()**. We pass the value in the super method. The **super()** method needs to be called before you use this keyword, or else JavaScript will give you an error. When you call **super()**, it calls the constructor of the parent class.

This is it. That is all you have to do to inherit the properties and methods of the **Person** class in **Pilot** class. If you print the object of the pilot, it will have all the properties and methods of the **Person** class, which is the parent class in this example.

Conclusion

JavaScript has launched many new features that overcome the shortcomings of the previous versions. These features have drastically improved the development experience. In this chapter, we got comfortable with the Modern JavaScript concepts that we are going to use to code ReactJS applications. In the upcoming chapter, we are going to learn about React, JSX, and component-style application architecture.

Questions

- Q1. What is ES6?
- Q2. What is the difference between var, let, and const?
- Q3. What is map(), filter(), and reduce()?
- Q4. What are classes? How are they different from prototypes?

Q5. How can destructuring help unpack-objects?

Multiple choice questions with answers

Q1. ES6 was released in which year?

- a. 2015
- b. 2016
- c. 2017
- d. 2018

Q2. var is block-scoped?

- a. True
- b. False

Q3. Which of the following statements is true?

- a. Spread is used to add dynamic arguments to a function.
- b. Spread is used to unpack array items only.
- c. Spread is used to unpack object key-value pairs only.
- d. Spread is used to unpack both objects and arrays.

Q4. What is the data type of accumulator in reduce()?

- a. Number
- b. String
- c. Array
- d. All of the above

Q5. Which keyword is used to inherit properties and methods from the parent class?

- a. class
- b. extends
- c. inherit
- d. parent

Answers

Questio n	Correct answer
Q1	a. 2015
Q2	b. False
Q3	d. Spread is used to unpack both objects and arrays.
Q4	d. All of the above
Q5	b. Extends

References

- <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let>
- <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const>
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions
- <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes>

CHAPTER 2

Up and Running with React Ecosystem

In the previous chapter, we learned about modern JavaScript features such as scoped variable creation, array functions, rest and spread operators, and destructuring. In this chapter, we will kick-start our React journey. We will understand what exactly this ReactJS is and how we write basic applications in React.

Structure

In this chapter, we will discuss the following topics:

- Introduction to ReactJS
- Installing node and NPM
- Creating our first React project
- Introduction to components
- Introduction to JSX
- Inline and external styles
- Rendering dynamic elements

Objectives

After studying this chapter, you will understand the fundamentals of ReactJS. You will know how to create a React project using CRA. You will know about JSX, which is used to write components. You will be able to style and create both static and dynamic React components.

Introducing ReactJS

ReactJS is a JavaScript library for building fast and interactive user interfaces. It was developed by Facebook in 2011 to solve code maintenance problems faced by Facebook's internal development team. It was open-sourced in 2013 for developers across the world to use it. Currently, it is the most popular library for building user interfaces.



Figure 2.1: Top 3 frontend framework trends in the last five years

And as you can see, React is way ahead of its major competitors, which are Angular and VueJs.

Component-based design

The entire React application can be thought of as a set of independent, isolated, and reusable components. These components are put together to design complex layouts. In other words, you can say components are the building blocks of a React application. For example, check out Myntra's Web app, as shown in the following screenshot:

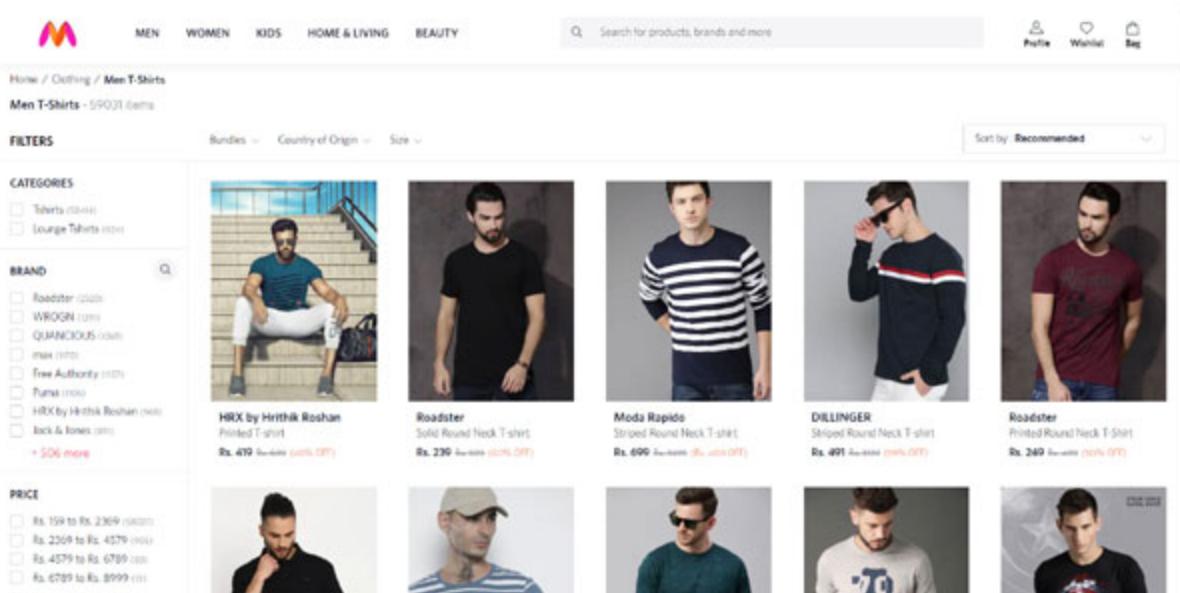


Figure 2.2: Myntra search page

This entire page can be split into independent components, say—Topbar, Products Grid, and Filter section. Refer to [Figure 2.3](#):

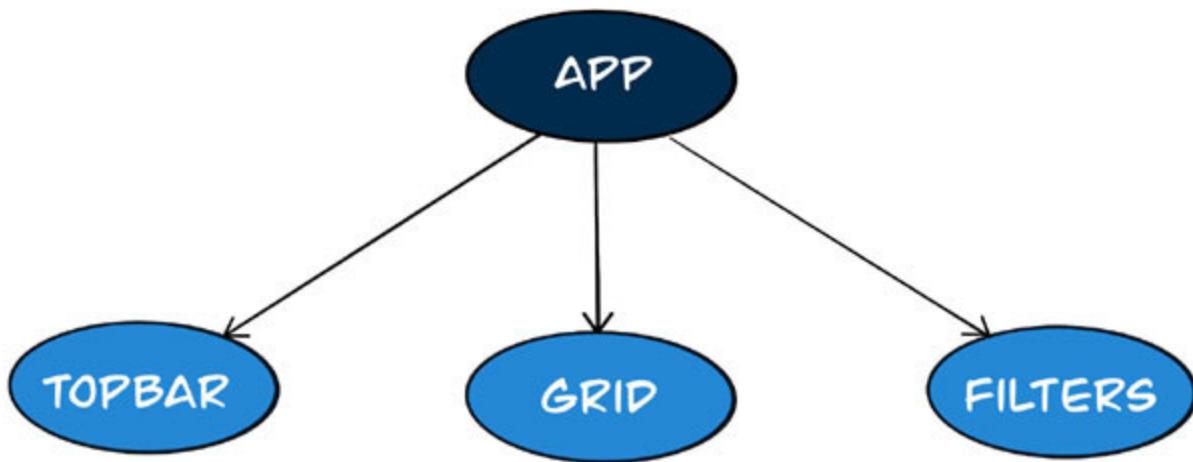


Figure 2.3: Components on Myntra's search page

These were at the top level. Even inside the top bar, you can have more components, say —Logo, Text menu items, SearchBox, and Icon menu items. Similarly, you can have more components even inside the Products Grid, as shown here:

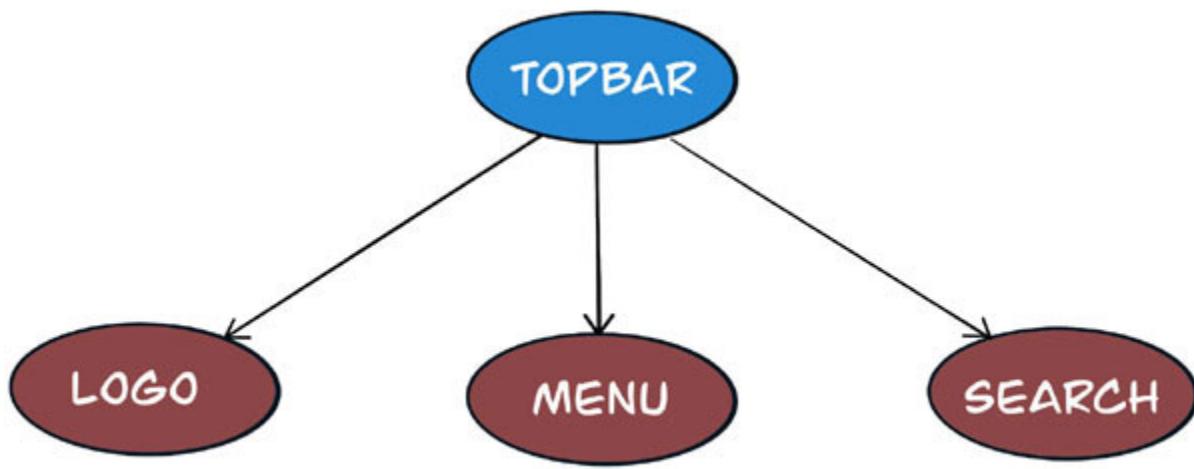


Figure 2.4: Topbar components

Every React application has at least one component, which we refer to as the **App component**. The app component represents our React app, and all the other components are placed inside it. This results in a tree-like component hierarchy, as shown here:

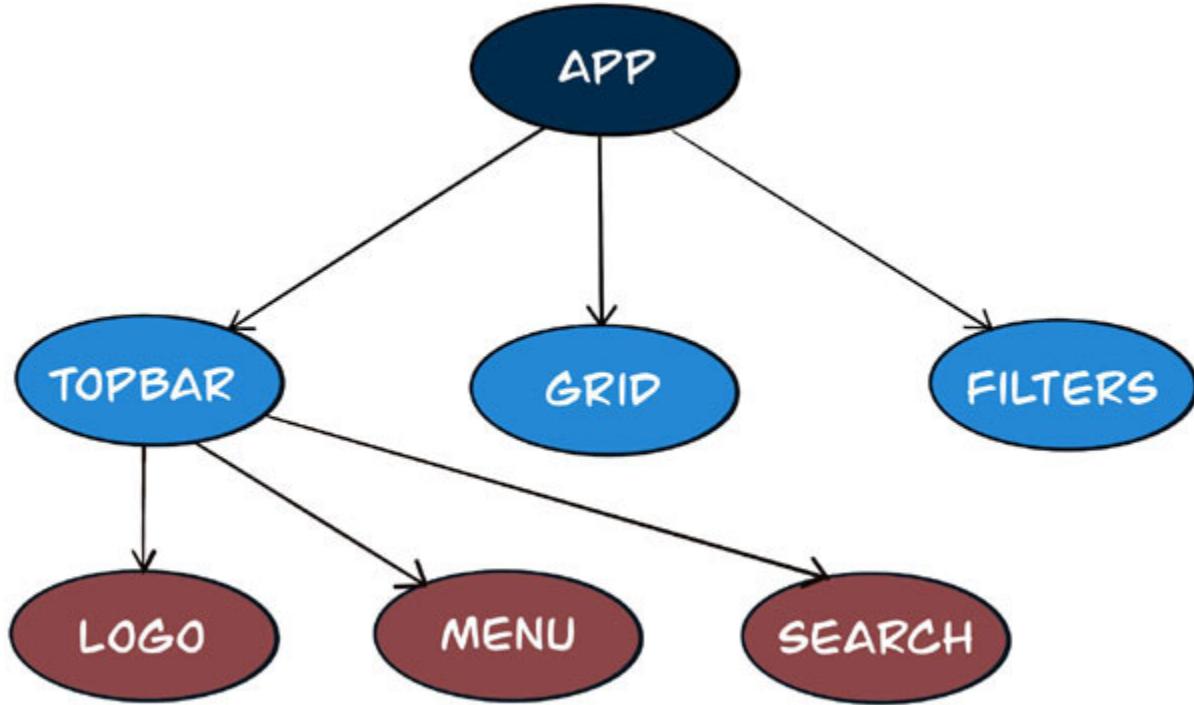


Figure 2.5: Application component tree

As you can see in the preceding diagram, we have the app component on the top of the tree. Under that, we have the topbar, places grid, and filter

components. Under the top bar, we have the logo, menu items, and search box.

If you remember, it was mentioned earlier that React is a library, which means it has a very specific purpose to design the user interface, also known as *Views*. This is where React gets more amazing, and it gives you absolute control of your application. It just takes care of the *view*, and everything else is left for you to decide. You can plan the architecture and even decide upon which libraries to use for different functionalities. If you want to add routing, you can use *react-router*, or if you want to make network requests, you can use libraries such as *Axios* or *Fetch*. You can even use React in parts of your existing Web application because it can be used to design just the *view* without disturbing your existing technology stack.

A lot of famous Web apps are written in ReactJS. For example, Facebook, Prime Videos, Netflix, Instagram, Twitter, and many more.

In the upcoming chapters, we will learn all about ReactJS.

Single-page versus multi-page Web apps

A single-page application is an application that works inside a browser and does not require a page reload during use.

Single-page applications are all about providing an exceptional **user experience (UX)** by providing a smooth page flow—no page reloads and no extra wait time. It is just a single HTML page that you visit, which then loads all other content using JavaScript.

You are already using these types of applications every day. For example, Gmail, Facebook, Instagram, Trello, and so on. If you visit these Web apps and navigate from one page to another, then it will change the browser URL, but it will not refresh the browser. This allows you to see the page loaders, which shows that there is some progress. You can try this right now, open Facebook or Gmail, and verify this.

Multi-page applications work in a “*traditional*” way. Every change on the Web app, for example, showing the list of data or submitting data back to the server, requests the server to render a new page in the browser. So, whenever a user navigates from one page to another, a request is sent to the

server to send a new HTML file for that URL. The server returns a file, and then that HTML file is loaded into the browser.

Some examples of multi-page applications are Dribbble, Harvard's Official Site, and so on. Please visit <https://www.dribbble.com> and click on the top menu items to navigate to different pages. You will notice that the browser will refresh.

React makes it possible to design both single-page applications as well as multi-page applications. In this book, we will learn how to design SPAs using React.

[Creating our first React project](#)

When you build a React app, there are a lot of libraries that are required to make the application function properly. For example, React, React-DOM, Webpack, Babel, and so on. Do not worry, and we will talk about all these things in the upcoming chapters.

For now, just know that we have two options to create a React app—either to install all those libraries and configure the entire project ourselves, or we can use a tool officially provided by the guys who developed React. It is called **create-react-app**. It installs all the libraries and packages required, and it creates a default configuration for our react project. It also adds some starter files in the newly created application.

Let us give it a try. You can simply search **create-react-app** in your browser. Open create-react-app's GitHub page, as shown in the following figure:

Create React App

Azure Pipelines succeeded PRs welcome

Create React apps with no build configuration.

- Creating an App – How to create a new app.
- User Guide – How to develop apps bootstrapped with Create React App.

Create React App works on macOS, Windows, and Linux.
If something doesn't work, please [file an issue](#).
If you have questions or need help, please ask in [GitHub Discussions](#).

Quick Overview

```
npx create-react-app my-app  
cd my-app  
npm start
```

If you've previously installed `create-react-app` globally via `npm install -g create-react-app`, we recommend you uninstall the package using `npm uninstall -g create-react-app` or `yarn global remove create-react-app` to ensure that npx always uses the latest version.

(*npx comes with npm 5.2+ and higher, see instructions for older npm versions*)

Figure 2.6: Create-react-app official documentation

If you scroll down on that page, you can see here all the installation instructions and other details about this tool—how will it work? What happens behind the scenes? And so on. You can check it out if you would like to know more about it. For now, let us learn how to use it to create a new React project. To create a react project using `create-react-app` we can simply run the following command:

Option one:

```
npm -g create-react-app  
create-react-app first-react-app
```

The preceding method is deferred. It is not used anymore. So, if you see this anywhere on the internet, then do not follow it. Instead, follow the instructions as follows:

Option two:

```
npx create-react-app first-react-app
```

If you notice here, there are two words in these commands—**npm** and **npx**. **npm** is the node package manager for JavaScript. It helps you manage all the third-party packages and libraries that you will install for your application. **npm** is installed automatically when you install NodeJS.

npx is a node package runner. It is used to download and run a package temporarily. In this case, we will just be using **create-react-app** once to create our project, and we would not need to run it afterward, so that is why we use **npx**, which helps us to use it without actually installing it.

First, let us install NodeJS. You can go to your browser to search for NodeJS. Open the link which says “*NodeJS Download*”, as shown in the following figure:

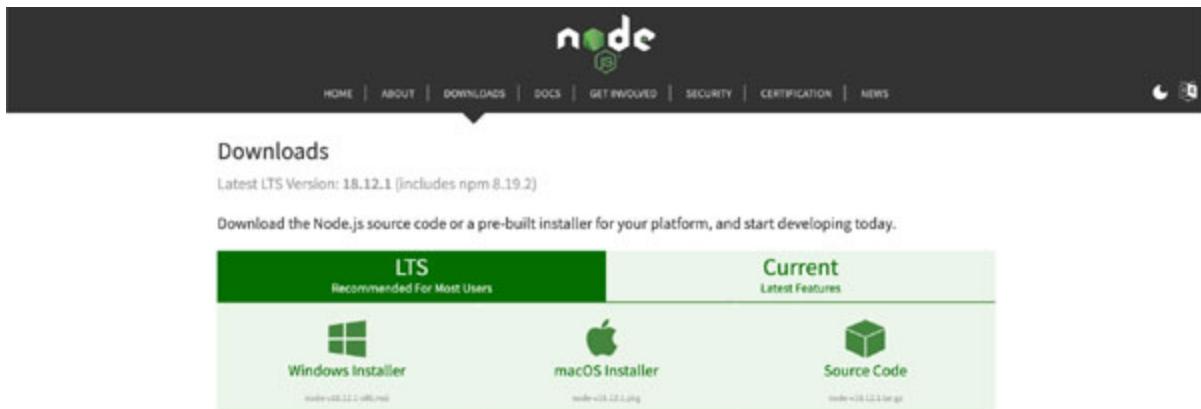


Figure 2.7: NodeJS download page

You need to download the LTS version because a lot of third-party libraries might not support the latest version of the node. You can click on the windows download button to download NodeJS if you are running a Windows operating system. If you are running Mac or Linux, you can click on the respective download button to download the version compatible with your OS. Once it is downloaded, install it just like you would install any other application.

Once it is installed, open your command prompt or terminal, depending on whether you are running Windows, iOS, or Linux.

Run the command **node -v** to confirm whether NodeJS was installed. Also, run another command **npm -v**, to confirm whether npm was installed.

Now, we can run **create-react-app** commands to create our first react application.

```
npx create-react-app first-app
```

We have named the application “*first-app*”; you can name it whatever you want to name it. This name will be used to create a directory/folder for your react project. Run the following command to navigate to your project directory:

```
cd first-app
```

Run the following command to verify if you are inside your project directory.

For Windows:

dir

For Linux or Mac:

1s -1

You should see files/folders with names such as `package.json`, `node-modules`, `src`, and so on, as shown in the following figure:

Figure 2.8: Project directory structure

Now, one final check, to run our application in the local system, we can run the command **npm start**. It starts a local server and loads the default application created by **create-react-app** in the browser.

Let us open this first-app folder in our code editor to see what all files are created by the create-react-app tool.

- **node_modules**: This folder contains all the files related to the packages and libraries required to run your React application. These packages and libraries are installed automatically by the create-react-app tool.
- **public**: This folder contains the following files:
 - **index.html**: This is the main HTML file where all the JavaScript code is loaded to show different pages of the Web app. You can still add the external scripts, font CDNs, and so on here.
 - **manifest.json**: This file is useful when you are building Progressive Web Apps, which are more commonly known as PWAs. PWAs are not part of this book, so we will not talk about them.

You can also add **sitemap.xml**, **robots.txt** files, favicon, and so on in this folder.

- **src**: This folder contains all the files where the code for your application is written. By default, as you can see, we have the following files:
 - **index.js**: This is the main js file that loads the application into the **index.html** file. It is configured automatically by create-react-app to do so. If you change the name of this file, your application will not function.
 - **index.css**: CSS file that contains some default styles which you can remove.
 - **App.js**: This is the main component. Remember the component hierarchy. This app is the same component that holds the rest of the application.
 - **App.css**: CSS file that contains default styles for the app component. You can clear this file too.

- **App.test.js**: This file comes in handy when you want to write test cases for your app component. If you ever decide to write unit test cases, then you will have to create **test.js** files for the components that you want to test.
- Some other files, such as **logo.svg**, **setupTests.js**, **reportWebVitals.js**, and so on. You do not have to worry about these files. In this book, we will work on the files mentioned previously, and we will also create our own files for different projects.
- **package.json**: This file contains the details about all the packages that are directly required to run your application.

Introducing JSX

Check out the following code:

```
<h1>Hello {username}</h1>
<p>Current time: {new Date()}</p>
```

It looks very much like HTML, but it is not HTML. It is called JSX, and it is a syntax extension to JavaScript. It allows us to define React elements using syntax that looks very similar to HTML. It is used to define the look of the UI. Basically, just like you write HTML code to define the structure of your Web page, you use JSX to define the structure of React components.

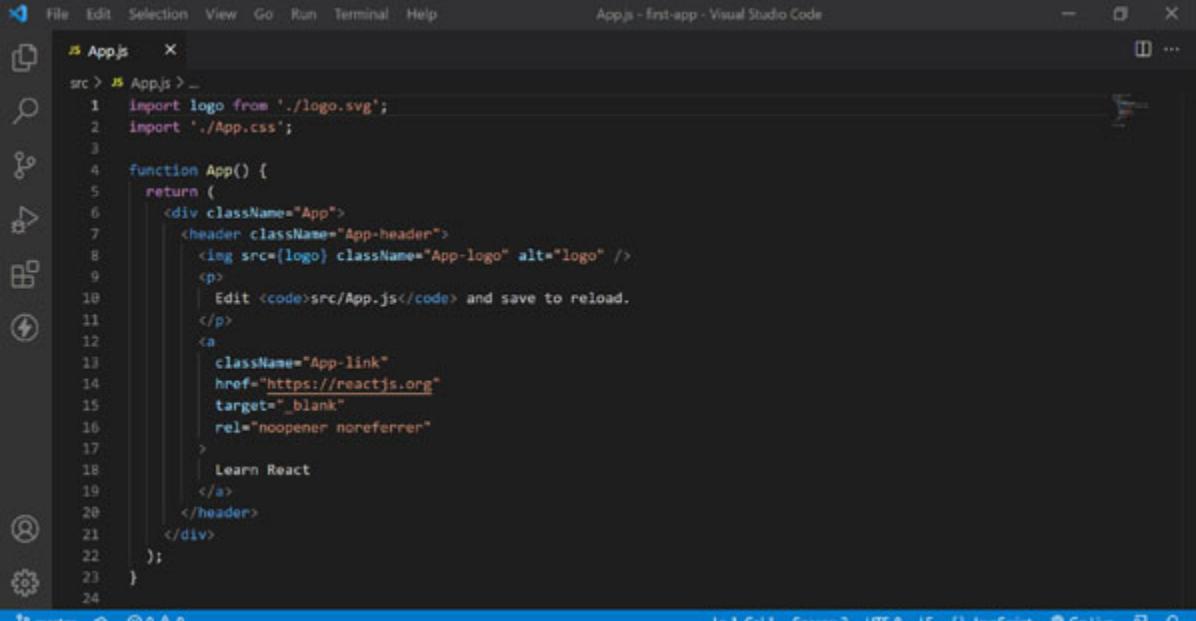
If this is the purpose of using JSX, then why cannot we simply use HTML? Well, it provides us with a lot more than just this. It couples the rendering logic with other UI logic, such as event handling, state changing, data displaying, and so on.

If you look at that code again. We have some curly brackets in the JSX code. Inside these curly brackets, you can write JavaScript expressions. For example:

```
const fullName = "John Doe";
<h1>Hello {fullName}</h1>;
```

The preceding code will print “Hello John Doe”. We have used a variable for rendering the **paragraph text**. Similarly, we can also use functions, operators, and much more.

As you can see in the following image, we have a function `App()` with a return statement inside its body. Whatever you write inside the return statement of this function is shown on the Web page. When you save this file, the local server automatically updates your application in the browser. This automatic update feature of React is called **Hot Reload**. We will talk more about the remaining code of this file in the upcoming chapters. Refer to [Figure 2.9](#):

A screenshot of the Visual Studio Code interface. The title bar says "App.js - first-app - Visual Studio Code". The left sidebar shows a tree view with "App.js" selected. The main editor area contains the following code:

```
App.js  x
src > App.js > ...
1 import logo from './logo.svg';
2 import './App.css';
3
4 function App() {
5   return (
6     <div className="App">
7       <header className="App-header">
8         <img src={logo} className="App-logo" alt="logo" />
9         <p>
10           | Edit <code>src/App.js</code> and save to reload.
11         </p>
12         <a
13           | className="App-link"
14           | href="https://reactjs.org"
15           | target="_blank"
16           | rel="noopener noreferrer"
17         >
18           | Learn React
19         </a>
20       </header>
21     </div>
22   );
23 }
24
```

The status bar at the bottom shows "In 1, Col 1 Spaces: 2 UTF-8 LF (JavaScript) Go Live R D".

Ln 1, Col 1 Spaces: 2 UTF-8 LF (JavaScript) Go Live R D

Figure 2.9: App.js default code

Let us try some examples to understand it better. Say you have to design the user card shown in the following image. Do not worry about the CSS right now, we will talk about it in a bit. You need to add a wrapper for the card, an image for the user avatar, a heading for the channel name, and probably a paragraph for the subscriber count.

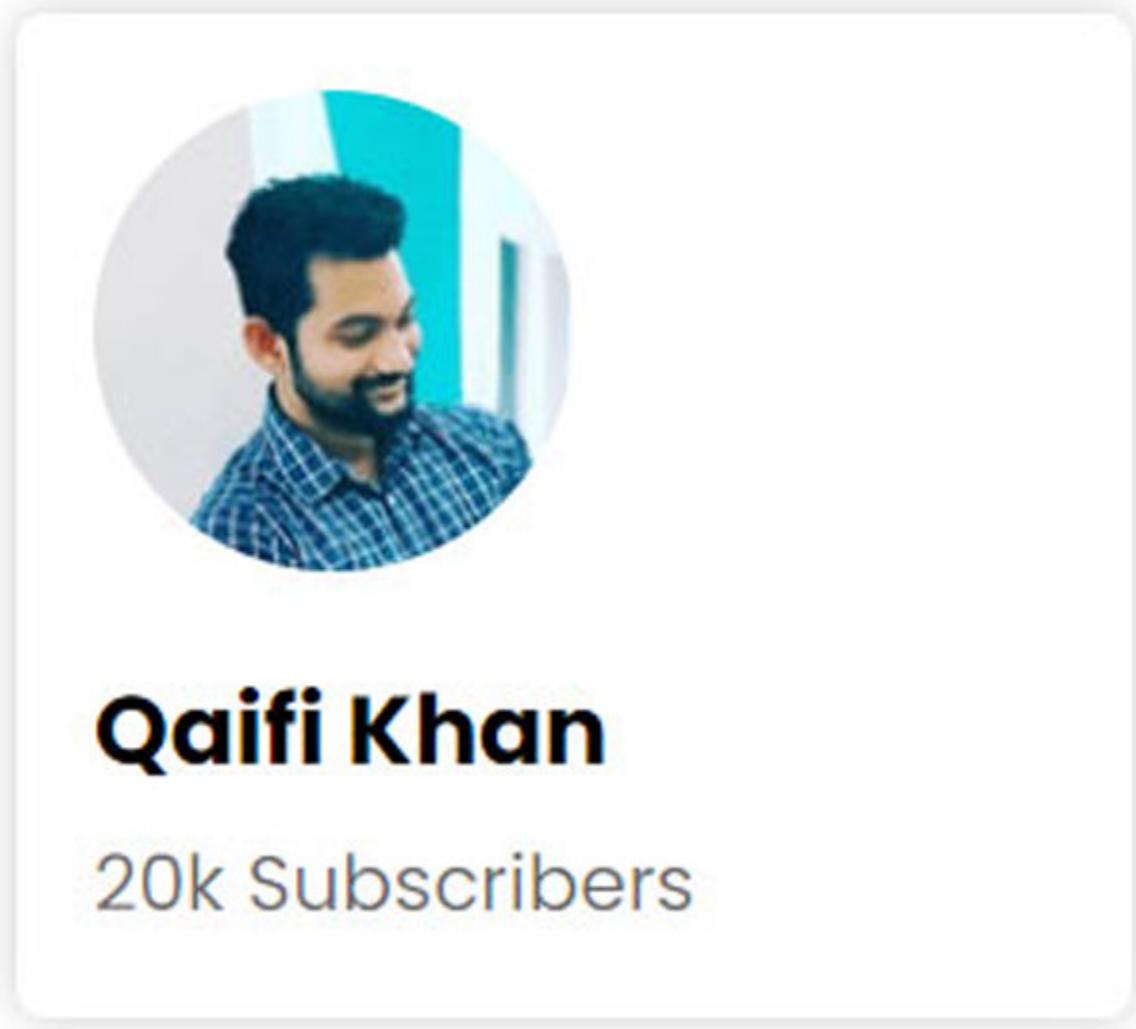
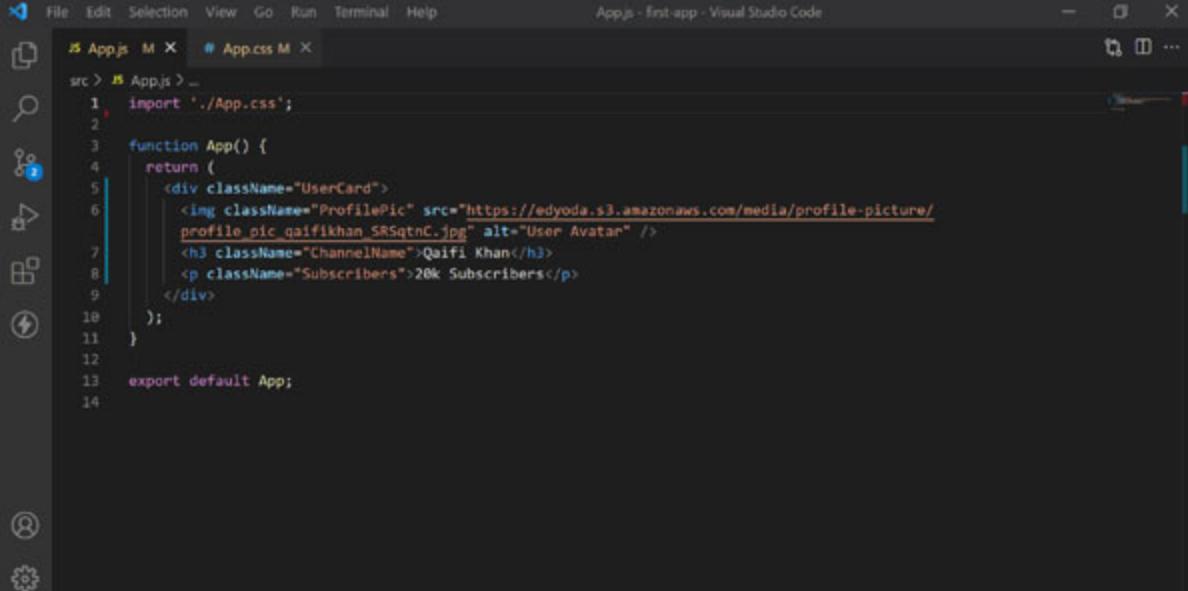


Figure 2.10: User details card

We will write the code inside the `return()` statement, as shown in the following screenshot:



The screenshot shows a dark-themed instance of Visual Studio Code. The left sidebar contains icons for file operations like Open, Save, and Find. The main editor area displays the following code:

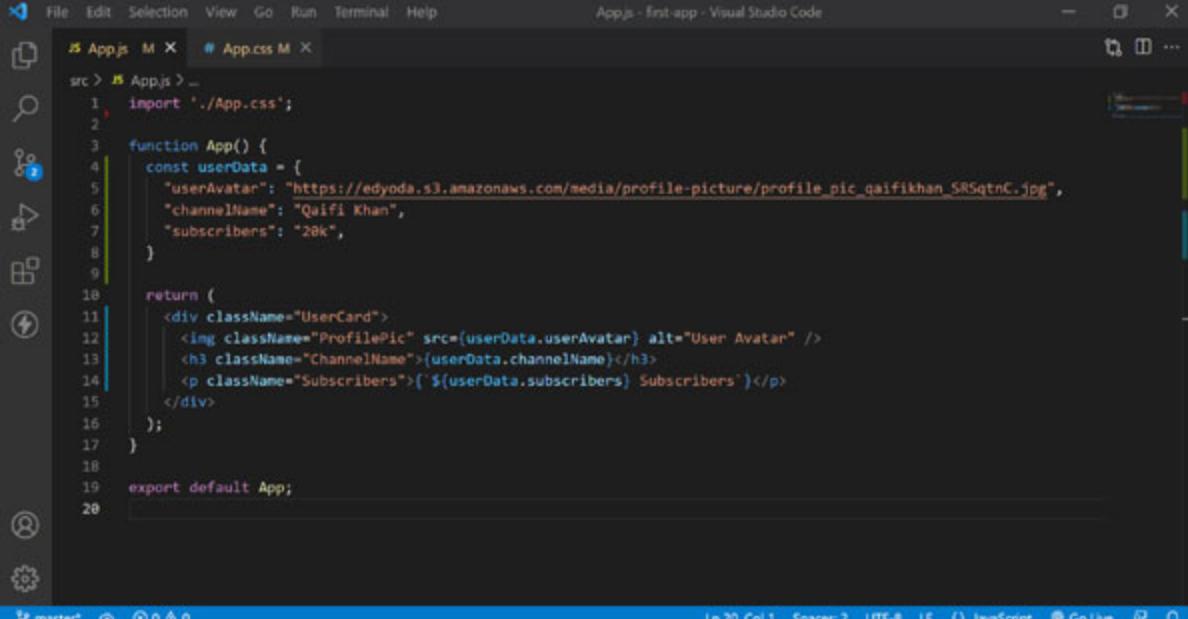
```
App.js M X # App.css M X
src > App.js > ...
1 import './App.css';
2
3 function App() {
4   return (
5     <div className="UserCard">
6       
7       <h3 className="ChannelName">Qaifi Khan</h3>
8       <p className="Subscribers">20k Subscribers</p>
9     </div>
10   );
11 }
12
13 export default App;
14
```

Figure 2.11: User card JSX code

This was all static data. Let us try using some variables for displaying this data. Say this data was available in the following object:

```
const userData = {
  "userAvatar": "https://edyoda.s3.amazonaws.com/media/profile-
picture/profile_pic_qaifikhansRSqtnC.jpg",
  "channelName": "Qaifi Khan",
  "subscribers": "20k",
}
```

To load the user data in UI from the **userData** object, our code will look something like as shown in [Figure 2.12](#):



```
File Edit Selection View Go Run Terminal Help
App.js M X # App.css M ...
src > App.js > ...
1 import './App.css';
2
3 function App() {
4   const userData = {
5     "userAvatar": "https://edyoda.s3.amazonaws.com/media/profile-picture/profile_pic_qaifikhhan_SR5qtnC.jpg",
6     "channelName": "Qaifi Khan",
7     "subscribers": "20k",
8   }
9
10  return (
11    <div className="UserCard">
12      <img className="ProfilePic" src={userData.userAvatar} alt="User Avatar" />
13      <h3 className="ChannelName">{userData.channelName}</h3>
14      <p className="Subscribers">{'${userData.subscribers}' } Subscribers</p>
15    </div>
16  );
17}
18
19 export default App;
20
```

Ln 20, Col 1 Spaces: 2 UTF-8 () JavaScript () Go Live ()

Figure 2.12: User card JSX code using object data

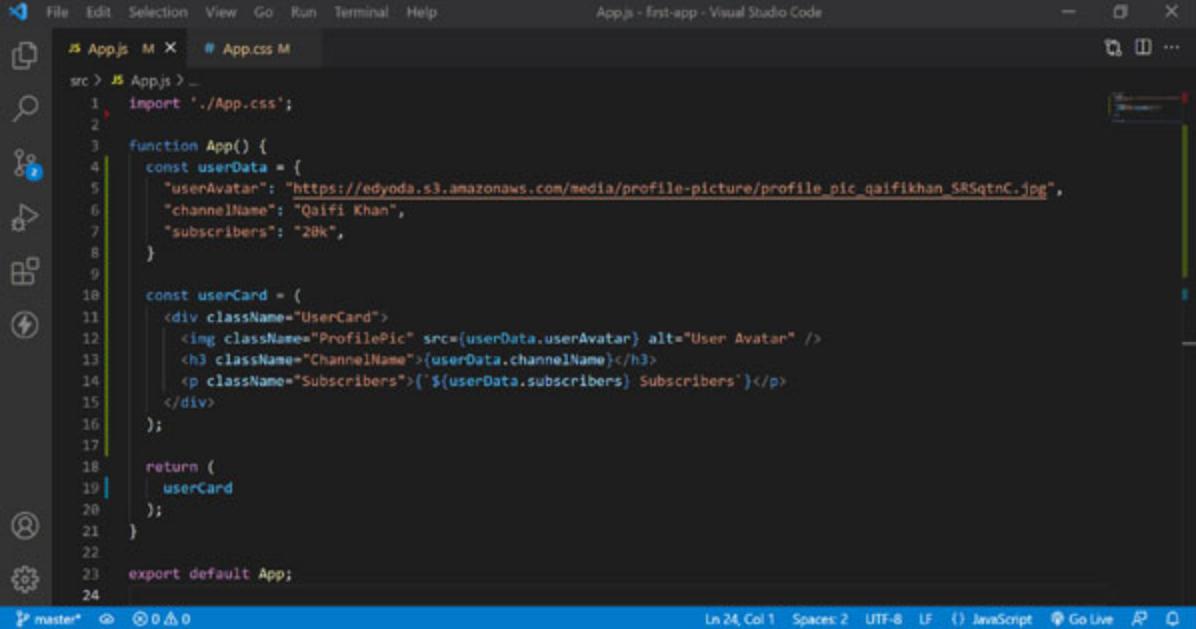
We can also call functions inside the JSX code. Say we want to show the full name in the UI, and we have a function that takes first and last names as arguments and returns the full name. We can write the following code:

```
<h3>{getFullName("John", "Doe")}</h3>
```

We can also add attributes just like normal HTML, but the only difference is that the name is converted to camel case. For example, autocomplete gets converted to **autoComplete**, onclick gets converted to **onClick**, and so on:

```
<button onClick={onLoginBtnClick}>Login</h3>
```

We can even store JSX in variables, as shown in the following screenshot:



The screenshot shows a Visual Studio Code interface with the file 'App.js' open. The code defines a function 'App' that returns a JSX component. Inside the component, there is a 'const userData' object with properties like 'userAvatar', 'channelName', and 'subscribers'. The JSX includes an image tag with 'src' set to the value of 'userData.userAvatar', and a paragraph tag with 'className' set to 'Subscribers' containing the value of 'userData.subscribers'.

```
File Edit Selection View Go Run Terminal Help
App.js - First-app - Visual Studio Code
src > App.js > ...
1 import './App.css';
2
3 function App() {
4   const userData = {
5     "userAvatar": "https://edify.s3.amazonaws.com/media/profile-picture/profile_pic_qaifikhani_SR5qtnC.jpg",
6     "channelName": "Qaifi Khan",
7     "subscribers": "20k",
8   }
9
10  const userCard = (
11    <div className="UserCard">
12      <img className="ProfilePic" src={userData.userAvatar} alt="User Avatar" />
13      <h3 className="ChannelName">{userData.channelName}</h3>
14      <p className="Subscribers">'{userData.subscribers} Subscribers'</p>
15    </div>
16  );
17
18  return (
19    userCard
20  );
21}
22
23 export default App;
24
```

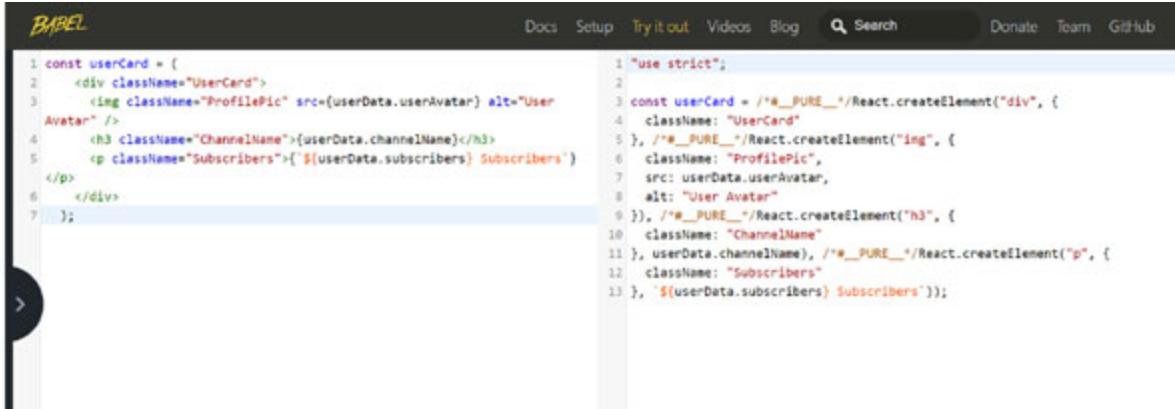
Figure 2.13: JSX stored in a variable

Now, just one important thing to remember, inside these curly brackets, you can only pass the expressions which generate some value by default. You cannot have a for loop inside it, but you can have a map statement because, by default the for loop does not return a value, but the map returns an array. Similarly, you cannot have if-else, but you can use a ternary operator.

What happens when you write JSX? It gets converted to **React.createElement()**. That is why the React module is imported from the React library. The **React.createElement()** method takes three arguments:

- The element name. For example, p, h1, div, and so on.
- An object which holds all the attributes of the element.
- An array of the element's child elements.

As you can see in the following image, on the left, we have the JSX code, and on the right-hand side, we have the converted code:



The screenshot shows the Babel.js interface. On the left, there is a code editor containing JSX code for a user card component. On the right, the converted ES6 code is displayed. The JSX code includes imports for React.createElement and React.PureComponent, and uses class-based components with `render` methods. The converted ES6 code uses the `use strict` directive and the `React.createElement` function to achieve the same functionality.

```

1 const userCard = (
2   <div className="UserCard">
3     <img className="ProfilePic" src={userData.userAvatar} alt="User Avatar" />
4     <h3 className="ChannelName">{userData.channelName}</h3>
5     <p className="Subscribers">{`${userData.subscribers} Subscribers`}</p>
6   </div>
7 )
  
```

```

1 "use strict";
2
3 const userCard = /*#__PURE__*/React.createElement("div", {
4   className: "UserCard"
5 }, /*#__PURE__*/React.createElement("img", {
6   className: "ProfilePic",
7   src: userData.userAvatar,
8   alt: "User Avatar"
9 }), /*#__PURE__*/React.createElement("h3", {
10  className: "ChannelName"
11 }, userData.channelName), /*#__PURE__*/React.createElement("p", {
12  className: "Subscribers"
13 }, `${userData.subscribers} Subscribers`));
  
```

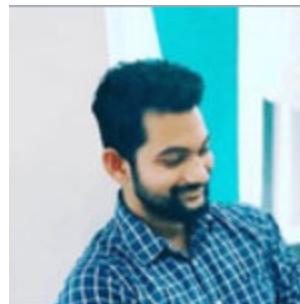
Figure 2.14: JSX converted by Babel

It is not mandatory to use JSX, and we can also use `React.createElement` to create elements in components. It is recommended to use JSX because it makes development easier. Hope this gives you more clarity on what is JSX and how you will write code using JSX.

Inline and external styles

To add CSS styles, there are two options—Inline styles and External CSS files.

As you can see in the following image, the user card does not really look like a card without the CSS. Let us try adding styles using both inline and external CSS, refer to the following image:

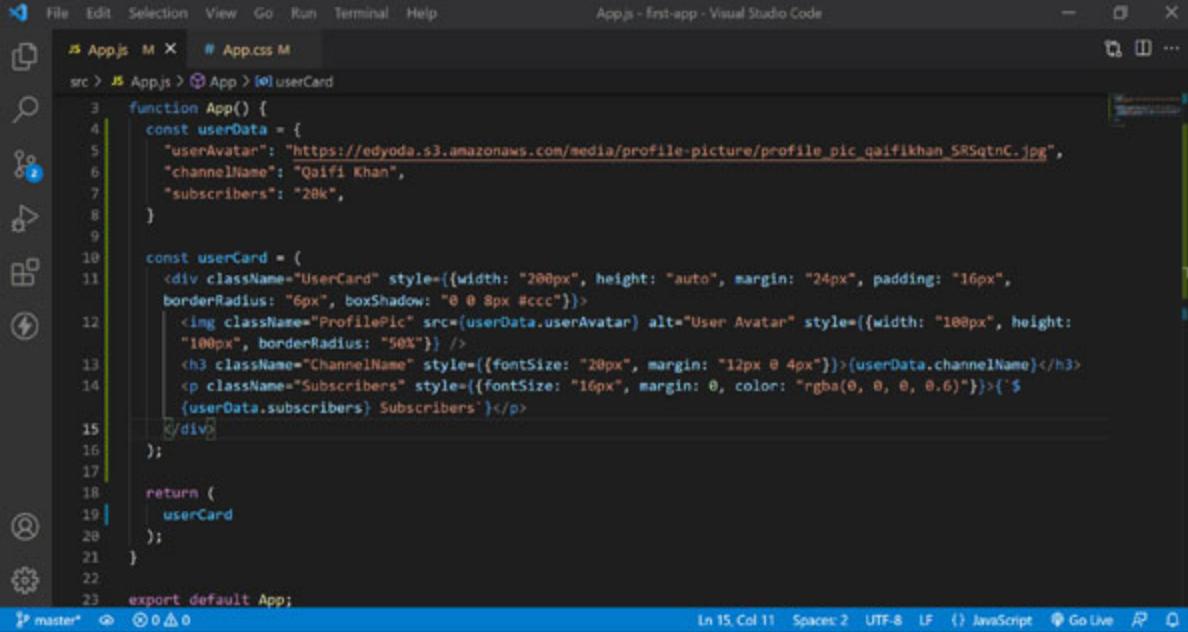


Qaifi Khan

20k Subscribers

Figure 2.15: User card without styles

Just like normal HTML, you have an attribute called `style` to add inline styles. This attribute expects an object. Inside that object, you pass CSS properties and their values as key-value pairs. Remember, for property names, the hyphens are removed, and the names are converted to camelCase, just like we did in vanilla JavaScript. So, let us give it a try:



The screenshot shows the Visual Studio Code interface with the file `App.js` open. The code defines a `function App()` that creates a `UserCard` component. The `UserCard` component is rendered as a `<div>` element with the class `UserCard` and an inline style object. This inline style object sets the width to 200px, height to auto, margin to 24px, padding to 16px, border-radius to 6px, and box-shadow to 0 0 8px #ccc. Inside this div, there is an `` element with the class `ProfilePic`, which has an `src` attribute set to `userData.userAvatar`. The `ProfilePic` element also has an inline style object with a width of 100px and a border-radius of 50%. Below the image, there is an `<h3>` element with the class `ChannelName` and an `<p>` element with the class `Subscribers`. Both of these elements have inline style objects with a font size of 20px and 16px respectively, and a margin of 12px and 8px. The `Subscribers` element also includes a template literal with the value `{userData.subscribers} Subscribers`.

```
File Edit Selection View Go Run Terminal Help
App.js - First-app - Visual Studio Code
App.js M X # App.css M
src > JS App.js > App > userCard
3 function App() {
4   const userData = {
5     "userAvatar": "https://edifyoda.s3.amazonaws.com/media/profile-picture/profile_pic_qaifikhhan_SRSGtnC.jpg",
6     "channelName": "Qaifi Khan",
7     "subscribers": "28k",
8   }
9
10  const userCard = (
11    <div className="UserCard" style={{width: "200px", height: "auto", margin: "24px", padding: "16px", borderRadius: "6px", boxShadow: "0 0 8px #ccc"}>
12      <img className="ProfilePic" src={userData.userAvatar} alt="User Avatar" style={{width: "100px", height: "100px", borderRadius: "50%"} />
13      <h3 className="ChannelName" style={{fontSize: "20px", margin: "12px 0 4px"}>{userData.channelName}</h3>
14      <p className="Subscribers" style={{fontSize: "16px", margin: 0, color: "rgba(0, 0, 0, 0.6)"}>{'$' + userData.subscribers} Subscribers</p>
15    </div>
16  );
17
18  return (
19    userCard
20  );
21}
22
23 export default App;
In 15, Col 11 Spaces: 2 UTF-8 LF {} JavaScript Go live P D
```

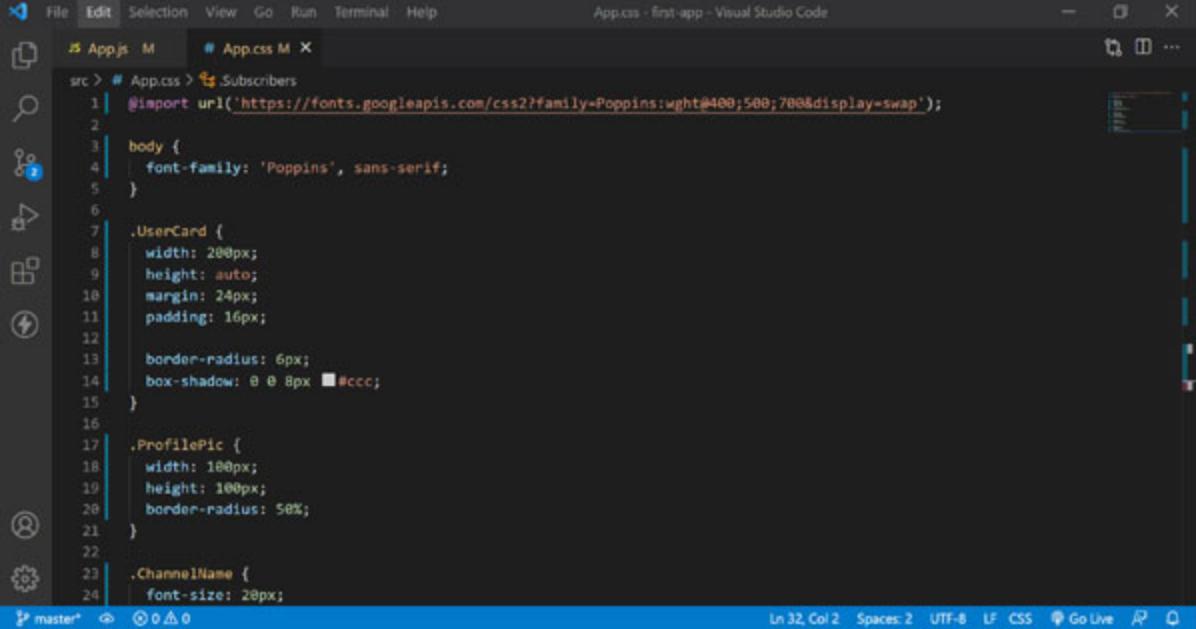
Figure 2.16: User card with inline styles

Because this is an object, we can also put it in a variable. You can use the same object for the elements, which require the same set of styles.

Now, the problem with inline style is that you cannot use pseudo-elements, pseudo-classes, media queries, and a lot more cool features of CSS. However, there are some libraries that help you do that, but we will not get into those right now.

Another option is, we put our styles in an external style sheet. Now to link it in our JS file, we do not use `<link rel="stylesheet" href=".app.css" />`; we can use the import statement. We will learn about import and export in detail in the next chapter. For now, just know that the syntax for CSS file import is “**import keyword followed by file path**”.

Let us move our styles to the CSS file. Now to apply these, we need to select our React elements. Just like in HTML, you would use `class`; similarly, you can use `className` in JSX:



```
File Edit Selection View Go Run Terminal Help
App.css - first-app - Visual Studio Code
App.js M App.css M
src > # App.css > ↗ Subscribers
1 @import url('https://fonts.googleapis.com/css2?family=Poppins:wght@400;500;700&display=swap');
2
3 body {
4   font-family: 'Poppins', sans-serif;
5 }
6
7 .UserCard {
8   width: 200px;
9   height: auto;
10 margin: 24px;
11 padding: 16px;
12
13 border-radius: 6px;
14 box-shadow: 0 0 8px #ccc;
15 }
16
17 .ProfilePic {
18   width: 100px;
19   height: 100px;
20   border-radius: 50%;
21 }
22
23 .ChannelName {
24   font-size: 20px;
25 }
```

Figure 2.17: User card with external styles

Just a quick TIP: We can use IDs, but it does not allow the reusability of react components. That is why it is advised not to use IDs, but we have something provided by React for the same purpose. It is called **Refs**. We will talk about Refs and this reusability issue of ID in the upcoming chapters.

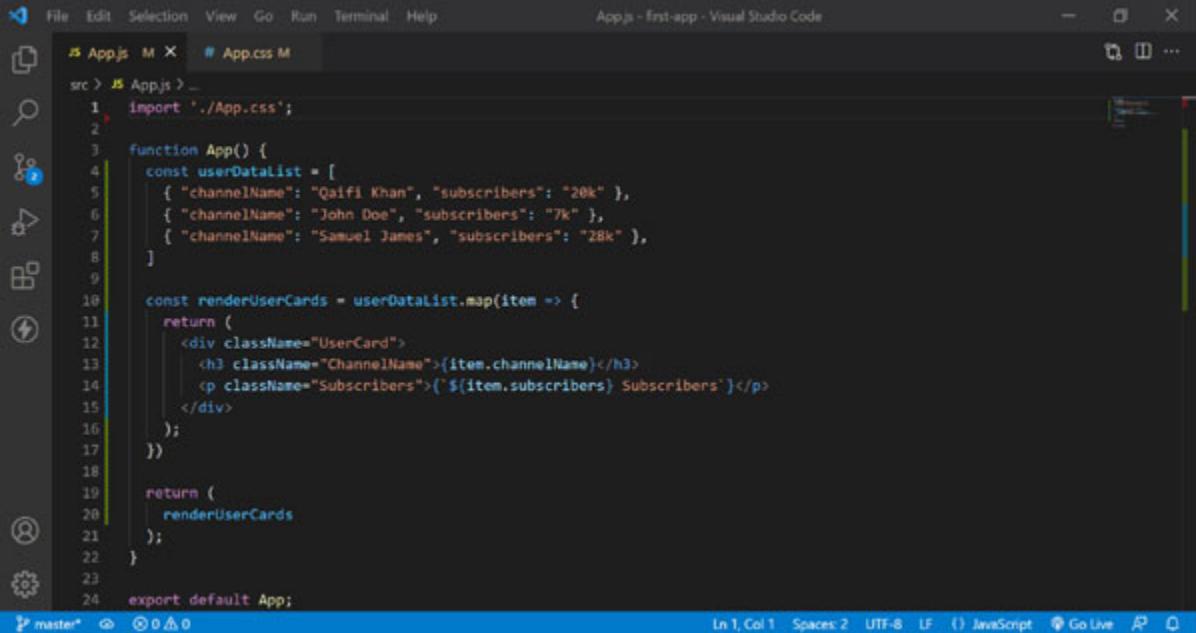
Rendering dynamic elements using objects and lists

In this chapter, we will learn how to dynamically generate react elements using arrays.

Let us say we have the following list of objects which hold data for user cards. Let us try creating a list of user cards using this array dynamically.

```
const userDataList = [
  { "channelName": "Qaifi Khan", "subscribers": "20k" },
  { "channelName": "John Doe", "subscribers": "7k" },
  { "channelName": "Samuel James", "subscribers": "28k" }
]
```

We can iterate through this array, and it will render multiple cards:



The screenshot shows a Visual Studio Code interface with a dark theme. The left sidebar contains icons for file operations like Open, Save, and Find. The main editor area displays the following JSX code:

```
src > App.js M X # App.css M
1 import './App.css';
2
3 function App() {
4   const userDataList = [
5     { "channelName": "Qaifi Khan", "subscribers": "20k" },
6     { "channelName": "John Doe", "subscribers": "7k" },
7     { "channelName": "Samuel James", "subscribers": "28k" },
8   ]
9
10  const renderUserCards = userDataList.map(item => {
11    return (
12      <div className="UserCard">
13        <h3 className="ChannelName">{item.channelName}</h3>
14        <p className="Subscribers">{item.subscribers}</p>
15      </div>
16    );
17  })
18
19  return (
20    renderUserCards
21  );
22}
23
24 export default App;
```

The status bar at the bottom shows the file path as 'master' and other details like 'Ln 1, Col 1', 'Spaces: 2', 'UTF-8', and 'JavaScript'.

Figure 2.18: Dynamic user cards—JSX code

As you can see in the following image, we get the elements rendered in the browser:

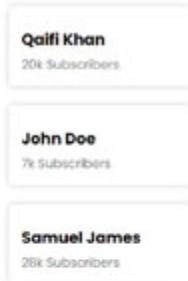


Figure 2.19: User card with external styles—UI

If you check the console, react must be giving us the following error:

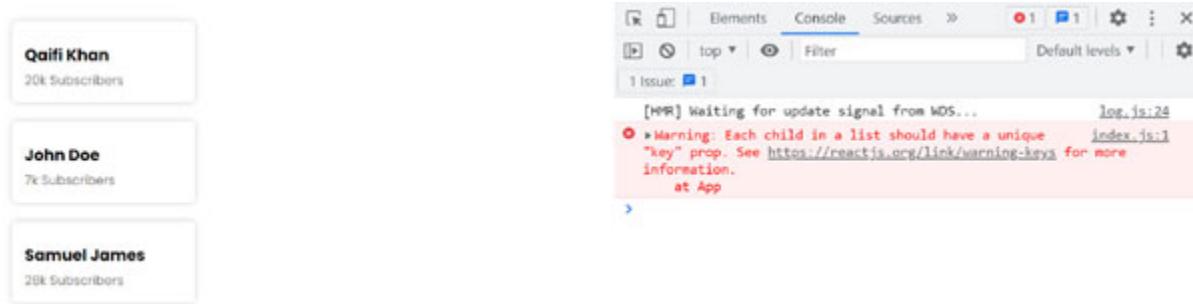


Figure 2.20: Key warning

When we create static elements, React keeps track of those elements automatically. When we create dynamically, we need to help React track these elements. For this reason, we have to add an attribute called the key. It helps React to identify which items have changed, added, or removed.

Now two things to remember, a key should:

- uniquely identify a list item among its siblings
- be given to the outermost element, which is rendered inside the list.

Let us give it a try.

We can either use the position which is being passed by map as an argument to the callback function. That is going to be unique for the entire list. We can use the id which might be provided for the data.

The screenshot shows the code for `App.js` in Visual Studio Code. The code defines a function `App` that creates a list of user cards. The `map` function is used to render each card, and the `key` prop is set to the current index `pos`.

```
File Edit Selection View Go Run Terminal Help
App.js - first-app - Visual Studio Code
src > App.js > ...
1, import './App.css';
2
3 function App() {
4   const userDataList = [
5     { "channelName": "Qaifi Khan", "subscribers": "20k" },
6     { "channelName": "John Doe", "subscribers": "7k" },
7     { "channelName": "Samuel James", "subscribers": "28k" },
8   ]
9
10 const renderUserCards = userDataList.map((item, pos) => {
11   return (
12     <div className="UserCard" key={pos}>
13       <h3 className="ChannelName">{item.channelName}</h3>
14       <p className="Subscribers">{item.subscribers} Subscribers</p>
15     </div>
16   );
17 }
18
19 return (
20   renderUserCards
21 );
22 }
23
24 export default App;
```

Figure 2.21: Key added for dynamic elements

Just a quick tip, using position is considered a bad practice, so use the object's id whenever possible. You can also create a function that generates unique keys.

And as you can see in the following image, the warning is resolved:

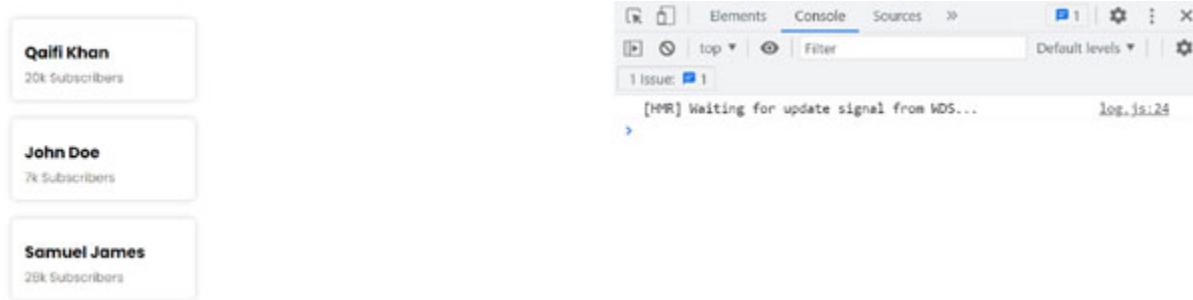


Figure 2.22: Key warning resolved

Conclusion

React is one of the most popular frontend libraries. It can be used to build both single and multi-page applications. It can be used to build an end-to-end application, or we can use it in our existing Web applications. It is based on component design. It uses JSX to create elements that allows us to bind the UI and business logic.

In the upcoming chapter, we will dive deep into components. We will learn about data management, CSS Modules, exports, and imports of components.

Questions

- Q1. What is ReactJS?
- Q2. What are SPAs and MPAs?
- Q3. What are JSX and Babel?
- Q4. Why do we need to use the key attribute when generating elements dynamically?
- Q5. Why is it not preferred to use IDs for styling?

Multiple choice questions

- Q1. React is a**

- a. package
- b. library
- c. framework
- d. compiler

Q2. It is mandatory to use JSX for the creation of React component

- a. True
- b. False

Q3. We can write expressions in JSX using the following:

- a. Curly Brackets
- b. Square Brackets
- c. Backticks
- d. None of the above

Q4. React can be used to develop:

- a. Single-page applications
- b. Multi-page applications
- c. Both (a) and (b)
- d. None of the above

Q5. Single-page application refreshes the browser while usage?

- a. True
- b. False

Answers

Question	Correct Answer
Q1	b. library
Q2	b. false
Q3	a. Curly Brackets
Q4	c. Both (a) and (b)

Q5	b. False
-----------	----------

CHAPTER 3

Understanding Components, State, and Props

In this chapter, we will learn about components and their types. We will deep-dive into class-based and functional components. We will learn about handling component data using state. We will learn how to pass data from one component to another using props. We will create our first component and pass custom values to it. We will learn about CSS modules and how they help us avoid all the challenges of global and inline styles. We will talk about both named and default exports. We will learn how to import methods across JavaScript files.

Structure

In this chapter, we will discuss the following topics:

- Class-based and functional components
- Data handling using state and props
- Extracting components
- Understanding props
- Introduction to CSS modules
- Setting up CSS modules in a React project
- Styling components using CSS Modules
- Media queries with CSS modules
- How to reuse components and modules in React projects
- Default exports
- Named exports
- Module import

Objectives

After studying this chapter, you will be able to understand the different types of components. You will know how to handle data inside a component and pass data from a parent to a child component. You will also be able to create components and should be able to reuse them across the application. You will also know how to work with CSS modules.

Introduction to components

Components let you split the UI into independent and reusable pieces. A component is a JavaScript class or function which returns some React elements. These React elements describe how a section of the UI will appear on the user screen. For example, the **ProductCard** component might return some elements that will render/show a product card on the screen.

In this section, we will talk about Functional components. As you can see in the following image, we have got this product grid. Let us say you have to design this using React.

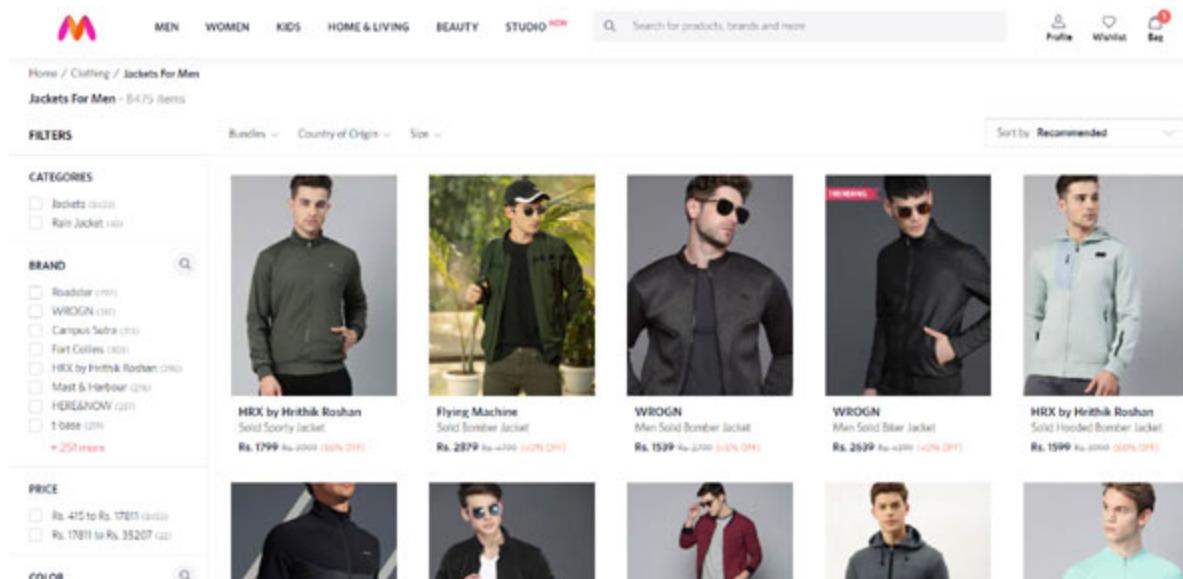


Figure 3.1: Product grid

As shown in the following image, you will write the following code to create one of these product cards:

The screenshot shows a code editor with a file named 'App.js' open. The code is JSX, defining a component named 'App' which returns a 'main' element containing a 'ProductCard' component. Inside 'ProductCard', there is an image of a person wearing an olive green zip-up jacket, followed by brand and product name ('HRX by Hrithik Roshan' and 'Solid Sporty Jacket'), price information ('Rs. 1799'), and a discount note ('(55% OFF)'). To the right of the code editor is a browser window showing the rendered static product card.

```

1 import classes from './styles.module.css';
2
3 export default function App() {
4   return (
5     <main className={classes.MainContainer}>
6       <div className={classes.ProductCard}>
7         
12        <h3 className={classes.Brand}>HRX by Hrithik Roshan</h3>
13        <h3 className={classes.ProductName}>Solid Sporty Jacket</h3>
14        <div className={classes.PriceWrapper}>
15          <p className={classes.ActualPrice}>Rs. 1799</p>
16          <p className={classes.MRP}>Rs. 3999</p>
17          <p className={classes.Discount}>(55% OFF)</p>
18        </div>
19      </div>
20    </main>
21  );
22}

```

Figure 3.2: Static product card

Do not worry about the import and export statements or setting class names as classes object we will learn all about these in the coming sections. For now, just look at Lines 5 to 20. Does it look familiar? It looks exactly like HTML but do not forget that this is JSX. Here, we are assuming that you are aware of core CSS concepts. If yes, then designing the layout will be a walk in the park for you. If you want, you can refer to the CSS file in the working sample that follows.

Open the following link to try a working example of a product card:

[https://codesandbox.io/s/mynttra-project-app-component-ii9tg?
file=/src/App.js](https://codesandbox.io/s/mynttra-project-app-component-ii9tg?file=/src/App.js)

The preceding code represents an individual unit which is a product card. You can have multiple such product cards all across the entire application—on the homepage, search page, order page, and so on, which means it will be reused. We can extract it and convert it into a component because the whole job of a component is to make UI elements reusable.

So, let us give it a try.

Start by creating a new file. Say **ProductCard.js**. The file has a “.js” extension because we need to write some JavaScript code. We are creating a function-based component, so let us write a function.

```
const ProductCard = () => {  
}
```

A component defines part of the UI, so this function needs to return some JSX code. In this case, the JSX code needs to return the following elements to design the product card—product image, product name, product brand, product cost, product MRP, and discount. Basically, the same structure that we created earlier.

```
const ProductCard = () => {
  return(
    
    <h3 className={classes.Brand}>HRX by Hrithik Roshan</h3>
    <h3 className={classes.ProductName}>Solid Sporty Jacket</h3>
    <div className={classes.PriceWrapper}>
      <p className={classes.ActualPrice}>Rs. 1799</p>
      <p className={classes.MRP}>Rs. 3999</p>
      <p className={classes.Discount}>(55% OFF)</p>
    </div>
  )
}
```

If you run the preceding code, it is going to throw an error. Try to guess the error before running the code. Do not worry if you are unable to guess the error. It throws the error as shown in the image:

SyntaxError

```
/src/ProductCard.js: Adjacent JSX elements must be wrapped in an enclosing tag. Did you want a JSX fragment <>...</>? (8:4)
```

```
6 |         alt="HRX by Hrithik Roshan"
7 |     />
> 8 |     <h3 className={classes.Brand}>HRX by Hrithik Roshan</h3>
|     ^
9 |     <h3 className={classes.ProductName}>Solid Sporty Jacket</h3>
10 |    <div className={classes.PriceWrapper}>
11 |      <p className={classes.ActualPrice}>Rs. 1799</p>
```

```
(anonymous function)
https://codesandbox.io/static/js/sandbox.5800721de.js:1:186608
```

```
e.value
https://codesandbox.io/static/js/sandbox.5800721de.js:1:186727
```

```
e.value
This screen is visible only in development. It will not appear if the app crashes in production.
Open your browser's developer console to further inspect this error.
This error overlay is powered by `react-error-overlay` used in `create-react-app`.
```

Figure 3.3: Single parent in a component error

The error says, “*Adjacent JSX elements must be wrapped in an enclosing tag*”, which means that a component needs to have a single parent element, or in other words, it cannot return multiple elements. So how do we fix this error? We have two options: either we can wrap it in a div or a section or some other HTML tag based on the code requirement. We can also use an empty tag `<></>` to wrap elements when it is not required. This empty tag is called **JSX Fragment**. In our case, all the card elements need to be wrapped together to design a card, so we will wrap it in a div tag as shown in the following, and the error will disappear.

```
const ProductCard = () => {
  return (
    <div className={classes.ProductCard}>
      
      <h3 className={classes.Brand}>HRX by Hrithik Roshan</h3>
      <h3 className={classes.ProductName}>Solid Sporty
      Jacket</h3>
      <div className={classes.PriceWrapper}>
        <p className={classes.ActualPrice}>Rs. 1799</p>
        <p className={classes.MRP}>Rs. 3999</p>
        <p className={classes.Discount}>(55% OFF)</p>
      </div>
    </div>
  );
}

```

Now our component is ready. But how do we use it in other files?

We can simply import it. Just like we imported the React module. Before your component can be imported, first, you need to export it.

Now, there are two ways we can export a module:

- **Default export:** We use default export when we want to export only one module, function, or component from a file.
- **Named export:** We use named export when we want to export multiple modules or functions from the same file.

We will talk more about exports and imports later in this chapter itself. But for now, just remember that this is the basic concept.

In this case, we want to just export one component, which is **ProductCard**. So, we can use default export. The syntax for default export is as follows:

```
export default {Component-Name};
```

So, we can simply do this,

```
export default ProductCard; //Add this code as the last line
in your component file.
```

This line of code will export our **ProductCard**, which means we can now import it into other files.

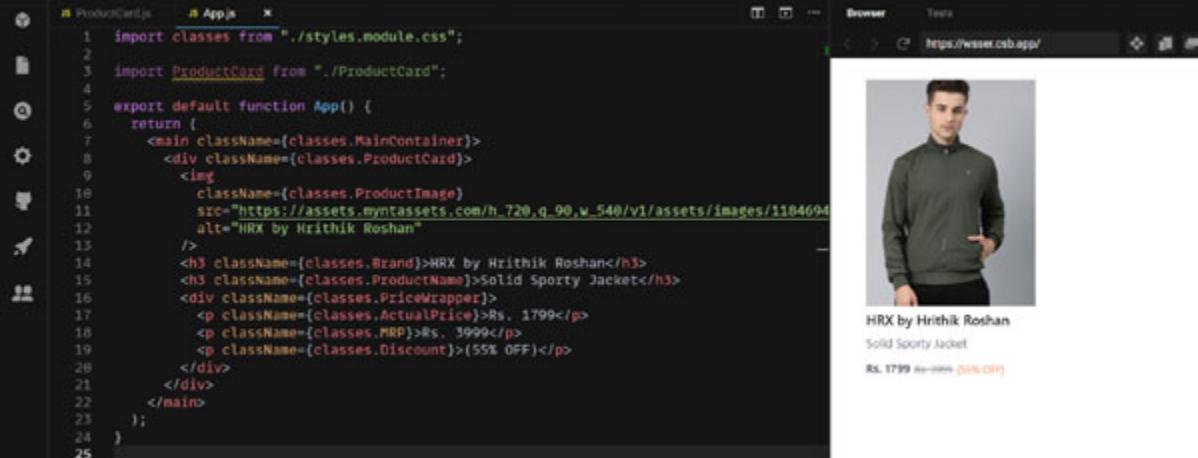
So, to import this component into the App component, we can write the following code:

```
import ProductCard from './ProductCard/ProductCard'
```

So, we have imported the component, and it is ready to use.

Please note: We do not write file extensions for JS files. It is automatically configured by create-react-app, but we will need to mention the extensions for other file types.

Currently, our App component looks something like this (refer to [figure 3.4](#)):



The screenshot shows a code editor with the file 'App.js' open. The code defines a functional component 'App' that returns a main container with a product card. The product card displays an image of a person wearing a green jacket, the brand name 'HRX by Hrithik Roshan', the product name 'Solid Sporty Jacket', the actual price 'Rs. 1799', the MRP 'Rs. 3999', and a discount message '(55% OFF)'. Below the code editor is a browser window showing the rendered output of the code.

```
ProductCard.js  App.js
1 import classes from "./styles.module.css";
2
3 import ProductCard from "./ProductCard";
4
5 export default function App() {
6   return (
7     <main className={classes.MainContainer}>
8       <div className={classes.ProductCard}>
9         
14         <h3 className={classes.Brand}>HRX by Hrithik Roshan</h3>
15         <h3 className={classes.ProductName}>Solid Sporty Jacket</h3>
16         <div className={classes.PriceWrapper}>
17           <p className={classes.ActualPrice}>Rs. 1799</p>
18           <p className={classes.MRP}>Rs. 3999</p>
19           <p className={classes.Discount}>(55% OFF)</p>
20         </div>
21       </div>
22     </main>
23   );
24 }
25 }
```

Figure 3.4: App component JSX code

To render a component, we can simply treat it as a JSX tag, as shown in the following code snippet:

```
<ProductCard />
```

Please note: This is a self-closing tag because it does not require any child components. We will learn more about it when we learn about something called higher-order components.

After adding the component, our code looks as shown in the following, and we can still see the product card in the output section.

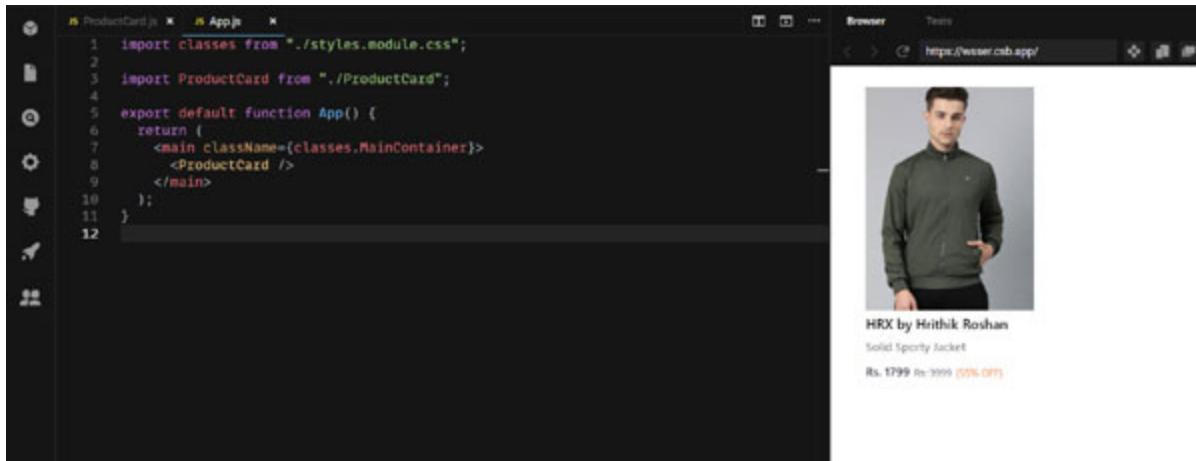


Figure 3.5: Loading product card using a component

If you try to inspect this in the browser, it will not show you the **ProductCard** component. It will show the HTML elements inside that component, as shown in the following screenshot:



Figure 3.6: Rendered component as HTML elements

Even though, as a developer, we will write JSX and components. When the application is loaded in the browser, it is converted to normal HTML because the browser does not understand components or JSX.

Hope this gives you some clarity on how to create and use components.

Understanding props

In the previous section, we created the **ProductCard** component. Now, this is a static component which means it just renders the same structure and

data every time we use it, as shown in the following screenshot:

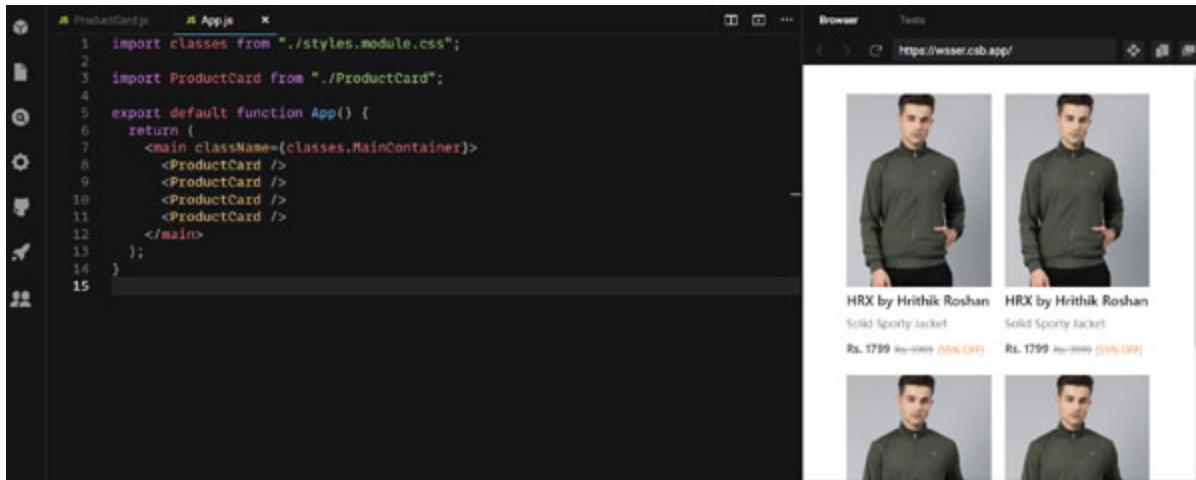


Figure 3.7: Static product grid

In order to fully use the power of components, we need this component to generate different components based on different data. This is where *props* come into play. If you remember, when we were working with Babel, we talked about how all the attributes got converted to an object, and that object is passed as the second argument in the **createElement()** method of React. Let us check it out.

To pass some data to the component, we can simply write it as properties to the component tag, as shown in the following code snippet. We can pass anything to the component—strings, numbers, arrays, objects, or even function references.

```
<ProductCard
  thumbnail={}
  brand={}
  title={}
  mrp={}
  discount={} />
```

So, heading to Babel. As you can see in the following image, Babel combines these properties into an object.

The screenshot shows the Babel REPL interface. On the left, there is a code editor with the following JavaScript code:

```

1 <ProductCard
2   thumbnail={
3     "https://assets.myntassets.com/h_720,q_90,w_540/v1/assets/images/11846944/
4 2020/8/6/448009c3-ea40-462c-bde1-8b23b6ff25041596710361858-HRX-by-Hrithik-
5 Roshan-Men-Jackets-1201596710359343-1.jpg"
6   }
7   brand="HRX by Hrithik Roshan"
8   title="Solid Sporty Jacket"
9   mrp={3999}
10  discount={55}
11 >

```

On the right, the generated ES6 code is shown:

```

1 "use strict";
2
3 /*#__PURE__*/
4 React.createElement(ProductCard, {
5   thumbnail:
6     "https://assets.myntassets.com/h_720,q_90,w_540/v1/assets/images/11846944/
7 2020/8/6/448009c3-ea40-462c-bde1-8b23b6ff25041596710361858-HRX-by-Hrithik-
8 Roshan-Men-Jackets-1201596710359343-1.jpg",
9   brand: "HRX by Hrithik Roshan",
10  title: "Solid Sporty Jacket",
11  mrp: 3999,
12  discount: 55
13 });

```

Figure 3.8: Component rendering behind-the-scenes using Babel

This object is passed as an argument to the component. This object is called *props*. You can name it anything else you want, but it is recommended to name it as *props*. The next question is how we access these values inside the component to make it dynamic. Let us give that a try.

First, we add props as an argument to the function-based component, as shown in the following screenshot:

The screenshot shows a code editor and a browser side-by-side. The code editor has the following code:

```

1 import classes from "./styles.module.css";
2
3 import ProductCard from "./ProductCard";
4
5 export default function App() {
6   return (
7     <main className={classes.MainContainer}>
8       <ProductCard
9         thumbnail={
10           "https://assets.myntassets.com/h_720,q_90,w_540/v1/assets/images/11846944/
11         }
12         brand="HRX by Hrithik Roshan"
13         title="Solid Sporty Jacket"
14         mrp={3999}
15         discount={55}
16       />
17       <ProductCard
18         thumbnail={
19           "https://assets.myntassets.com/h_720,q_90,w_540/v1/assets/images/12446056/
20         }
21         brand="Flying Machine"
22         title="Solid Bomber Jacket"
23         mrp={4799}
24         discount={40}
25       />
26     </ProductCard>

```

The browser window shows the rendered output of the component, displaying two products:

- HRX by Hrithik Roshan Solid Sporty Jacket**: Price: Rs. 1799, Sale: Rs. 1399 (30% OFF)
- Flying Machine Solid Bomber Jacket**: Price: Rs. 2879, Sale: Rs. 1799 (40% OFF)

Figure 3.9: Passing data to a component

As discussed earlier, props are an object which is accessible inside a component. In this case, the props object will have the following properties—**thumbnail**, **brand**, **price**, **mrp**, and **discount**. To access any of these properties, we can simply write the following code—**props.thumbnail**, **props.title**, **props.mrp**, and so on. We need to refactor our **ProductCard** component to use these props values rather than hard-coded or static values. After the changes, it will look as shown in the following:

The screenshot shows a code editor on the left displaying `App.js` and a browser window on the right showing the output of the code. The code in `App.js` defines a `ProductCard` component that takes props like `brand`, `title`, `MRP`, and `discount`. It calculates a discounted price and displays it along with the brand name and product title. The browser window shows four product cards with images, brand names, titles, and prices.

```

1 import classes from "./styles.module.css";
2
3 const ProductCard = (props) => {
4   const calcPrice = () => Math.floor(props.MRP * (1 - props.discount / 100));
5   return (
6     <div className={classes.ProductCard}>
7       <img
8         className={classes.ProductImage}
9         src={props.thumbnail}
10        alt={props.title}
11      />
12      <h3 className={classes.Brand}>{props.brand}</h3>
13      <h3 className={classes.ProductName}>{props.title}</h3>
14      <div className={classes.PriceWrapper}>
15        <p className={classes.ActualPrice}>{'Rs. ${calcPrice()}'}</p>
16        <p className={classes.MRP}>{'Rs. ${props.MRP}'}</p>
17        <p className={classes.Discount}>{'(${props.discount} % OFF)'}</p>
18      </div>
19    </div>
20  );
21 }
22
23 export default ProductCard;
24

```

Figure 3.10: Reading values from the `props` object

Open the following link to try a working example of a product card:

[https://codesandbox.io/s/productcard-static-component-wsser?
file=/src/App.js](https://codesandbox.io/s/productcard-static-component-wsser?file=/src/App.js)

Hope this gives you clarity on how to pass props to generate dynamic components.

Imports and exports

There are two ways we can export a module—Default export and Named export. You should use default export when you want to export only one module from a file. In other words, a file cannot have multiple default exports. For example, as a standard practice, we will only create one component in one file, and to make these components available in the application, we will use default export. When working with default exports, you can import the module by any name you like.

Earlier, we exported the `ProductCard` component using default export, and we imported it with the same name as shown in the following:

The screenshot shows a code editor with a file named `ProductCard.js`. The code defines a component `ProductCard` that takes props like `brand`, `title`, `mrp`, and `discount`. It calculates a discounted price and renders a card with the brand name, product title, price, and discount percentage. The browser preview shows two products: 'HRX by Hrithik Roshan Solid Sporty jacket' and 'Flying Machine Solid Bomber Jacket', both with their respective prices and discounts.

```

1 import classes from "./styles.module.css";
2
3 const ProductCard = (props) => {
4   const calcPrice = () => Math.floor(props.mrp * (1 - props.discount / 100));
5   return (
6     <div className={classes.ProductCard}>
7       <img
8         className={classes.ProductImage}
9         src={props.thumbnail}
10        alt={props.title}
11      />
12      <h3 className={classes.Brand}>{props.brand}</h3>
13      <h3 className={classes.ProductName}>{props.title}</h3>
14      <div className={classes.PriceWrapper}>
15        <p className={classes.ActualPrice}>Rs. ${calcPrice()}</p>
16        <p className={classes.MRP}>Rs. ${props.mrp}</p>
17        <p className={classes.Discount}>{`(${props.discount} % OFF)`}</p>
18      </div>
19    </div>
20  );
21}
22
23 export default ProductCard;
24

```

Figure 3.11: Import with the default name

Now, let us try importing it with a different name, say `ClothingCard` then we will also have to render the card using `ClothingCard` as the name rather than `ProductCard`, as shown in the following figure:

The screenshot shows a code editor with a file named `App.js`. It imports `ClothingCard` from `./ProductCard`. The `App()` function returns a main container with two `ClothingCard` components. Each card has a thumbnail, brand name ('HRX by Hrithik Roshan' or 'Flying Machine'), product title ('Solid Sporty Jacket' or 'Solid Bomber Jacket'), price ('Rs. 1799'), and discount ('55%' or '40%'). The browser preview shows the same two products as Figure 3.11.

```

1 import classes from "./styles.module.css";
2
3 import ClothingCard from "./ProductCard";
4
5 export default function App() {
6   return (
7     <main className={classes.MainContainer}>
8       <ClothingCard
9         thumbnail={
10           "https://assets.myntassets.com/h_720,q_90,w_540/v1/assets/images/11846944"
11         }
12         brand="HRX by Hrithik Roshan"
13         title="Solid Sporty Jacket"
14         mrp={3999}
15         discount={55}
16       />
17       <ClothingCard
18         thumbnail={
19           "https://assets.myntassets.com/h_720,q_90,w_540/v1/assets/images/12446056"
20         }
21         brand="Flying Machine"
22         title="Solid Bomber Jacket"
23         mrp={4799}
24         discount={40}
25       />
26     </ClothingCard>
27   )
28 }
29
30 export default App;
31

```

Figure 3.12: Import with a custom name

Let us talk about the named exports. You will use named exports when you want to export multiple modules from the same file. Say, for example, we have a utility file where we keep all our helper methods. Let us say this file has two methods: the first one checks whether a string is empty or not, and the second method prints the logs and sends a copy to a tool where you track the production logs.

Now, this file has two methods, and to make them accessible in other files, we can use named exports.

Syntax:

```
export const moduleName = value;
```

For example,

```
export const isEmptyString = (str) => {
    //Function Body
}
export const logger = (msg) => {
    //Function Body
}
```

To import these named export modules or functions, the syntax is a bit different.

Syntax:

```
import {module-name} from 'module-relative-path';
```

For example, to import the **isEmptyString()** method, you will have to write the following code:

```
import { isEmptyString } from './utils/CommonMethods';
```

One important thing to remember, when working with named exports, you cannot import modules with different names. If you write the following code, then it will throw an error:

```
import { checkIfStringEmpty } from './utils/CommonMethods';
```

The preceding code will try to search for a named export module with the name **checkIfStringEmpty** in the **CommonMethods.js** file. This **checkIfStringEmpty()** does not exist in the file because we did not write it, and that is why it will throw an error saying, “*CommonMethods.checkIfStringEmpty is not a function*”.

We do have a solution; if you want to import it with a different name, you can do something like this.

```
import { isEmptyString as checkIfStringEmpty } from
'./utils/CommonMethods';
```

When you are importing multiple modules from the same file, you can write both module names inside a single import statement, as shown in the following:

```
import { isEmptyString as checkIfStringEmpty, logger } from
'./utils/CommonMethods';
```

That is all about module import and export. Hope now you know how to reuse components and modules across the entire application.

CSS modules

If you have worked with Vanilla JS or static Web apps, you must be familiar with issues that arise due to global CSS. It is so difficult to track which styles are being loaded from which file, and everything gets messy. This issue is solved by using CSS modules.

Another important thing to remember is that components are modular and independent. Having global styles adds a dependency between different components. CSS changes in one stylesheet might affect other components. To make components truly independent, we need to also make style classes scoped locally to the component.

This is where the CSS module comes into the picture. It creates a local scope for the CSS classes. It makes our components completely independent and saves our app from all the troubles of global classes.

So, how exactly does it create locally scoped styles? A unique class name is created for all the classes that we write in the CSS modules file. This unique class name is made up of the CSS module file name, class name, and a random hash value, as shown in the following image:

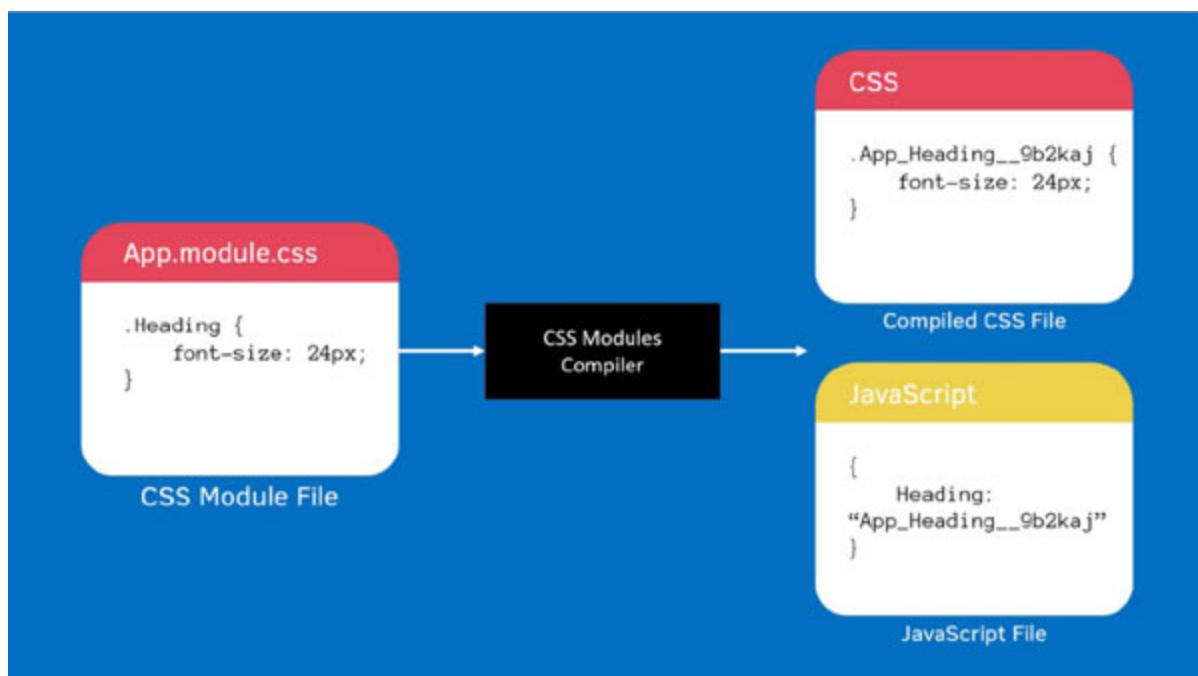


Figure 3.13: Working of CSS modules

In the CSS module file, the class name would be **Heading**, but in the compiled CSS file, the class name gets converted to **App_Heading_9b2kaj**. This conversion is done by a CSS modules compiler which is also set up automatically by create-react-app.

So how would you write the class name in the JSX code? Would you write **App_Heading_9b2kaj** as the **className**?

No, this CSS modules compiler also creates a JavaScript object where all the original classes are mapped to the modified names. This object is what we will be using inside our components to add classes to React elements. This mapping object is what we import into the component for writing class names.

The create-react-app automatically installs the CSS modules for you, so to enable it, you do not need to install a new package or library. To create a CSS module file, we simply give a file an extension as “**filename.module.css**”. This **.module** tells ReactJS that this is a CSS module file and it is handled differently than a normal CSS file.

Now that we know how to create a CSS module file let us understand how to use it. Please refer to the following syntax for importing the CSS module file.

Syntax:

```
import classes from './ProductCard.module.css';
```

As shown in the preceding code, we are importing the CSS module file as default export, which means it is not mandatory to import it as classes; you can import it with any name you want. You can call it styles or whatever else you like, but classes make sense; that is why we use them as classes. Just remember you are loading a file that is not a JavaScript file, so you have to mention the entire name along with the extension.

To write a classnames you can simply do the following:

```
classes.{ClassName-In-CSS-Module-File}
```

For example,

```
<div className={classes.ProductCard}>  
  <img  
    className={classes.ProductImage}  
    src={props.thumbnail}>
```

```

        alt={props.title} />
      <h3 className={classes.Brand}>{props.brand}</h3>
      <h3 className={classes.ProductName}>{props.title}</h3>
      <div className={classes.PriceWrapper}>
        <p className={classes.ActualPrice}>{`Rs. ${calcPrice()}`}</p>
        <p className={classes.MRP}>{`Rs. ${props.mrp}`}</p>
      </div>
    </div>
  
```

If you inspect the HTML elements, you will notice that all the localized class names are used.

Hope this gives you clarity on what CSS modules are and how we can use them to avoid global class issues.

Responsive components

Say you want to make your ProductGrid responsive. In other words, you want two cards on mobile screens, three cards on tablet screens, four cards on laptop screens, and so on.

To make components responsive, you can simply use media queries in the CSS modules file just like you would use it in normal CSS files, as shown in the following screenshot:

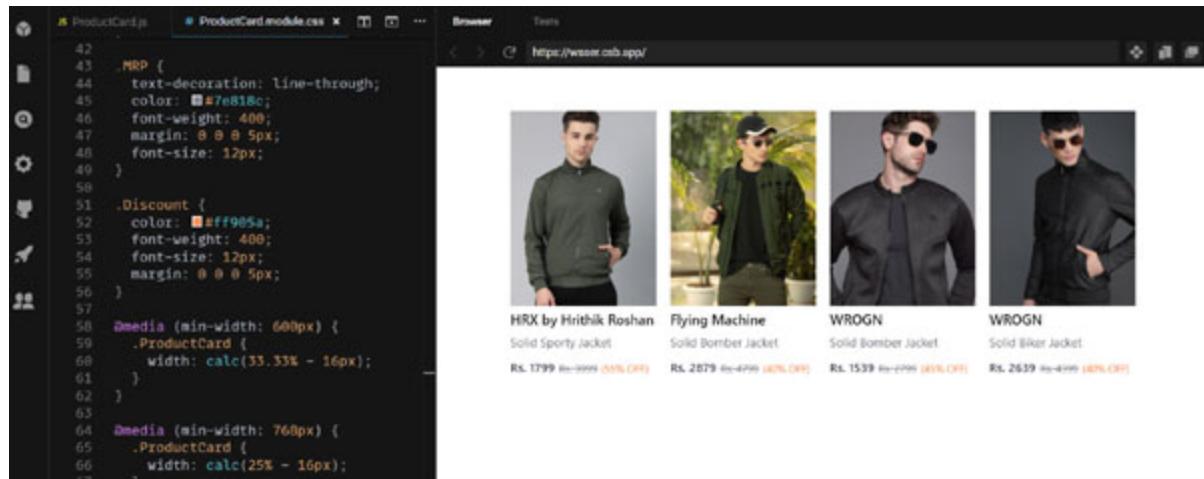


Figure 3.14: Responsive component using media query

Stateful and stateless components

So far, we have made components that just show something on the screen. They do not do anything; they have no functionality. These kinds of components are called stateless or presentational components because they do not have any state of their own. They simply get some props and render the UI accordingly.

But say, for example, you want to create a component that creates a dropdown. Now, this dropdown can be opened or closed. In the simplest terms, open or closed is the state of the component. And the components with the state are called stateful components.

The state is similar to props, but it is private and fully controlled by the component. A state is basically an object, and it can hold any type of value, such as a number, string, boolean, array, or objects.

This state functionality is provided by React's component class. So, to make a stateful component, you need to create a class-based component and inherit React's component class. We will talk more about it and create it in the coming sections.

This concept of state divides components into two major types, which are as follows:

- **Stateful components:** They are also known as—Class-based, Container, or Smart components.
- **Stateless components:** They are also known as Function-based, Presentational, or Dumb components.

The literal difference is that one has a state, and the other does not. This just means that the stateful components are keeping track of changing data via state, whereas stateless components print out what is given to them via props, or they always render the same thing.

Just a quick tip: You should keep the number of stateful components as minimum as possible. It will help you avoid a lot of complexity and will save you a lot of debugging time. We will learn how to do that in the upcoming chapters.

Another very important thing to remember is that in React v16.8, a new feature called Hooks was introduced. This feature allows us to have a state even in the functional components. We will introduce the Hooks concept in

the upcoming chapters. For now, let us get you comfortable with the concept of state using class-based components.

Hope this gives you some sense of what is a stateful and stateless component.

Class-based components

Let us say we want to create a counter. We will have an increment counter button which, on click, will increase the counter and another button decrement counter to decrease the counter value by one.

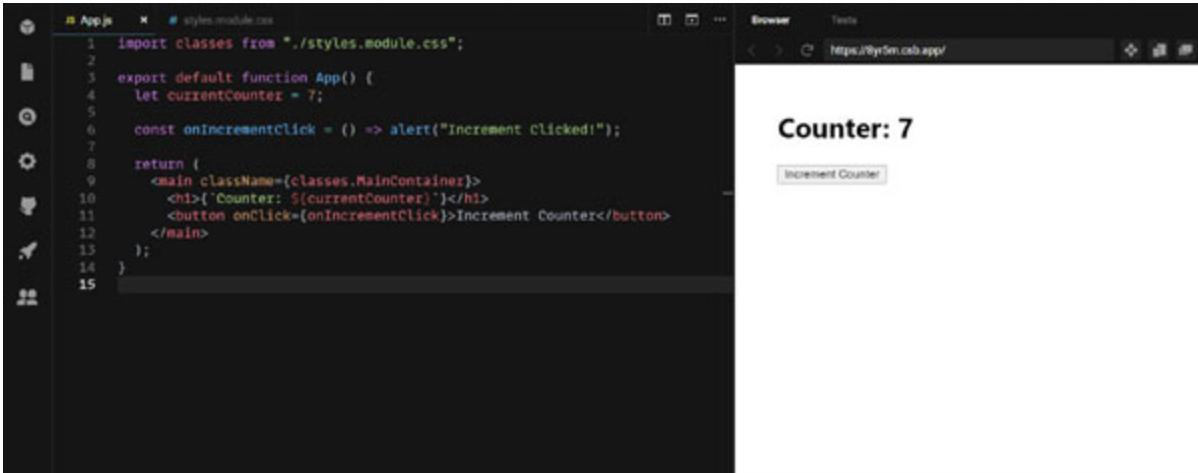
First, let us add a button. To add this button, you can simply write the code shown as follows:

```
<button>Increment Counter</button>
```

Now, we need to do something with a button click. That means we need to add an **onClick** listener. For that, we can simply add it as an attribute. This will expect a callback function so let us create an anonymous function. For now, let us just add an **alert()**.

```
const onIncrementClick = () => alert("Increment Clicked!")
<button onClick={onIncrementClick}>Increment Counter</button>
```

If you run the preceding code on the button click, it will show you the alert, which means our click event is working fine. Now, we want to have a counter which we can increase on button click. So, let us add the element to show the counter and create a global variable that will show the counter value. Our code will look like the one shown in the following image:



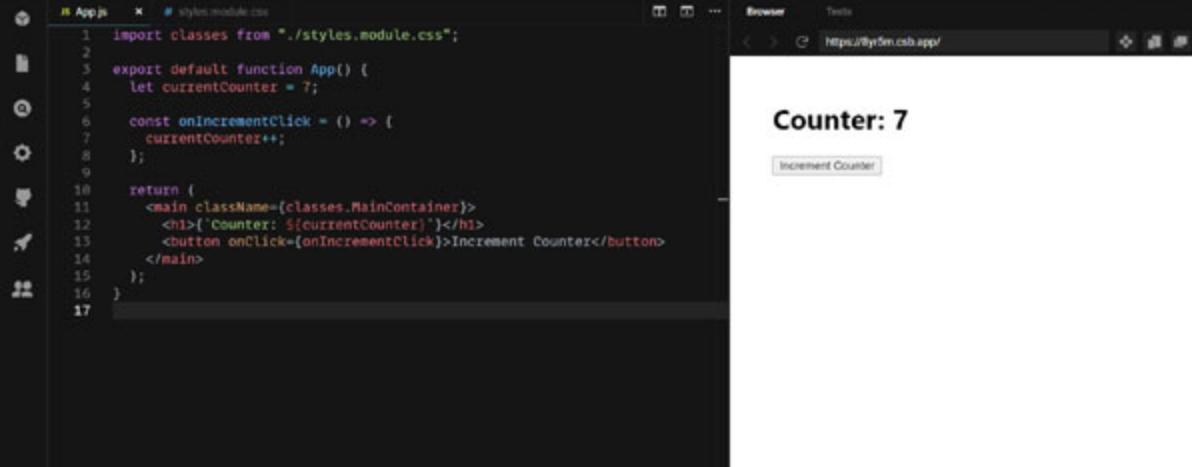
The screenshot shows a code editor on the left and a browser window on the right. The code editor displays the following code:

```
App.js
import classes from './styles.module.css';
export default function App() {
  let currentCounter = 7;
  const onIncrementClick = () => alert("Increment Clicked!");
  return (
    <main className={classes.MainContainer}>
      <h1>Counter: ${currentCounter}</h1>
      <button onClick={onIncrementClick}>Increment Counter</button>
    </main>
  );
}
```

The browser window shows the rendered output of the component. It has a title "Counter: 7" and a single button labeled "Increment Counter".

Figure 3.15: Rendered counter

We want to increase the counter value by one on the button click. Let us write the code for that.



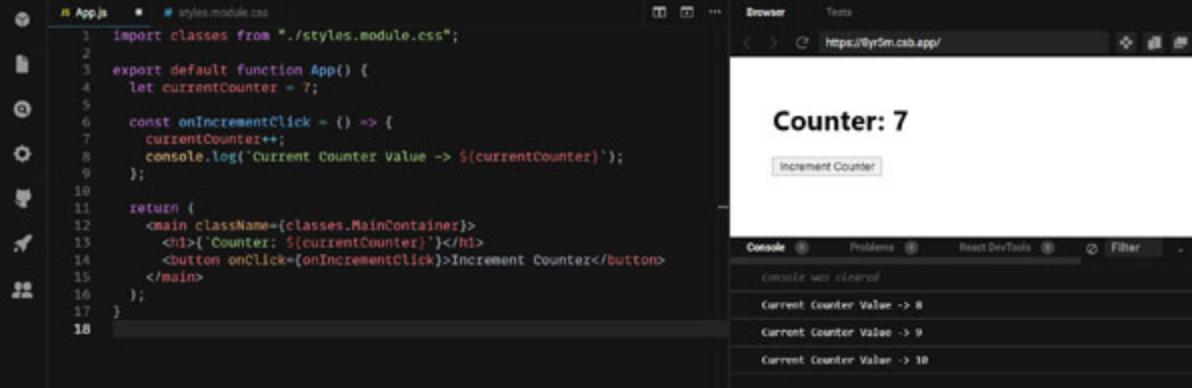
The screenshot shows a code editor on the left and a browser window on the right. The code editor contains the following App.js code:

```
1 import classes from './styles.module.css';
2
3 export default function App() {
4   let currentCounter = 7;
5
6   const onIncrementClick = () => {
7     currentCounter++;
8   };
9
10  return (
11    <main className={classes.MainContainer}>
12      <h1>Counter: ${currentCounter}</h1>
13      <button onClick={onIncrementClick}>Increment Counter</button>
14    </main>
15  );
16}
17
```

The browser window shows a simple UI with the heading "Counter: 7" and a button labeled "Increment Counter".

Figure 3.16: Function to handle counter

If you run the preceding code and click on the button, the counter will still show “7” on the screen. Wait! What is happening? It is a simple code, on button click, the **onIncrementClick()** function is called, and it should increase the **currentCounter** value to “8”. Let us print the updated value in the console to verify our code.



The screenshot shows a code editor on the left and a browser window on the right. The code editor contains the same App.js code as Figure 3.16, but with an additional line of code at line 8:

```
8   const onIncrementClick = () => {
9     currentCounter++;
10    console.log('Current Counter Value -> ${currentCounter}');
11  };
12
```

The browser window shows the same UI as Figure 3.16. In the bottom right corner of the browser window, there is a "Console" tab showing the following log entries:

- Console was cleared
- Current Counter Value -> 8
- Current Counter Value -> 9
- Current Counter Value -> 10

Figure 3.17: Counter values in console

If you check out the console in the bottom right corner, you will notice that on every click, the updated value has been printed. This means our function is working fine, but the UI is not getting updated or re-rendered.

How do we re-render our component on the button click?

This is where the concept of state is helpful. The flow is really simple; we initialize a **state** object and add the **currentCounter** in that **state** object.

The **h1** tag will fetch the counter value from the state. So far, everything is normal stuff, and the magic happens when we use the **setState()** method provided by React's Component class. When we call **setState()** to update the component's state, it updates the state and calls the **render()** method with the updated state. Calling the **render()** method after the component is loaded is called **re-rendering**. For your better understanding, we have added the flow in the following screenshot:

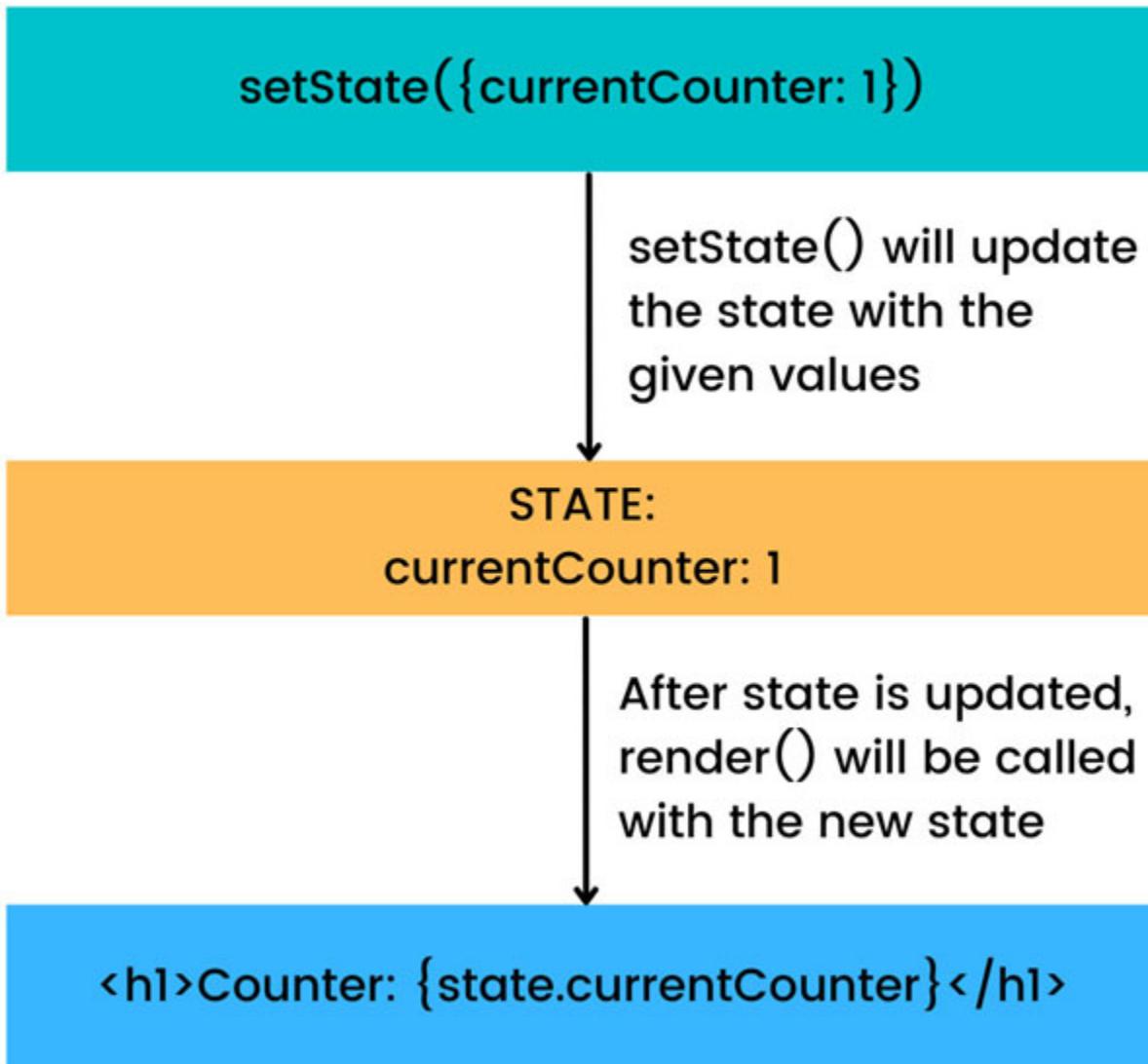
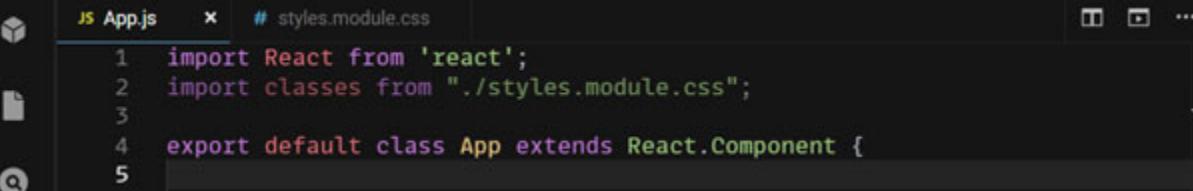


Figure 3.18: Data flow on component update

This state object and **setState()** method are made available in React's component class. So, we need to convert our functional component to a class-based component.

Follow the following steps:

First, change the function to class and inherit React's **Component** class. Always remember to make a class as a React's class-based component; we need to inherit the **Component** class. This **Component** is the base class for all class-based components. To access the parent **Component** class, we can import React from the react module, as shown in the following screenshot:

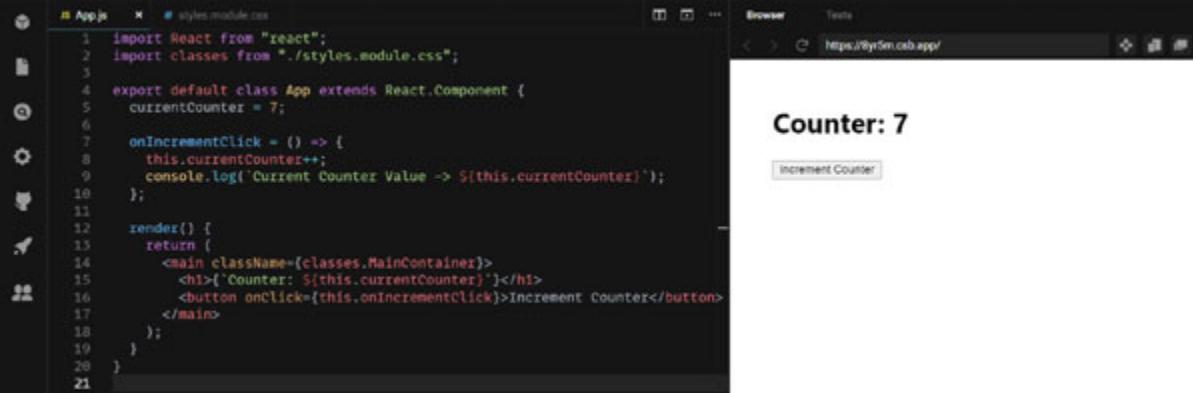


The screenshot shows a code editor with an open file named "App.js". The code starts with importing React and a CSS module, then defines a class "App" that extends React.Component. The code is as follows:

```
1 import React from 'react';
2 import classes from './styles.module.css';
3
4 export default class App extends React.Component {
5 }
```

Figure 3.19: Importing component class from the React module

This **Component** class gives us access to a method called **render()**. It returns all our JSX code or React elements that need to be rendered on the screen. This **render()** method is called every time the component is loaded or updated. So, let us add it. This render method will return all the JSX code, so we need to add a return statement. Let us move all our JSX code inside this return statement. After this change, your code should look similar to the one shown in the following screenshot:



The screenshot shows a code editor with the same "App.js" file, now including a render method that returns the JSX code. The code is as follows:

```
1 import React from "react";
2 import classes from "./styles.module.css";
3
4 export default class App extends React.Component {
5   currentCounter = 7;
6
7   onIncrementClick = () => {
8     this.currentCounter++;
9     console.log(`Current Counter Value -> ${this.currentCounter}`);
10  };
11
12   render() {
13     return (
14       <main className={classes.MainContainer}>
15         <h1>Counter: ${this.currentCounter}</h1>
16         <button onClick={this.onIncrementClick}>Increment Counter</button>
17       </main>
18     );
19   }
20 }
```

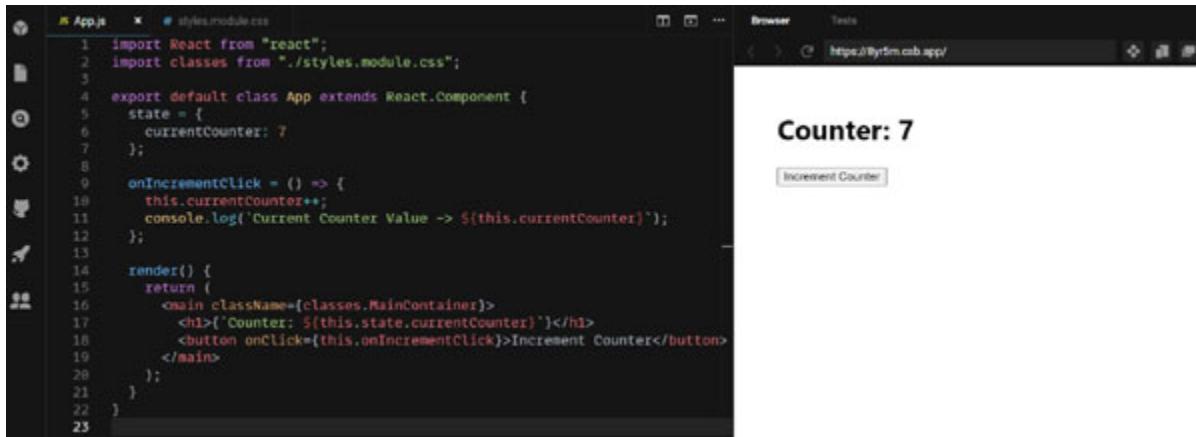
To the right of the code editor, a browser window is open at the URL <https://try5m.net/app/>, displaying the UI. The UI shows the text "Counter: 7" and a button labeled "Increment Counter".

Figure 3.20: Code refactor for class-based component

If you notice, there are a few other changes due to class. We have removed **let** and **const** keywords for variable and function creation because, based on modern JS, these will be treated as class properties and methods. Another change we have made is to access these properties and methods. Now, we have to use **this** keyword. After making these changes, the UI should still look the same, and the button click should still print the updated

currentCounter values in the console. Please verify it before moving to the next step.

The right way to handle the component's data is to store it in the state as shown in the following screenshot:



The screenshot shows a code editor on the left and a browser window on the right. The code editor contains the following code for **App.js**:

```
1 import React from "react";
2 import classes from "./styles.module.css";
3
4 export default class App extends React.Component {
5   state = {
6     currentCounter: 7
7   };
8
9   onIncrementClick = () => {
10     this.currentCounter++;
11     console.log('Current Counter Value -> ${this.currentCounter}');
12   };
13
14   render() {
15     return (
16       <main className={classes.MainContainer}>
17         <h1>{`Counter: ${this.state.currentCounter}`}</h1>
18         <button onClick={this.onIncrementClick}>Increment Counter</button>
19       </main>
20     );
21   }
22 }
23
```

The browser window shows the output of the code. It displays the text "Counter: 7" above a button labeled "Increment Counter".

Figure 3.21: Handling state data

Notice we have moved **currentCounter** as a property inside the state property. Also, in Line 17, we have updated the code to access **currentCounter** value.

Based on the flow which was explained earlier, if we use **setState()** to update the value of **currentCounter** in the component's state, then the **render()** method will be called again with the latest values, and we should see the increased value in the UI.

When we use the **setState()** method to update the state, we do not have to pass all the properties which are available in the component's state. We only pass the properties which we want to update. Try the code shown in the following screenshot:

The screenshot shows a code editor on the left and a browser window on the right. The code editor displays the file `App.js` with the following content:

```
1 import React from "react";
2 import classes from "./styles.module.css";
3
4 export default class App extends React.Component {
5   state = {
6     currentCounter: 7
7   };
8
9   onIncrementClick = () => {
10     const updatedValue = this.state.currentCounter + 1;
11     this.setState({ currentCounter: updatedValue });
12   };
13
14   render() {
15     return (
16       <main className={classes.MainContainer}>
17         <h1>{`Counter: ${this.state.currentCounter}`}</h1>
18         <button onClick={this.onIncrementClick}>Increment Counter</button>
19       </main>
20     );
21   }
22 }
23
24
25
```

The browser window shows a simple application titled "Counter: 10". Below the title is a button labeled "Increment Counter".

Figure 3.22: Passing arguments in the setState method

As you can see, now, when you click on the button, it will show the increased counter.

Let us add some more functionality to this component. Say you wanted to toggle the visibility of the counter on a button click. Let us give it a try.

Before we get into how to use the state to show or hide the counter, first, let us understand what the logic should be to show or hide the counter. We have two options, either we can hide using JavaScript, or we can hide using CSS.

Let us try JavaScript first. If we simply add a condition as shown in the following code, then based on `showCounter`, either it will render null or it will render the `<h1>` tag to show counter.

```
!showCounter ? null : <h1>{`Counter:  
${this.state.currentCounter}`}</h1>
```

We can store the `showCounter` in the component's state. We can add a button that on click can update the value to either true or false to toggle the visibility of the counter. Refer to the following code shown for this implementation:

The screenshot shows a code editor on the left displaying the contents of `App.js`. The code defines a `React.Component` named `App` with state and methods for incrementing the counter and toggling its visibility. To the right, a browser window displays a simple application titled "Counter: 7". It has two buttons: "Increment Counter" and "Show/Hide Counter".

```

4  export default class App extends React.Component {
5    state = {
6      currentCounter: 7,
7      showCounter: true
8    };
9
10   onIncrementClick = () => {
11     const updatedValue = this.state.currentCounter + 1;
12     this.setState({ currentCounter: updatedValue });
13   };
14
15   onToggleCounterVisibility = () => {
16     this.setState({ showCounter: !this.state.showCounter });
17   };
18
19   render() {
20     return (
21       <main className={classes.MainContainer}>
22         {this.state.showCounter ? null : (
23           <h1>[Counter: ${this.state.currentCounter}]</h1>
24         )}
25         <button onClick={this.onIncrementClick}>Increment Counter</button>
26         <button onClick={this.onToggleCounterVisibility}>Show/Hide Counter</button>
27       </main>
28     );
29   }

```

Figure 3.23: Button to toggle visibility of counter

Let us go through the changes one by one. We added the `showCounter` property in the state, and we are using it in Line 22 to conditionally decide if we have to show the counter or not. We added a “**Show/Hide Counter**” button to toggle the visibility of the counter. We have added an `onClick` event, and in its handler, we negate the `showCounter` value to change the visibility and update the state. Once the state is updated, it calls the `render()` method with the updated values.

Now, just one very important thing to remember. The `setState()` method is asynchronous. So, when you call the method, your state might take some time to update if you have a very complex and deep component tree. And why is it being told to you here? Because we are using the current state to update the next state. So, it is possible that you might not get the latest values, so to avoid any unexpected data issues, the `setState()` method also gives you access to the previous state and props; you can use those to update the next state as shown in the following code snippet:

```

setState((state, props) => {
  showBlogs: !state.showBlogs
})

```

If we print something in the console inside the `render` method, it should get printed in the console every time we click the “**Show/Hide Counter**” button and also when the page loads for the first time because that is when the component is created. Always remember this `render()` method is called when the component is created or updated.

So, let us give it a try.

As you can see in the following image, “**Inside Render**” gets printed in the console multiple times because the component was created and the `render()` method was called. The component is updated when the user clicks on the button, which again calls the `render()` method.

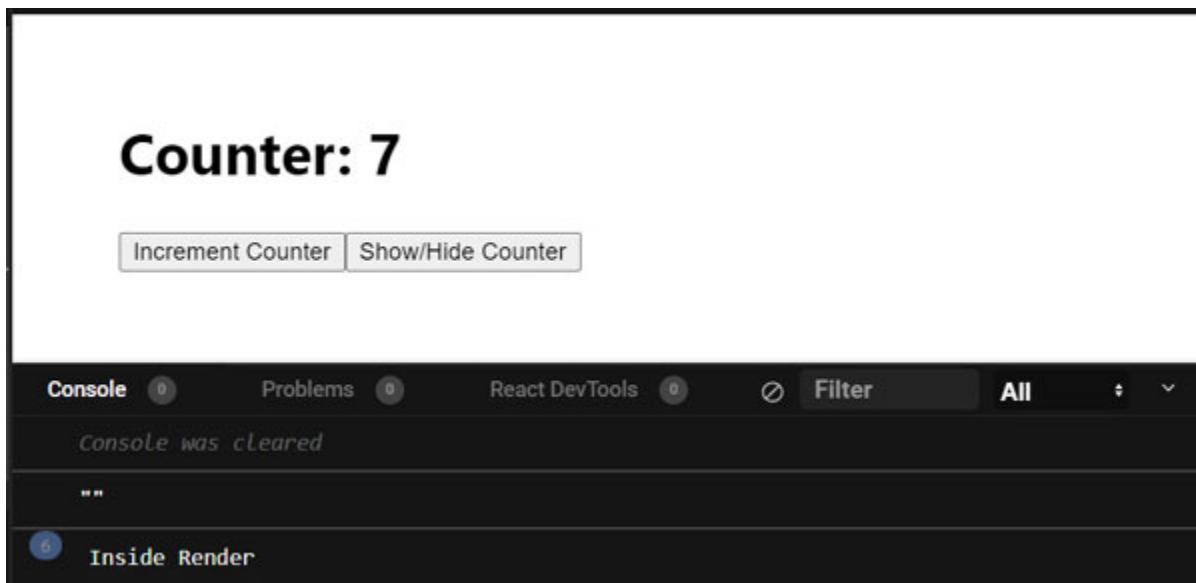


Figure 3.24: Render method calls

We can also change the button label conditionally. Say when the counter is visible, the label can be “**Hide Counter**” and when the counter is hidden, the label can be “**Show Counter**”. We will add it conditionally, as shown in the following screenshot:

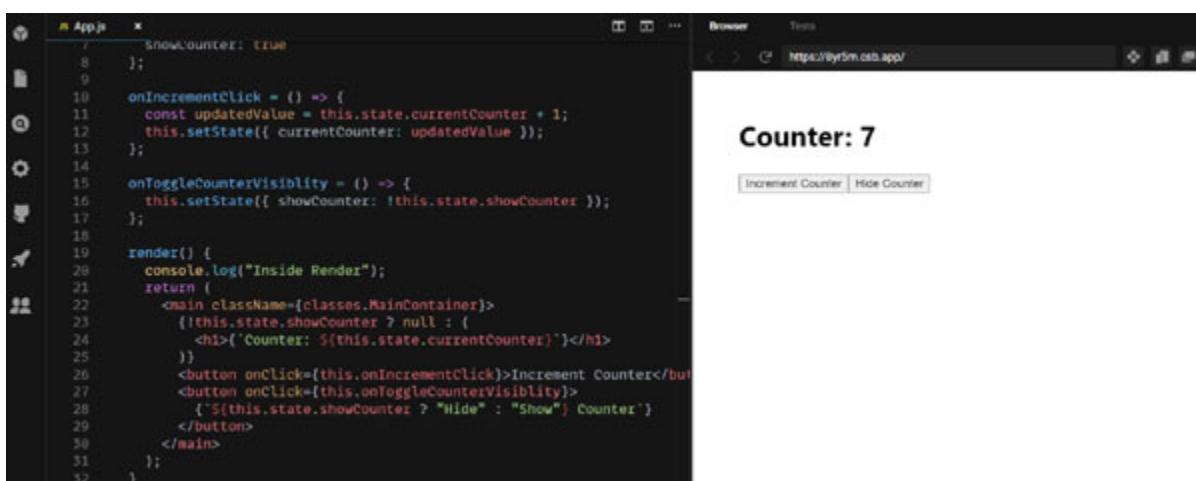


Figure 3.25: Change button label conditionally

Please refer to [figure 3.26](#) to change the button label conditionally:

The screenshot shows a code editor with `App.js` open. The code defines a class component with methods for incrementing the counter and toggling its visibility. It also includes a conditional render logic for the button label. To the right, a browser window displays a simple application with two buttons: "Increment Counter" and "Show Counter".

```

7     showCounter: true
8   ];
9
10  onIncrementClick = () => {
11    const updatedValue = this.state.currentCounter + 1;
12    this.setState({ currentCounter: updatedValue });
13  };
14
15  onToggleCounterVisibility = () => {
16    this.setState({ showCounter: !this.state.showCounter });
17  };
18
19  render() {
20    console.log("Inside Render");
21    return (
22      <main className={classes.MainContainer}>
23        {this.state.showCounter ? null : (
24          <h1>{'Counter: ' + this.state.currentCounter}</h1>
25        )}
26        <button onClick={this.onIncrementClick}>Increment Counter</button>
27        <button onClick={this.onToggleCounterVisibility}>
28          {'S' + this.state.showCounter ? 'Hide' : 'Show'} Counter
29        </button>
30      </main>
31    );
32  }

```

Figure 3.26: Change button label conditionally

Open the following link to try a working example of the counter app:

<https://codesandbox.io/s/stateful-counter-8yr5m?file=/src/App.js>

Passing props to components

Going back to our product listing app. As you can see in the following image, we have a new component, **Topbar**.

The screenshot shows a code editor with `App.js` open. It includes imports for `ClothingCard`, `Topbar`, and `AppData`. The `App` component uses a `Topbar` component. To the right, a browser window displays a product listing page titled "ApparelStore". It shows two products: "HRX by Hrithik Roshan Solid Sporty Jacket" and "Flying Machine Solid Bomber Jacket".

```

1 import classes from "./styles.module.css";
2
3 import ClothingCard from "./ProductCard/ProductCard";
4 import Topbar from "./Topbar/Topbar";
5 import { PRODUCT_DATA } from "./utils/AppData";
6
7 export default function App() {
8   const data = [...PRODUCT_DATA];
9
10  return (
11    <div>
12      <Topbar />
13      <main className={classes.MainContainer}>
14        {data.map((item, key) => <ClothingCard
15          </main>
16        </div>
17      );
18    }
19  )
20}

```

Figure 3.27: Topbar in the product list app

Let us say we want to add a wishlist functionality to this application. We want to add an icon button on the product card for adding the product to the user's wishlist. When this button is clicked, it should get highlighted, which

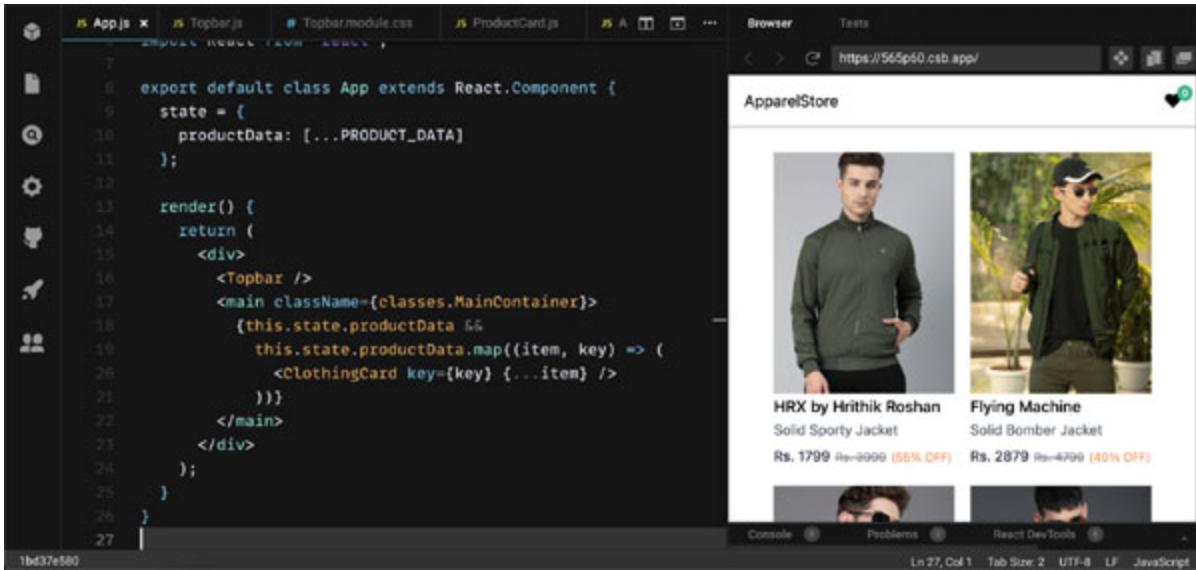
would mean that the product is added to the wishlist, and the wishlist count should increase on the topbar. We will not store the data in DB or send network requests. We just want to store it in the state, which is temporary.

Now, we have two options to handle the wishlist status for the products. Either we can manage it in the `<ProductCard />` component, or we can handle it in the parent component where we are rendering the products grid, which in our case is the `<App />` component.

Handling it in the `<ProductCard />` component will work absolutely fine, but then you will have four components that manage their own state. Say you had 100 product cards, then all these 100 cards would be handling their own state, and it would just add to the complexity of your application.

So, a better way is to handle the wishlist status in the parent component.

Currently, our `<App />` component is a functional component so let us convert it to a class-based component, as shown in the following screenshot:

A screenshot of a code editor and a browser window. The code editor on the left shows the file 'App.js' with the following code:

```
7
8 export default class App extends React.Component {
9   state = {
10     productData: [...PRODUCT_DATA]
11   };
12
13   render() {
14     return (
15       <div>
16         <Topbar />
17         <main className={classes.MainContainer}>
18           {this.state.productData.map((item, key) => (
19             <ClothingCard key={key} {...item} />
20           ))}
21         </main>
22       </div>
23     );
24   }
25 }
26
27 }
```

The browser window on the right displays a product grid for 'Appare&Store'. It shows two items: 'HRX by Hrithik Roshan Solid Sporty Jacket' and 'Flying Machine Solid Bomber Jacket'. Both items have a price of Rs. 1799/- and a discounted price of Rs. 8999/-, indicating a 50% off discount. The Flying Machine jacket also has a note '(40% OFF)'.

Figure 3.28: App component refactored to class-based component

Now, pay very close attention here.

The array items in the state variable `productData` represent the product card, so having the wishlist status info here would make sense.

Let us make it all `false` by default so that when we click on “**Add to wishlist**”, it updates to `true` for that specific card.

The screenshot shows a code editor with several tabs open: Topbar.js, Topbar.module.css, ProductCard.js, and AppData.js. The AppData.js tab is active, displaying the following code:

```
1 export const PRODUCT_DATA = [
2   {
3     thumbnail:
4       "https://assets.myntassets.com/h_720,q_90,w_540/v1/assets/images/categories/HRX_by_Hrithik_Roshan/Solid_Sporty_Jacket.jpg",
5     brand: "HRX by Hrithik Roshan",
6     title: "Solid Sporty Jacket",
7     mrp: 3999,
8     discount: 55,
9     isWishlisted: false
10 },
11 {
12   thumbnail:
13     "https://assets.myntassets.com/h_720,q_90,w_540/v1/assets/images/categories/Flying_Machine/Solid_Bomber_Jacket.jpg",
14     brand: "Flying Machine",
15     title: "Solid Bomber Jacket",
16     mrp: 4799,
17     discount: 40,
18     isWishlisted: false
19 },
20 {
21   thumbnail:
22 }
```

To the right of the code editor is a browser window showing the 'ApparelStore' application. It displays two products: 'HRX by Hrithik Roshan Solid Sporty Jacket' and 'Flying Machine Solid Bomber Jacket'. Each product card includes a small thumbnail image, the brand name, product title, original price (mrp), discounted price, and the percentage off.

Figure 3.29: Adding default value for wishlist status in product list data

Now, let us add the wishlist button on the card, as shown in the following screenshot:

The screenshot shows a code editor with two tabs open: App.js and Topbar.js. The App.js tab is active, displaying the following code:

```
1 import React from "react";
2
3 import classes from "./Topbar.module.css";
4
5 class Topbar extends React.Component {
6   render() {
7     return (
8       <div className={classes.Topbar}>
9         <p>ApparelStore</p>
10
11         <div className={classes.WishlistCountWrapper}>
12           <p className={classes.WishlistCount}>0</p>
13           <i className="fas fa-heart"></i>
14         </div>
15       );
16     }
17   }
18
19   export default Topbar;
```

To the right of the code editor is a browser window showing the 'ApparelStore' application. The product cards now include a 'Wishlist' button, represented by a heart icon. The rest of the interface remains the same, displaying the two products and their details.

Figure 3.30: Add the wishlist button on the product card

You will need to write the JSX code to show the wishlist button, as shown in the following screenshot:

```

6 <div className={classes.ProductCard}>
7   <div className={classes.PreviewWrapper}>
8     <img
9       className={classes.ProductImage}
10      src={props.thumbnail}
11      alt={props.title}
12    />
13   <div className={classes.WishlistWrapper}>
14     <p>
15       className={classes.WishlistButton}
16       onClick={() => props.wishlistClick(props.pos)}
17     <{props.isWishlisted ? (
18       <i className="fas fa-heart"></i>
19       Wishlisted
20     ) : (
21       <i className="fas fa-heart"></i>
22       Wishlist
23     )}
24     </p>
25   </div>
26 </div>
27 </div>
28 </div>
29 </div>
30 </div>
31 </div>
32 </div>
33 </div>
34 </div>
35 </div>
36 </div>
37 </div>
38 </div>
39 </div>
40 </div>
41 </div>
42 </div>
43 </div>
44 </div>
45 </div>
46 </div>
47 </div>
48 </div>
49 </div>
50 </div>

```

Figure 3.31: JSX code to add wishlist button

You will also need to write the CSS code to style the wishlist button, as shown in the following screenshot:

```

18 .PreviewWrapper {
19   position: relative;
20 }
21
22 .WishlistWrapper {
23   display: none;
24   position: absolute;
25   bottom: 0;
26   background-color: #rgba(255, 255, 255, 0.8);
27   width: 100%;
28   text-align: center;
29 }
30
31 .WishlistButton {
32   font-size: 12px;
33   cursor: pointer;
34 }
35
36 .WishlistButton i {
37   margin-right: 4px;
38 }
39
40 .ProductCard:hover .WishlistWrapper {
41   display: block;
42 }

```

Figure 3.32: CSS code to style the wishlist button

We will also need an `onClick` listener. And `onClick`, we will need to update the wishlist status, which we will store in the `<App />` component state, as shown in the following screenshot:

The screenshot shows a code editor on the left and a browser window on the right. The code editor displays the `App.js` file, which contains a class-based component. The component's state includes `productData` and `totalWishlistCount`. The `onWishlistBtnClick` method is defined to handle button clicks. The browser window shows a product listing page for 'ApparelStore'. It features two main sections: 'HRX by Hrithik Roshan' and 'Flying Machine'. Each section displays a jacket image, the brand name, the product name, and its price. Below these sections are two smaller images of a man wearing sunglasses.

```
1 export default class App extends React.Component {
2   state = {
3     productData: [...PRODUCT_DATA]
4   };
5
6   onWishlistBtnClick = (pos) => {
7     const updatedList = [...this.state.productData];
8     const updatedData = updatedList[pos];
9     updatedData.isWishlisted = !updatedData.isWishlisted;
10    updatedList[pos] = { ...updatedData };
11    this.setState({ productData: updatedList });
12  };
13
14  render() {
15    return (
16      <div>
17        <Topbar />
18        <main className={classes.MainContainer}>
19          {this.state.productData.map((item, key) => (
20            <ClothingCard
21              key={key}
22              pos={key}
23              {...item}
24            >
25          ))}
26        </main>
27      </div>
28    );
29  }
30}
31
32
```

Figure 3.33: Handle wishlist button click event

We also need to show this count at the topbar. We can simply pass the `productData` list as a prop to the `Topbar` and calculate the total products in the wishlist, or we can manage a separate state variable in the `<App />` component, which can be passed to the `<Topbar />` to simply render the count.

We are going to follow the second approach, as shown in the following screenshot:

The screenshot shows a code editor on the left and a browser window on the right. The code editor displays the `App.js` file, which now includes a `totalWishlistCount` state variable. The `onWishlistBtnClick` method is modified to update this count. The browser window shows the same product listing page as Figure 3.33, but the topbar now displays a count of '1' in the wishlist icon, indicating that one item has been added to the wishlist.

```
1 export default class App extends React.Component {
2   state = {
3     productData: [...PRODUCT_DATA],
4     totalWishlistCount: 0
5   };
6
7   onWishlistBtnClick = (pos) => {
8     const updatedList = [...this.state.productData];
9     const updatedData = updatedList[pos];
10    let updatedWishlistCount = this.state.totalWishlistCount;
11
12    if (updatedData.isWishlisted) {
13      updatedData.isWishlisted = false;
14      updatedWishlistCount--;
15    } else {
16      updatedData.isWishlisted = true;
17      updatedWishlistCount++;
18    }
19    updatedList[pos] = { ...updatedData };
20    this.setState({
21      productData: updatedList,
22      totalWishlistCount: updatedWishlistCount
23    });
24  };
25
26
```

Figure 3.34: Handling the total wishlist count

Additional info to remember: How do you add multiple classes using CSS modules? Normally, you would just write the following code shown:

```
<div class="class1 class2"></div>
```

But, with CSS modules, it is a bit different. If you remember, the CSS module file exports an object that matches the classes with hashed class names, which would basically be a string when we do `classes.something` it gives us a String value. To have multiple classes, we can add them to an array and join the array by space, as shown in the following code:

```
<div className=[classes.Class1, classes.Class2].join(" ")>  
</div>
```

Conclusion

We have two types of components—class-based and functional components. We use the state to manage data inside the components. We use props to pass data from a parent component to a child component.

Questions

- Q1. What is the difference between class-based and functional components?
- Q2. What is the difference between state and props?
- Q3. Can you pass functions from a parent component to a child component? If yes, then how?
- Q4. Can you pass data from a child component to a parent component? If yes, then how?
- Q5. How do you add multiple classes to an element when using CSS modules?

Multiple choice questions with answers

Q1. Which keyword do you use to make a component available in other components?

- a. export
- b. default

- c. import
- d. const

Q2. Which keyword do you use to make a component available inside a component?

- a. export
- b. default
- c. import
- d. const

Q3. Based on what we have learned so far, which type of component supports the state?

- a. class-based
- b. function-based

Q4. Do you use the state to pass data from the parent component to the child component?

- a. True
- b. False

Q5. Which type of export supports importing a module by any name?

- a. named
- b. default

Answers

Question	Correct Answer
Q1	a. export
Q2	c. import
Q3	a. class-based
Q4	b. False
Q5	b. default

CHAPTER 4

Lifecycle of Components

In this chapter, we will understand what happens when the `render()` method is called and how each component undergoes different stages of its lifecycle. We will learn how to use these lifecycle stages to implement functionalities in class-based components.

Structure

In this chapter, we will discuss the following topics:

- Major lifecycle stages: creation, updation, and deletion
- Lifecycle methods in class-based components
- Introduction to virtual DOM
- Browser DOM versus virtual DOM
- How virtual DOM helps React with performance
- Implementing the lifecycle methods

Objectives

After studying this chapter, you will be able to understand the different stages of a component's lifecycle and the callback methods available to override the default behavior of these lifecycle methods. You should also understand what happens behind the scenes when a component is created and updated.

DOM versus virtual DOM

We all know about the browser DOM. It stands for Document Object Model, and it represents our Web app in a tree-like structure. You must have learned it while learning JavaScript. As you can see in [figure 4.1](#), we have a DOM tree diagram for a simple table.

Figure 4.1: Browser DOM

Every time something changes in our application, the browser DOM gets updated to reflect those changes in the browser. This is all well and good, but the problem arises when we have to frequently update the DOM. It affects the performance by making our application slow.

Whenever a DOM element is updated, its children nodes also need to be re-rendered. This re-rendering or re-painting of the UI elements is what makes the whole process slow and very expensive on memory. Therefore, the more UI elements you have, the more expensive the DOM updates will be because the browser will need to re-render more nodes.

This is where React brings virtual DOM to solve the previously-mentioned performance issue.

As the name suggests, the virtual DOM is a virtual representation of the actual DOM. Just like the actual DOM, virtual DOM is a node tree that lists elements and their attributes and content as objects and properties.

Virtual DOM is a representation of the user interface. So, whenever the state of a component changes, a new virtual DOM is created. The new virtual DOM is compared with the previous one to find the updated nodes. If there are some nodes that need to be updated, then only those are updated in the browser DOM. If there is no difference between the old and the new virtual DOM, then the browser DOM is not updated. This makes the performance far better when compared to manipulating the browser DOM directly, as shown in [*figure 4.2*](#):

Figure 4.2: Virtual DOM

As shown in the preceding [*figure 4.2*](#), say an HTML element represented by Node-3 is updated. In Vanilla JS (Plain JavaScript), it will update the entire DOM tree for that Web page. In ReactJS, a new virtual DOM is created whenever the state is updated. This virtual DOM will be compared with the previous version of the virtual DOM to compute differences in the new virtual DOM so that only the required subtree is updated in the browser DOM. In the preceding scenario, only the subtree of Node-3 will be updated in the browser DOM.

You might be wondering why to update the entire subtree and why not just Node-3 in the browser DOM. Those optimizations are not required in most of the applications because this entire diffing process is really fast, but if our application is really complex and is facing some performance issues, then we get into these specific optimizations using something called Pure Components and `shouldComponentUpdate()`. We will learn about these in the upcoming chapters.

Can you guess when this process of creating new Virtual DOM starts?

Well,

If you remember, we have a method called `render()`, which is responsible for rendering the UI. When the component is created or updated, the `render()` method is called, which creates a new virtual DOM. This newly created virtual DOM is compared to the previous virtual DOM, and the UI is updated based on the differences. This process of comparing the old and new Virtual DOMs is called **diffing**.

I hope this gave you a better understanding of how the Virtual DOM works and how it improves your application performance.

Component lifecycles

Just like in the real world, everything single thing has a lifecycle—birth, growth, and death. Similarly, in the world of React, the components undergo these lifecycles. They are created, updated, and deleted. In this section, we will learn about the different lifecycles of React components.

There are following three stages in a component's lifecycle:

1. Mounting
2. Updating
3. Unmounting

There are different lifecycle methods that React provides for different stages of a component's lifecycle. React automatically calls the responsible method according to the stage in which the component is. These methods give us better control over our components, and we can manipulate them using these methods.

Mounting lifecycle methods

Mounting is the stage when the component is created and inserted in the browser DOM. This is the stage when the `render()` function is called for the first time. The following diagram shows different stages of the mounting lifecycle.

Figure 4.3: Lifecycle—mounting

The first step during the mounting of a component is initialization. This is where the component starts its journey by setting up the state and the props. The initialization process is done inside the constructor. Say your component is expecting props to set the initial state; you can do that inside the constructor as shown in the code snippet as follows:

```
class ProductCard extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      isWishlisted: this.props.wishlisted  
    }  
  }  
}
```

Due to the *class field declaration* introduced in ES6, the previous code snippet can also be written alternatively without the constructor, as shown as follows:

```
class ProductCard extends React.Component {  
  state = {  
    isWishlisted: this.props.wishlisted  
  }  
}
```

Once the component is initialized, the `render()` method is triggered, and the code written inside the `render()` method is executed. If more components are used inside the `render()` method, then those child components are rendered too.

When the current component, along with its child components, is rendered. React calls the final callback which is `componentDidMount()`. It just tells us

that the component is mounted in the DOM. This is the stage when we should make API requests to fetch data and update the component state if required.

Can you guess why it is recommended to make API calls in **componentDidMount()**?

Say you make an API request in the **constructor()** or **render()** method, and the API fails. You would want to show some error message in the component for the failed API request, but the problem is that your component might not be rendered at this point, and you will not be able to show the error to the user.

There is one more callback method, **getDerivedStateFromProps()**. This is called right before the **render()** method. It is not commonly used, and we will not talk about it in this book.

Now, let us write some code to check out the creation lifecycle methods. As shown in [figure 4.4](#), I have added logs to be printed in the console. When I load the application in the browser, it will mount the component, and you will see the sequence in which the lifecycle methods are triggered.

Figure 4.4: Lifecycle mounting—app component

As you can see in the image shown previously, we have added **console.log()** inside the **constructor()**, **componentDidMount()**, and **render()**. The sequence of the method written in the code is **constructor()**, **componentDidMount()**, and **render()**, but the sequence printed in the console on the right-hand side is “Inside constructor()”, “Inside render()”, and “Inside componentDidMount()”. This shows that the components follow the standard sequence of mounting lifecycle methods irrespective of the sequence of lifecycle methods written in the code.

Let us see what happens if we try to mount another component inside our app component. As you can see in [figure 4.5](#), it also triggers the mounting lifecycle of the nested component.

Figure 4.5: Lifecycle mounting with child components—app component

You can find the **Book** component structure in [figure 4.6](#) as follows:

Figure 4.6: Lifecycle mounting with child components—book component

As you can see in the preceding images, we have written the book component once, inside the `render()` method of the app component, which means every time we create the app component, it will create the book component. Just like any other component, when the book component is created, it will follow the mounting lifecycle methods. If you notice the sequence of logs in the console, then you will see that the `constructor()` and `render()` of the app component are triggered, followed by `constructor()`, `render()`, and `componentDidMount()` of the book component. Once all the elements and components inside the `render()` method of the app component are mounted, only then the `componentDidMount()` of the app component is triggered. Read this paragraph again, and this is an important concept to understand. You will face unexpected issues and difficulties in debugging if you do not understand the lifecycle methods.

Any guesses what will be printed in the console if we write the book component three times inside the `render()` method of the app component? Write your solution on a piece of paper before reading the answer.

Figure 4.7: Lifecycle mounting with multiple child components—app component

You can find the **Book** component structure in [figure 4.8](#) as follows:

Figure 4.8: Lifecycle mounting with multiple child components—book component

As you can see, the console result in the preceding screenshots shows that the first Book1 is initialized and rendered, followed by Book2 and Book3, and then `componentDidMount()` is triggered for them in the same sequence. This is not true universally, which means the result can differ based on how much time it takes for each component to render. If Book2 is taking more time to render than Book3, then Book2's `componentDidMount()` will be triggered after Book3's `componentDidMount()`. In other words, you do not have a way to tell what the sequence will be because it depends on the time taken for each component to render.

Open the following link to try a working example of the mounting lifecycle:

<https://codesandbox.io/s/creation-lifecycle-ntmouy>

That was all about the mounting or creation stage of a component's lifecycle. Let us talk about the updation stage.

Updating lifecycle methods

Updation is the stage when the component is re-rendered due to **state** and/or **props** changes. Hope you remember that a component can update its **state**, but it cannot update its **props**. The **props** can only be updated by the parent component, which is rendering your current component.

In other words, the updation cycle is triggered when you call **setState()** inside the component, or the props are updated by the parent component.

As you can see in [figure 4.9](#), there are three major lifecycle methods involved in the updation phase—**shouldComponentUpdate()**, **render()**, and **componentDidUpdate()**.

Figure 4.9: Lifecycle—updation

When the component is updated, the first callback method is **shouldComponentUpdate()**. This method returns a boolean value that decides whether the component should be updated or not. By default, it returns **true**, but we can conditionally return **true** or **false** to avoid any unnecessary re-renders. We will talk more about avoiding unnecessary re-renders in the “*Performance Optimization*” chapter.

If the **shouldComponentUpdate()** method returns **false** then the updation cycle is stopped.

If the **shouldComponentUpdate()** method returns true, only then the **render()** method is called. As you already know, this is when the browser DOM is actually updated. When the **render()** method is called, it also renders all the child components of that component.

When the current component, along with its child components, is updated. React calls the final callback which is **componentDidUpdate()**. It just tells us that the component is updated in the browser DOM. This is the stage when we can make API requests to fetch data from the backend.

There are two more callback methods in the update lifecycle **getDerivedStateFromProps()** and **getSnapshotBeforeUpdate()**. These are not commonly used, and we will not talk about them in this book.

Let us check these callback methods in our code shown in [*figure 4.10*](#).

Figure 4.10: Lifecycle updation—app component loaded

As you can see in the image shown previously, we have added **console.log()** inside the **shouldComponentUpdate()**, **componentDidUpdate()**, and **render()**. Remember that the components follow a standard sequence of updation lifecycle methods irrespective of the sequence of lifecycle methods written in the code.

If you notice, when the component is loaded, it is only showing “Inside render()”. It is not showing “Inside constructor()” and “Inside componentDidMount()” because we have removed those lifecycle methods from our code. It is not showing “Inside shouldComponentUpdate()” and “Inside componentDidUpdate()” because we have to trigger a component update for the updation lifecycle to start.

How do we trigger the updation cycle in this component? Write your answer on a piece of paper before reading further.

We will click on the button to trigger the updation lifecycle for this app component.

Figure 4.11: Lifecycle updation—button triggered

If you remember, we mentioned that the updation lifecycle will be stopped if the **shouldComponentUpdate()** method returns **false**. In the code shown in the preceding screenshot, we are returning **true**, so if we change it and return **false**, the component should not get updated.

Figure 4.12: Lifecycle updation—shouldComponentUpdate() return false

As shown in the image previously, when we click on the “Show/Hide button”, it starts the updation lifecycle. It triggers **shouldComponentUpdate()** method, which is why “Inside

`shouldComponentUpdate()`" is printed in the console. "Inside render()" is not printed after "Inside `shouldComponentUpdate()`," which means the updation cycle stopped at `shouldComponentUpdate()` because we returned false.

Open the following link to try a working example of the mounting lifecycle:
<https://codesandbox.io/s/component-update-uxrvvb?file=/src/App.js>

Unmounting lifecycle method

Unmounting is the stage when the component is unmounted or removed from the browser DOM. This happens automatically when the browser tab is closed or the route is changed, which results in the mounting of other components and unmounting of a current component or conditionally removing the component.

Here, we just have one callback method `componentWillUnmount()`. It is called right before the component is unmounted.

Figure 4.13: Lifecycle unmount

That was all about the mounting, updating, and unmounting stages of the component lifecycle.

Just a quick thing to remember, these lifecycle methods are only available in class-based components. We can also access lifecycle methods in function-based components using a feature provided by React called **Hooks**. We will talk about hooks in the upcoming hooks chapter.

Conclusion

There are three stages of a component's lifecycle—mounting, updating, and unmounting. React provides different callback methods to override the default behavior of these lifecycle methods. Virtual DOM is what makes React really fast; using the process of diffing improves performance by only updating the DOM nodes which are updated.

In the upcoming chapter, we will learn about Axios and its implementation to make network requests.

Questions

- Q1. What is virtual DOM? How does it differ from browser DOM?
- Q2. What are the different stages of a component lifecycle?
- Q3. What happens when you return false in shouldComponentUpdate()?
- Q4. When is a component unmounted?
- Q5. What is diffing process?

Multiple choice questions with answers

Q1. When the state is updated React directly updates the browser DOM?

- a. True
- b. False

Q2. componentWillUpdate() is a part of the creation lifecycle?

- a. True
- b. False

Q3. What happens if you call setState() in render() method?

- a. It throws an error
- b. It does not update the state
- c. It will update the state once
- d. It will update the state an infinite number of times

Q4. What happens when you return 100 from shouldComponentUpdate()?

- a. It throws an error
- b. It allows the update of component
- c. It does not allow the update of component
- d. It will update the component only when the state value is 100.

Q5. A component can only be updated once?

- a. True

b. False

Answers

Questio n	Correct Answer
Q1	b. false
Q2	b. false
Q3	d. It will update the state an infinite number of times
Q4	b. It allows the update of component
Q5	b. False

CHAPTER 5

Connecting to Backend

In this chapter, we will learn how to connect our React app to the backend using packages like Axios. We will learn how to make different API requests and handle their responses.

Structure

In this chapter, we will discuss the following topics:

- Different networking libraries and packages
- Intro to Axios
- GET, POST, PUT, and DELETE API requests and response
- Global Axios Setup
- Read and manipulate header and response data using Axios
- Scalable code architecture for network requests

Objectives

After studying this chapter, you will be able to understand HTTP, different request methods, and response codes. You will know how to make API requests using Axios. You will also know how to architect your application for scalability.

The purpose of backend

Generally, Web applications follow a 3-tier architecture where you have a frontend, backend, and a database, as shown in the following diagram:

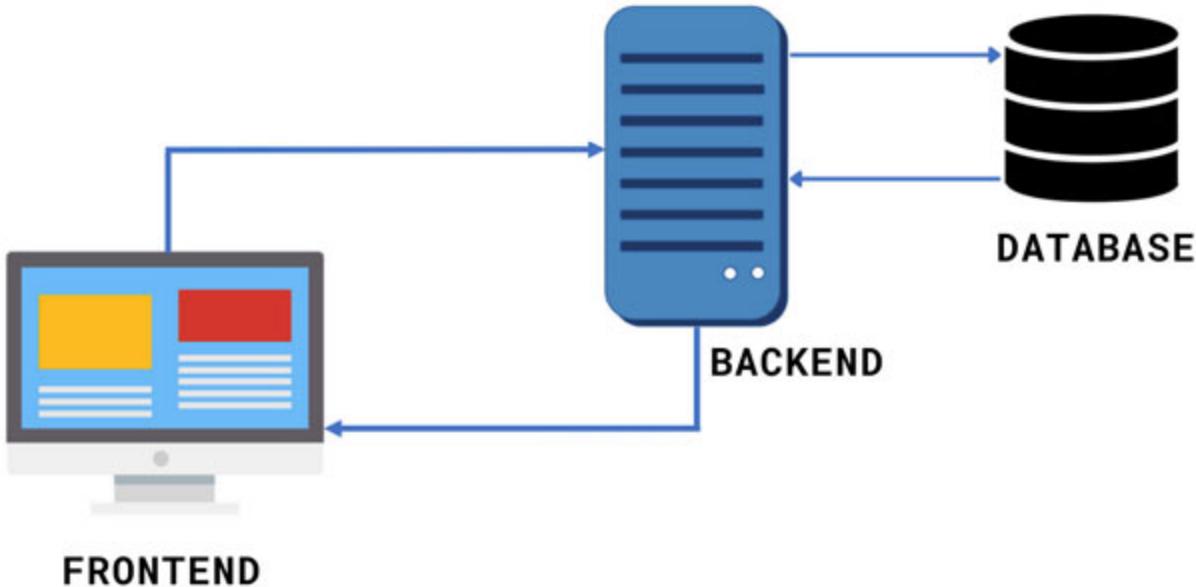


Figure 5.1: 3-Tier architecture

We have discussed the 3-tier architecture in [Chapter 1, Introduction to Web Development](#). Please refer to it for a quick brush-up. The frontend application connects with the backend for the exchange of data using REST APIs. Let us try to understand this data flow. Say your application wants to show a list of TODO items on the Web page. It sends an API request to the backend, saying, hey, I need the list of TODOs. The backend searches for this data in the database and responds to the request by sending the list to the Web app, as shown in [figure 5.2](#).

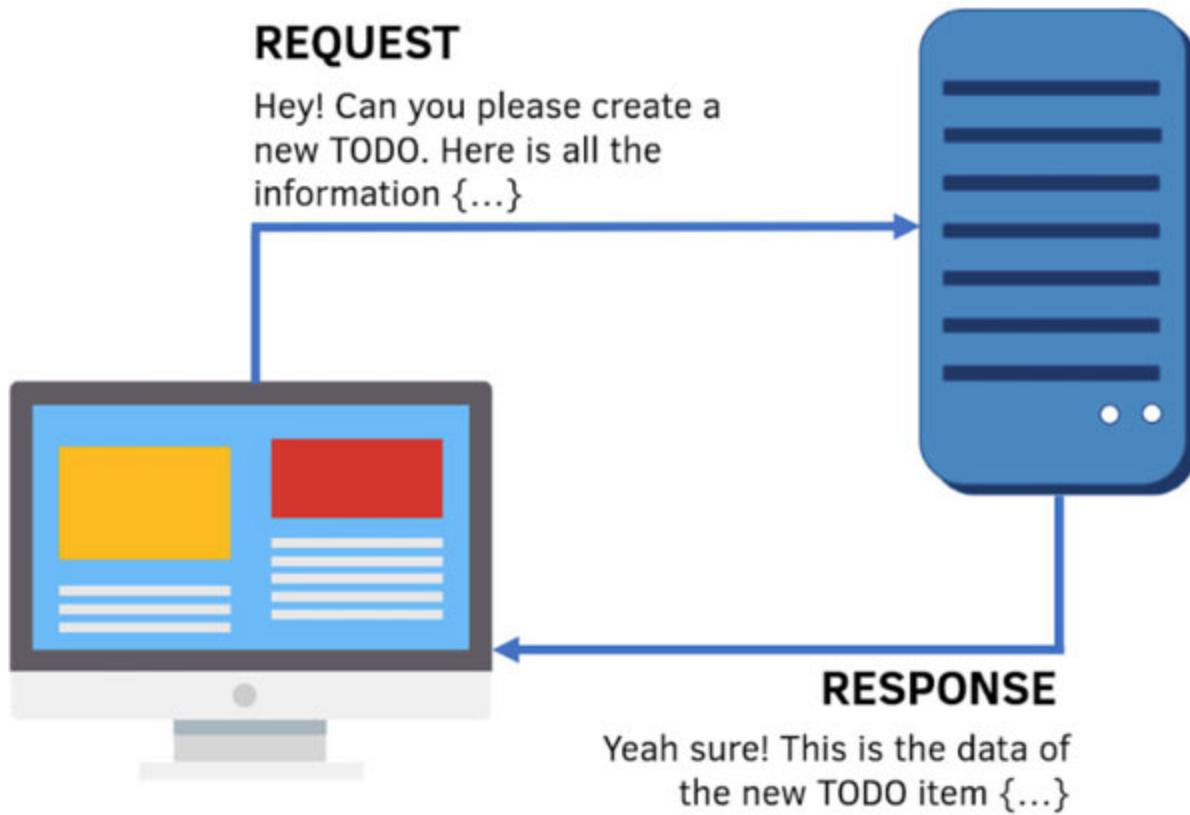


Figure 5.2: GET request

Similarly, say you wanted to create a new TODO item. For that, again, you send another API request to the backend along with some data saying, hey, I need to create a new TODO item; here is the TODO message. Can you please create it for me? The backend takes the data, creates a new entry in the database, and responds by returning some data about this newly created TODO item.

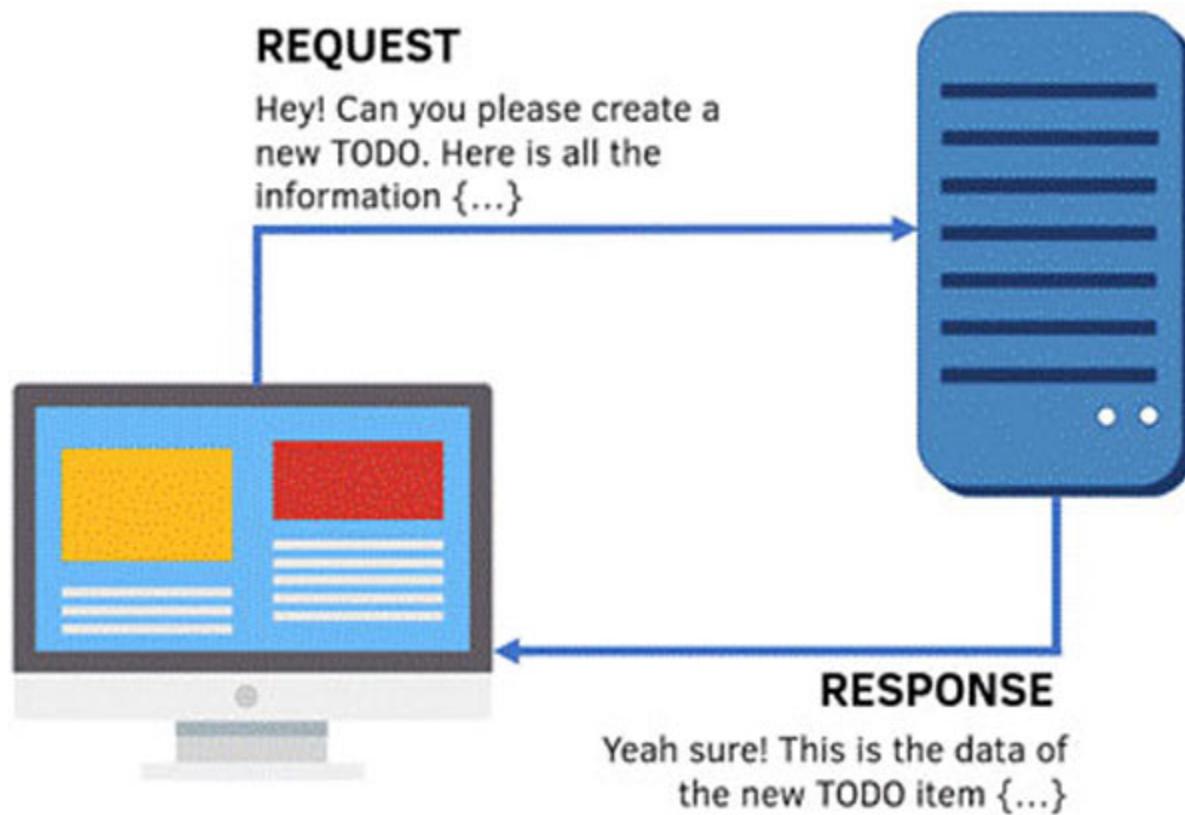


Figure 5.3: POST request

This is how the frontend interacts with the backend. The exchange of data is the most common use case where your frontend application (ReactJS) will connect to the backend.

To make API requests, we can use Fetch or Axios. Out of these two, Axios is more popular amongst react developers.

What is HTTP?

The requests are sent using HTTP. This HTTP stands for Hypertext Transfer Protocol. It is a protocol that allows the fetching of data from the server. It is the foundation for any data exchange on the Web. Clients and servers communicate by exchanging individual messages.

The messages sent by the client, in this case, our frontend application, are called requests, and the messages sent by the server as an answer are called responses.

API Endpoints

To send requests to your backend, you will use a URL that will point to your backend application and will tell your backend how to handle the request. This URL is called API Endpoint. Say, for example, here is a sample/dummy backend that will give you a TODO list—
<https://jsonplaceholder.typicode.com/todos>

The part of the URL before `/todos` points to your server. Now this second part, `/todos` tells your server to return a TODO list because a backend developer would have written some code to handle these URLs or endpoints. Just like on the frontend you handle different HTML pages for different URLs. Say `/index.html` loads `index.html` file, or `/contact.html` loads the contact HTML page. Similarly, the backend developers write some code to handle different endpoints.

HTTP methods

HTTP provides us with some methods for different kinds of requests. Here are the most commonly used methods:

- To get some data from the backend, you can use GET.
- To create new data entries on the backend, you can use POST.
- To update an existing data entry on the backend, you can use PUT.
- To delete an existing data entry on the backend, you can use DELETE.

There are a few other methods, but these four are the major ones that you will use very often, and we are going to learn how to trigger these requests using a network library called Axios.

HTTP response codes

The response from the backend server also includes a status code that tells whether the request was a success or failed. These are some common Response Status Codes:

- **200**: Represents “Success”, which means the request was successful.
- **400**: Represents a “Bad Request”, which means that the backend did not understand the request.

- **401**: Represents “*Unauthorized*”, which means that the user is not authorized to access the response data.
- **404**: Represents “*Not Found*”, which means that the URL is not found by the backend.
- **500**: Represents “*Something Went Wrong*” at the server.

There are many other HTTP response status codes, but these are the most commonly used.

Intro to Axios

Axios is a “*Promise-based HTTP client for the browser and node.js*”, which means using Axios, you can make XMLHttpRequests from the browser and HTTP requests from Node.js. In this book, we are only going to talk about XMLHttpRequests requests from the React application.

Before we can use Axios, we need to first install it in our project. Please follow the following steps to install Axios:

1. Open the terminal or command prompt.
2. Navigate to your project folder. Verify that you are at the location where the **package.json** file is located.
3. Run the following command, “**npm i axios**” and wait for the installation to be completed.
4. You can verify your installation by opening the **package.json** file. Axios must be inside the dependencies object, as shown in the following screenshot:

```
9   "main": "src/index.js",
10  "dependencies": {
11    "axios": "0.26.1",
12    "react": "18.0.0",
13    "react-dom": "18.0.0",
14    "react-scripts": "4.0.0"
15  },
```

Figure 5.4: Axios in package.json file

Making our first API request

Axios provides us with inbuilt functions to make API requests for different HTTP methods. Let us see how to make a GET call.

As you can see in the following screenshot, to make a GET request, we use the “**get()**” method provided by Axios. The **get()** method returns a promise, which, when successful, takes the control flow to **then()** method, and when the promise fails, then the control flow goes to the **catch()** method. Also, notice that the API request is triggered inside the **componentDidMount()** lifecycle method of class-based components.

When the API request is successful, the response object is received as an argument in the **then()** method. The response object holds the returned data, status code, status message (if required), request headers, and other configurations. To access the response data received from the backend, we read the “**data**” property of the response object.

The screenshot shows a code editor with two tabs: `App.js` and `Homepage.js`. In `App.js`, there is a `componentDidMount()` method that uses the `axios` library to make a GET request to `https://jsonplaceholder.typicode.com/todos/`. The response is logged to the console, showing an array of 200 items. The `render()` method returns an empty `<div>`.

```
10 componentDidMount() {
11   axios
12     .get("https://jsonplaceholder.typicode.com/todos/")
13     .then((response) => {
14       console.log(response);
15     })
16     .catch((error) => {
17       console.log(error.toString());
18     });
19 }
20
21 render() {
22   return <div className={classes.MainContainer}>{}</div>;
23 }
24
25
26 export default Homepage;
```

Figure 5.5: Axios GET request: response printed in the console

We are using a dummy backend called JSONPlaceholder. It gives us dummy API endpoints that can be used to mock backends. Let us use the response to render the TODO list on the homepage screen. We can simply iterate through the list in the `data` property to render the TODO cards, as shown in the following screenshot:

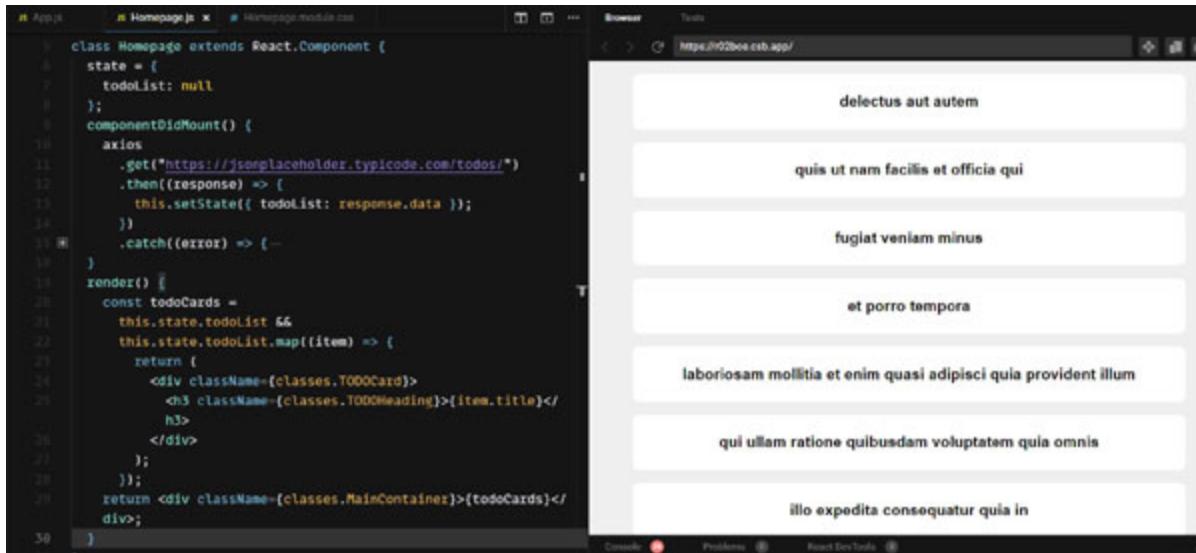
The screenshot shows the same code editor. The `componentDidMount()` method now iterates through the `response.data` array and creates a list of TODO cards. The `render()` method is removed. The console shows the generated TODO cards.

```
9
10 componentDidMount() {
11   axios
12     .get("https://jsonplaceholder.typicode.com/todos/")
13     .then((response) => {
14       console.log(response.data[0]);
15       const todoCards =
16         response.data.map((item) => {
17           return (
18             <div className={classes.TODOCard}>
19               <h3>{item.title}</h3>
20             </div>
21           );
22         });
23     })
24     .catch((error) => {
25       console.log(error.toString());
26     });
27 }
```

Figure 5.6: Axios GET request: generate TODO cards

Let us see the flow so far, and our application was loaded, the homepage component started the creation lifecycle, `render()` method was called, which rendered an empty `<div>`, then `componentDidMount()` was called, which sent an API request to the backend, we got a successful response, we iterated through the response data and created React elements for TODO Cards. Now, the next question is how do we render the TODO list we received?

We need to re-render, which means we need to call the `setState()` method. We can store the response list in the state. Once the state is updated, it will automatically call the `render()` method, and then we can iterate through the todo list from the state and create React elements for TODO Cards as shown in the following screenshot:

A screenshot of a development environment showing a code editor and a browser. The code editor on the left shows a file named 'Homepage.js' with the following content:

```
class Homepage extends React.Component {
  state = {
    todoList: null
  };
  componentDidMount() {
    axios
      .get("https://jsonplaceholder.typicode.com/todos/")
      .then((response) => {
        this.setState({ todoList: response.data });
      })
      .catch((error) => {
        console.log(error);
      });
  }
  render() {
    const todoCards =
      this.state.todoList === null
        ? null
        : this.state.todoList.map((item) => {
            return (
              <div className={classes.TODOCard}>
                <h3 className={classes.TODOHeading}>{item.title}</h3>
                <p>{item.description}</p>
              </div>
            );
          });
    return <div className={classes.MainContainer}>{todoCards}</div>;
  }
}
```

The browser window on the right shows the rendered output of the code, displaying a list of five TODO items with their titles and descriptions.

delectus aut autem
quis ut nam facilis et officia qui
fugiat veniam minus
et porro tempora
laboriosam mollitia et enim quasi adipisci quia provident illum
qui ullam ratione quibusdam voluptatem quia omnis
illo expedita consequatur quia in

Figure 5.7: Axios GET request: render TODO list

Open the following link to try a working example of a GET request:

<https://codesandbox.io/s/axios-get-r02boe>

POST request using Axios

Just like GET, Axios provides us with inbuilt functions to make POST API requests. Remember, POST requests are used to send creation requests to the backend, which means we need to send some data to the backend. Sending data to the backend brings in a small syntax change, “*we have to pass the data as an argument*”, as shown on Line 17 in [figure 5.8](#):

The screenshot shows a code editor with `App.js` open, displaying React code for creating a blog post using Axios. The browser window shows a simple form with fields for 'Enter Blog Title' and 'Enter Blog Description', and a 'Submit' button. The URL in the browser is `https://9ln7w0.csb.app/`.

```
1 export default class App extends React.Component {
2   onFormSubmit(e) {
3     e.preventDefault();
4
5     //I have hardcoded the data for simplicity but in real-app we
6     //will have to fetch that data from the form fields.
7
8     const data = {
9       title: "How to write clean React code?", 
10      description: "It's pretty simple - follow DRY"
11    };
12
13    axios
14      .post("https://jsonplaceholder.typicode.com/posts", data)
15      .then((response) => {
16        alert("Post created successfully!");
17      })
18      .catch((error) => {
19        alert("Post creation failed!");
20      });
21  }
22
23  render() {
24    return (
25      <div>
26        <form onSubmit={this.onFormSubmit} className="m...
27      </div>
28    );
29  }
}
```

Figure 5.8: Axios POST request

Open the following link to try a working example of a POST request:

<https://codesandbox.io/s/axios-post-8h7rl0>

PUT request using Axios

Axios provides us with inbuilt functions to make PUT API requests. It works exactly the same way as POST requests, and the only difference is that we use POST to create on the backend, but we use PUT to update data on the backend. To make a PUT request, we use the `put()` method, as shown on Line 17 in the following screenshot:

The screenshot shows a code editor with `App.js` open, displaying React code for updating a blog post using Axios. The browser window shows a simple form with fields for 'Enter Blog Title' and 'Enter Blog Description', and a 'Submit' button. The URL in the browser is `https://uvt369.csb.app/`.

```
1 export default class App extends React.Component {
2   onFormSubmit(e) {
3     e.preventDefault();
4
5     //I have hardcoded the data for simplicity but in real-app we
6     //will have to fetch that data from the form fields.
7
8     const data = {
9       title: "How to write clean React code? [Updated]", 
10      description: "It's pretty simple - follow DRY [Updated]"
11    };
12
13    axios
14      .put("https://jsonplaceholder.typicode.com/posts", data)
15      .then((response) => {
16        alert("Post updated successfully!");
17      })
18      .catch((error) => {
19        alert("Post updation failed!");
20      });
21  }
22
23  render() {
24    return (
25      <div>
26        <form onSubmit={this.onFormSubmit} className="m...
27      </div>
28    );
29  }
}
```

Figure 5.9: Axios PUT request

Open the following link to try a working example of a PUT request:

<https://codesandbox.io/s/axios-put-o0f589>

DELETE requests using Axios

Axios provides us with inbuilt functions to make DELETE API requests. Remember, DELETE requests are made to remove data entries on the backend, which means we need to tell the backend which data needs to be removed. Usually, we send a unique identifier either in the URL or in the body (it depends on how it is implemented on the backend) to tell the backend which entry needs to be deleted.

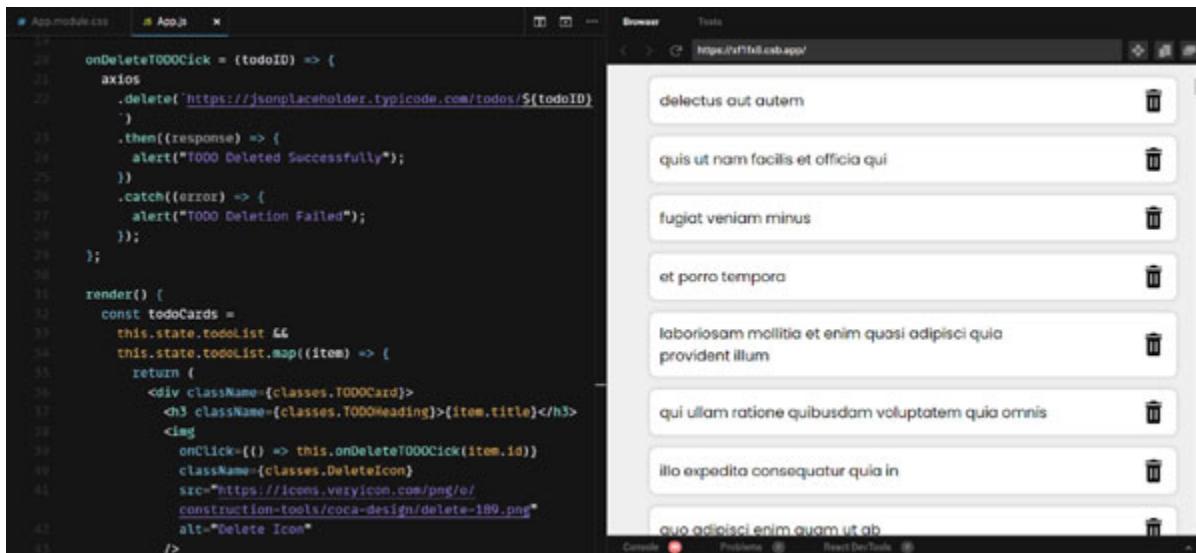


Figure 5.10: Axios DELETE request

Open the following link to try a working example of a PUT request:

<https://codesandbox.io/s/axios-delete-xf1fx8>

Global Axios setup

This is important when you are working with large applications. So far, we have worked with simple API calls where we do not have to add extra information such as headers, access tokens, timeout info, base URL, and so on.

When you work on large enterprise-level applications, you will usually have three applications for both frontend and backend—development, staging, and production. These are used at different stages of development. Development applications are used by the development team to develop new features. Staging applications are used for testing and demo purposes for business teams and other stakeholders. Production application is the finished and tested product that is used by the end-users.

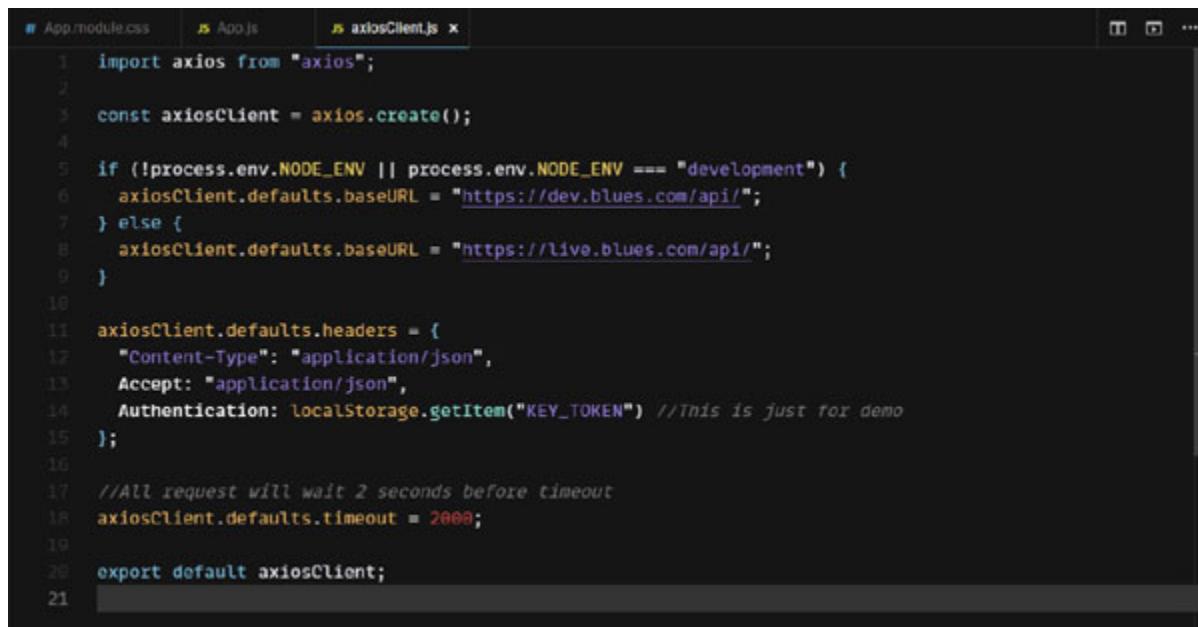
Our frontend application will have URLs for each backend. The same API will be available for different stages/versions of the application. For example:

Development: <https://dev.blues.com/api/users>

Staging: <https://staging.blues.com/api/users>

Production: <https://live.blues.com/api/users/>

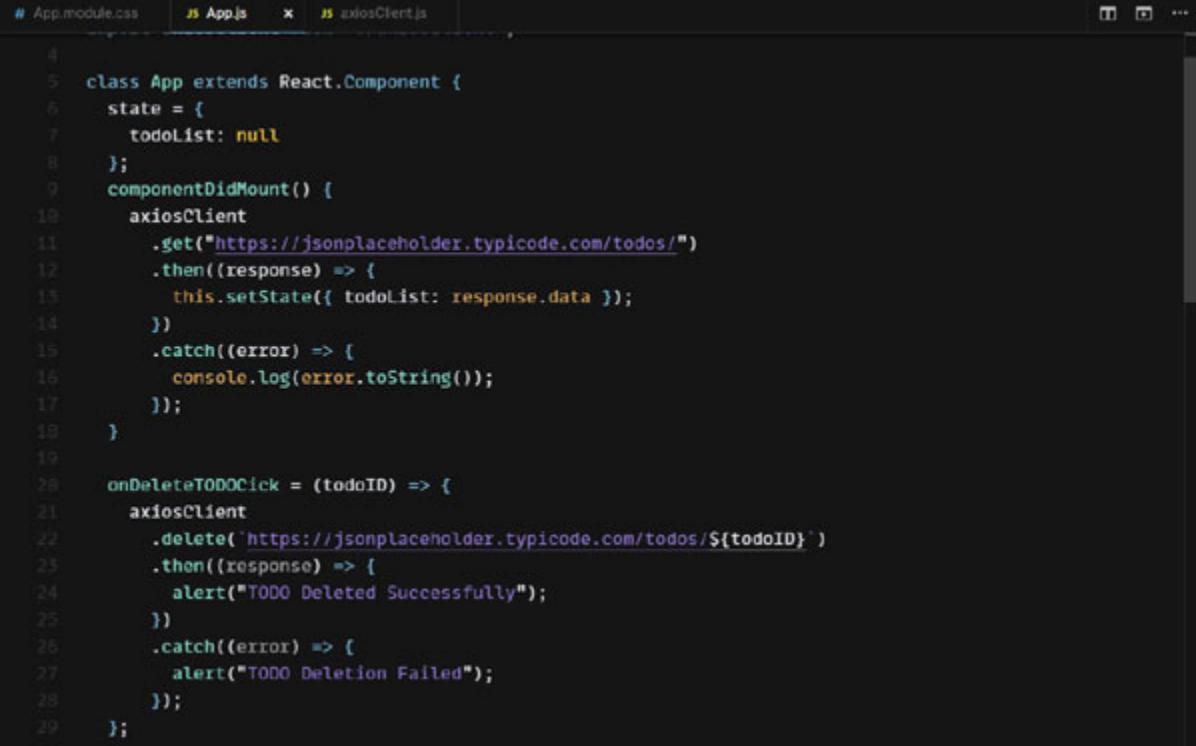
Notice how the subdomain differs for all three API endpoints. Imagine having to write conditions all over your application to select the right API endpoint based on the application version. This and similar issues can be resolved by creating a global instance of Axios, as shown in the following screenshot:



```
# App.module.css  # App.js  # axiosClient.js X
1 import axios from "axios";
2
3 const axiosClient = axios.create();
4
5 if (!process.env.NODE_ENV || process.env.NODE_ENV === "development") {
6   axiosClient.defaults.baseURL = "https://dev.blues.com/api/";
7 } else {
8   axiosClient.defaults.baseURL = "https://live.blues.com/api/";
9 }
10
11 axiosClient.defaults.headers = {
12   "Content-Type": "application/json",
13   Accept: "application/json",
14   Authentication: localStorage.getItem("KEY_TOKEN") //This is just for demo
15 };
16
17 //All request will wait 2 seconds before timeout
18 axiosClient.defaults.timeout = 2000;
19
20 export default axiosClient;
21
```

Figure 5.11: Axios global instance—creation

We can use this global instance as shown in the following screenshot:



The screenshot shows a code editor with three tabs: App.module.css, App.js, and axiosClient.js. The App.js tab is active and displays the following code:

```
4
5 class App extends React.Component {
6   state = {
7     todoList: null
8   };
9   componentDidMount() {
10     axiosClient
11       .get("https://jsonplaceholder.typicode.com/todos/")
12       .then((response) => {
13         this.setState({ todoList: response.data });
14       })
15       .catch((error) => {
16         console.log(error.toString());
17       });
18   }
19
20   onDeleteTodoClick = (todoID) => {
21     axiosClient
22       .delete(`https://jsonplaceholder.typicode.com/todos/${todoID}`)
23       .then((response) => {
24         alert("TODO Deleted Successfully");
25       })
26       .catch((error) => {
27         alert("TODO Deletion Failed");
28       });
29   };
}
```

Figure 5.12: Axios global instance—usage

The **axiosClient** will automatically select the right API point based on the development/production mode. It will automatically add the authentication token to the request, and it will also set the request timeout time to 2 seconds for all the API requests.

Intercept request and response

A request interceptor is used for adding extra info before the API request is sent to the backend. The interceptor for both request and response can also be added in the global Axios instance if they are required for all the API requests.

Check out the following code snippet for request intercept. Here we have added the **contentType** header before the request is made. Request interceptors are asynchronous by default, which might cause some delay in Axios request execution. To avoid this, we have used synchronous: true.

```
//request interceptor
axios.interceptors.request.use(function (request) {
  request.headers['Content-Type'] = 'multipart/form-data';
  return request;
}, null, { synchronous: true });
```

Figure 5.13: Axios intercept—request

Using response intercept, we can write or execute before the response reaches `then()` method. Response interceptors can be used to log out the user on token expiry (401 status), or you can refresh the token and make the failed request again or redirect to 404 or 500 pages based on the response status code.

```
//response interceptor
axios.interceptors.response.use(
  function (response) {
    //Dispatch any action on success
    return response;
  },
  function (error) {
    if (error.response.status === 401) {
      //Add Logic to
      //1. Redirect to login page or
      //2. Request refresh token
    } else if (error.response.status === 404) {
      //Add Logic to Redirect to 404 Not found page
    } else if (error.response.status === 500) {
      //Add Logic to Redirect to 500 Something went wrong page
    }
    return Promise.reject(error);
  }
);
```

Figure 5.14: Axios intercept—response

Scalable code architecture for network requests

So far, we have created simple applications and written all the network request-related code in the components themselves. Here is the list of changes we would make to write more maintainable and scalable code:

1. Define `BASE_URL`
2. Create a separate file for API Endpoints: Add all the URLs as constants and use those across your application. This way, you avoid spelling mistakes, and when you have to make a change, you will just have to update it in a single file.
3. **Write data parsers:** Say you are making an API call to get video details from the backend. You might receive more data than you require. Maybe from the backend, you are getting the following data: `VideoTitle`, `VideoDuration`, `VideoThumbnail`, `VideoDescription`, `VideoFileURL`, `Comments`, `ChannelInfo`, `RecommendedVideos`, and so on, but you might only need `VideoTitle`, `VideoDuration`, and `VideoThumbnail`. When you receive the data from the backend in your data parser, you will transform the data into smaller objects based on your requirement. Another benefit will be that even if the response keys change from the backend, then you will only have to update the data parser and not every component where you were using the data.

Conclusion

React does not provide an inbuilt mechanism to handle API requests. To make network requests, we use packages like Axios or fetch. Axios is the most popular way of making network requests used with React. It provides inbuilt methods to create the most common requests, such as GET, POST, PUT, and DELETE. We can create global instances to offload configuration details to a single file to avoid repetition.

In the upcoming chapter, we will learn about React Router package, which will be used to implement routing in React applications.

Questions

- Q1. What is HTTP? What are the different request types?
- Q2. What are the different response codes?
- Q3. How do you make a POST API using Axios? Explain.

Q4. Explain three scenarios when you might need to intercept the response.

Q5. Why do we need a global instance for Axios? How is it set up?

Multiple choice questions with answers

Q1. The GET API request used for?

- a. Reading the entries from the backend
- b. Creating new entries on the backend
- c. Updating an existing entry on the backend
- d. Deleting an existing entry on the backend

Q2. Which method is used for updating data at the backend?

- a. GET
- b. POST
- c. PUT
- d. DELETE

Q3. Which response code means the requested item was not found?

- a. 200
- b. 400
- c. 401
- d. 404

Q4. Which response code means the requested item cannot be accessed?

- a. 200
- b. 400
- c. 401
- d. 404

Q5. Which is the preferred package for network requests in React?

- a. Axios
- b. Fetch

- c. XMLHttpRequestObject
- d. None of the above

Answers

Questio n	Correct Answer
Q1	a. Reading the entries from the backend.
Q2	c. PUT
Q3	d. 404
Q4	c. 401
Q5	a. Axios

CHAPTER 6

React Hooks

In this chapter, we will learn about the new Hooks feature introduced in React 16.8. We will discuss about different hooks made available by the React library, and we will implement the most commonly used ones.

Structure

In this chapter, we will discuss the following topics:

- Introduction to Hooks
- Hooks provided by React
- Converting a class-based component to functional with Hooks
- `useEffect()` hook to handle Lifecycle methods
- Creating custom Hooks

Objectives

After studying this chapter, you will be able to understand the concept of hooks. You will also know how to implement state-related functionality and make API requests in functional components.

Introduction to Hooks

Hooks were introduced in React 16.8. They let you use state and allow making side effects inside a functional component.

Let us implement a simple counter using Hooks, as shown in the screenshot in [figure 6.1](#):

Figure 6.1: Simple counter app using Hooks

Let us go through the preceding code line-by-line:

- **Line-4:** We have used an inbuilt hook called `useState()`. This hook allows us to create state variables and methods to update those state variables. In the preceding example, we have created a counter as a state variable, and the `setCounter()` method will be used to update the counter value. We use destructuring syntax to create the state variable and the method to update it. We also pass an initial state value inside the `useState()` hook. The data type of this value can be anything based on what we need in our state variable.
- **Line-13:** We are using the state variable to render the counter value from the state variable.
- **Line-8:** We are using the counter update method “`setCounter()`” to update the state variable.

Notice that we do not have any “`this`” keyword in our code; we do not have `setState()` or the state property. If we required another state variable, let us say, “likes”, then we could simply create another using `useState()` as shown in the screenshot in [figure 6.2](#):

Figure 6.2: Multiple state variables

Converting a class-based component to functional with Hooks

Let us say we have implemented a functionality using a class-based component, and we want to convert them to lighter function-based components.

In the following screenshot, you can see a component written as a class-based component:

Figure 6.3: User list—class-based component

Following is the screenshot of the same component’s render method:

Figure 6.4: User list—class-based component render()

If you run the project and open the browser, you will see the following preview:

Figure 6.5: User listing page

As you can see in the preceding example, we have a component that sends a request to a dummy backend to get a list of users and then renders the user list.

The following are the steps to set up the preceding project in your local system:

1. Create a new react project in your local system using the following command “`npx create-react-app fetch-users`”.
2. Open the following URL in your browser:
<https://github.com/qaifikhān/rfjs-fetch-users-using-class-component/tree/main>.
3. Replace the content of App.js with the App.js version on the preceding URL.
4. Create a new file, “`App.module.css`”.
5. Replace the content of “`App.module.css`” with the version on the preceding URL.
6. Install axios and then run the project using the following command “`npm start`”.
7. Open the project in your browser using the following command “`http://localhost:3000/`”.

Let us convert this component to a function component using hooks.

There is one extra hook we are going to use, which is “`useEffect()`”. This hook is used to perform side effects. If you remember, from class-based components, we used lifecycle methods such as `componentDidMount()` and `componentDidUpdate()` to perform side-effects. In hooks, we have a single hook `useEffect()` to handle all lifecycle methods. We will talk about it in the coming sections. For now, just know that we use `useEffect()` to execute some code when the component is mounted, as shown in the following figure:

Figure 6.6: Trigger an API call

We will also need to remove the `render()` method with a return statement, as shown in the following screenshot:

Figure 6.7: No render method in functional component

This was a simple implementation using Hooks; hopefully, this gave you some clarity on how to use hooks to implement state and side-effects in a functional component. Just an additional information for you, hooks cannot be used inside a class-based component.

Hooks provided by React

The following is the list of some of the commonly used hooks provided by React:

- **useState**: This hook is used to add state-related functionality to functional components, as we saw in the previous examples.
- **useEffect**: This hook is used to add side-effects, as we saw in the fetch users example.
- **useContext**: This hook is used to read the context and subscribe to its changes. This is a hooks version of **ContextProvider**.
- **useRef**: This hook is used to create a Ref. We will learn about Refs in the upcoming [Chapter 8, Controlled and Uncontrolled Components](#).
- **useReducer**: This hook is related to Redux. We will learn about this once we learn Redux in [Chapter 9, Connecting Redux in React App](#).
- **useMemo**: This hook allows us to cache values, and we use it to return that memoized or cached value. We will learn about this in [Chapter 11, Performance Optimization](#).

We also have inbuilt hooks provided by the commonly used libraries that we use. React-router has exposed hooks to give us the option to manage the location, search parameters, URL parameters, history, and so on. React-redux has exposed hooks to manage the entire state management flow using

redux such as dispatch, select state variable, and so on. We will learn about these in their respective chapters.

As you can notice in the list of inbuilt hooks, their name always starts with “use”; think of it as a recommended practice. Remember this when you are creating your custom hooks. Now let us deep dive into two of the most important hooks—**useState()** and **useEffect()**.

useState() Hook

To add state to a functional component, we have to use a React hook called **useState**. It looks like this:

To be able to use this hook, we have to import the **useState** hook from React, as shown in the code snippet as follows:

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

We can create state variables and give them an initial value. The state variable is called **backgroundColor**, and **setBackgroundColor()** is the function for updating its value. To provide an initial value to state variables, we can pass it in the **useState()** method, as shown in the following code snippet:

```
const [ backgroundColor, setBackgroundColor ] =  
  useState("#fff");
```

If you are wondering about the syntax on the left side of the assignment operator, then remember what we learned in [Chapter 1, Introduction to Web Development](#). Let us do a quick recap. Let us say you have an array, and you want to access the first and second elements of the array. You can do that by writing the following code:

```
const mArr = [10, 20, 30, 40, 50]  
const [firstElem, secondElem] = mArr;  
console.log(firstElem); //This will print 10  
console.log(secondElem); //This will print 20
```

We are doing the same thing with **useState()**. It returns an array that contains a state variable and a method to update that state variable. We simply assign these values to a variable that we create.

How about another example to strengthen our understanding of the **useState()** hook? Let us say we want to create a component that contains a button. When the user clicks on this button, it randomly changes the

background color of the page. You can see the component code in the following screenshot :

Figure 6.8: Change background color application using Hooks

Let us understand the hook-related code.

In Line 5, we have created a state variable, “**backgroundColor**” and a method to update this state variable. We have also set “#fff” as the initial value to **backgroundColor**.

In Line 20, we are using the state variable to set the background color of the **MainContainer** div.

In Line 14, we are setting the randomly generated rgba string to the state variable. When we update the state, the component is re-rendered with the updated value of **backgroundColor**, and that is how we see a different background color.

You can try the preceding example at the following link:

<https://codesandbox.io/s/bold-wind-wn9eg9?file=/src/App.js>

If you want to add multiple state variables, then simply create them using multiple **useState()** hooks, as shown in the following code snippet:

```
const [ backgroundColor, setBackgroundColor ] =  
  useState("#fff");  
const [ counter, setCounter ] = useState(0);  
const [ isLoggedIn, setIsLoggedIn ] = useState(false);  
const [ userList, setUserList ] = useState([]);  
const [ videoData, setVideoData ] = useState({id: 1, title:  
  "Intro to React"});
```

Hopefully, this gives you some clarity on how the **useState()** hook can be used to add state in a functional component.

useEffect() Hook

As the name suggests, the **useEffect()** hook is used to carry out an effect each time there is a state change. By default, effects run after every completed render, which includes the first and any subsequent re-render, but you can choose to fire them only when certain values have changed.

Here is the syntax for creating effects using the `useEffect()` hook:

```
import { useEffect } from 'react';
useEffect(() => {
  //Callback function body
});
```

Let us try to simply print a message in the console that the effect was triggered. As mentioned earlier, that effect is triggered after every render; we should see a console statement on the page load. You might notice two console statements printed in the following screenshot, and this is because React does two initial renders in development mode but a single initial render in production mode. We will be talking about development and production mode in [Chapter 10, Build Tools](#).

Figure 6.9: Double renders in development mode

It should also be triggered after every render. So, let us add like and dislike counter state variables and trigger the state update on the button click, as shown in the subsequent screenshot:

Figure 6.10: A counter application using hooks

As you can see, the effect is triggered on page load because the app is rendered for the first time. But it also gets triggered when we click on either the “**Increase Like**” or “**Increase Dislike**” button because the component gets re-rendered.

The `useEffect()` hook also accepts a second argument, which is an array. This array is called **dependency array**. If the values inside the dependency array change only, then the effect is triggered after the initial render. The values inside the dependency array can be state variables or props, or they can even be empty.

Let us say we do not want to trigger the effect after the first render. In that case, we can leave the dependency array empty, as shown in the screenshot in [figure 6.11](#):

Figure 6.11: Empty dependency array

In Line 10, we have added a second argument to `useEffect()` hook, which is empty. We have also clicked on both the “**Increase Like**” and “**Increase Dislike**” buttons multiple times, but the console does not have any print statement after the initial render. This just proves that the effect was triggered on the initial render only.

If you notice, this behavior is kind of like that of lifecycle methods in class-based components. Can you guess which lifecycle method works like the preceding example? `componentDidMount()` is only triggered once when the component is rendered for the first time.

If you want to also trigger the effect after every render, then simply remove the second argument. This will trigger the effect both on the initial render as well as any subsequent state change. To trigger only on state change, we can store a flag that can tell us whether the component was initially rendered or was updated. If the component was initially rendered, then we can simply exit the effect without executing any further lines of code.

We can trigger effects only when a certain state variable or prop is updated. Let us say we want to trigger an effect on like count change, which will print the updated value of `likeCounter`. We also want to trigger an effect that will print the updated value of `dislikeCounter`. After making the changes, our code will look like as shown in the following screenshot:

Figure 6.12: Passing variables in the dependency array

On initial load, both the effects are triggered, but only `likeCounter` effect will be triggered when we update the `likeCounter` state variable, as shown in the following screenshot:

Figure 6.13: Multiple effects in a component

As you can also see in the preceding example, we can also have multiple effects inside a single component when we want to trigger different effects based on different state and/or prop changes.

We can use the `useEffect` hook to trigger API calls and make other side effects as we did in class-based components.

Let us try an example to make an API request using Hooks. To make a simple GET request to render a list of users, our code will look something like as shown in the following screenshot:

Figure 6.14: API request using Hooks

The preceding code triggers the API and renders the user list when the response is received. Just by looking at the UI, everything looks perfect, but there is an issue here. Can you go through the code once and find the issue?

The problem is that on the initial load, we trigger a request to the backend, and once the response is received, we update the state, which will trigger another render, which, in turn, will trigger an effect, and the loop will continue. We will make infinite API requests; you can see that if you open the networks tab of developer tools, as shown in the following screenshot:

Figure 6.15: Infinite API requests

So how do we fix it? Well, there are two ways we can fix the issue.

Solution One: We add a condition inside the effect to trigger API requests only when the users' state variable is empty, as shown in the following screenshot:

Figure 6.16: Infinite API request issue: Solution One

Solution Two: We add an empty dependency array; that way, the effect will only be triggered on component load, as shown in the following screenshot:

Figure 6.17: Infinite API request issue: Solution Two

You can play around with the fetch user list example on the link as follows:

<https://codesandbox.io/s/nice-parm-45i11w?file=/src/App.js>

Creating custom hooks

One of the biggest benefits of hooks is that we can extract common state-related functionalities into custom hooks. For example, sending API requests is one of the most common operations in an application. It requires state change and data management, which can be done in a class-based component, but because of hooks, we can create a custom component that will take care of API requests for us.

As you can see in the screenshot that follows, we are sending an API request to the backend:

Figure 6.18: API request to fetch users

We can create a new file and move the API request-related code into a custom hook, as shown in the following screenshot:

Figure 6.19: API requests using a custom hook

Just remember, when you are creating a new file for a custom hook, the name of the file should start with “use”. This is how React knows that the file contains a hook.

Now that our custom hook is ready, we can use it in the components to create API requests, as shown in the following screenshot:

Figure 6.20: Using a custom hook to make the API request

Notice that we removed the state variables for users and **showLoader**. On component load, we are calling the **useGETAPIRequest()** hook. The hook returns data, **isLoading**, and error, which we are using in the component. Notice that we are using the same names for the variables returned by the hook as they were inside the custom hooks file.

You can play with the previous custom hook example at the following link:

<https://codesandbox.io/s/competent-minsky-yd10sh?file=/src/App.js>

Conclusion

That was all about the hooks. We learned how to use hooks to add state-related functionality to functional components, trigger API calls from functional components, and create custom hooks to extract common state-related functionalities to reuse them.

In the upcoming chapter, we will learn about implementing routes in a React application using the react-router package.

Questions

- Q1. What are Hooks? When were they introduced?
- Q2. What is the reason for introducing hooks in React?
- Q3. What is the useState() hook?
- Q4. What is the useEffect() hook?
- Q5. How do you simulate componentDidMount() lifecycle using hooks?

Multiple choice questions with answers

Q1. Which hook is used to add a state in functional components?

- a. useEffect
- b. useState
- c. useRef
- d. useMemo

Q2. Which hook is used to send API requests?

- a. useEffect
- b. useState
- c. useRef
- d. useMemo

Q3. Which hook is used in the performance optimization of react components?

- a. useEffect
- b. useState
- c. useRef

d. useMemo

Q4. What is the second argument of useEffect called?

- a. Filter array
- b. Filter object
- c. Dependency array
- d. Dependency object

Q5. What should be the value of the second argument of useEffect to trigger effect only on component load?

- a. [props]
- b. [state]
- c. []
- d. None of the above

Answers

Question	Correct Answer
Q1	b. useState
Q2	a. useEffect
Q3	d. useMemo
Q4	c. Dependency array
Q5	c. []

CHAPTER 7

Routing in React Apps

In this chapter, we will learn how to load different pages on different URLs of our Web app. We will learn about React Router and how it can help us with handling different routes. We will see how to pass and read data from URLs. We will also learn how to handle exception pages such as 401, 404, and 500.

Structure

In this chapter, we will discuss the following topics:

- What is routing?
- Introduction to React-Router package
- React-Router setup in React app
- Handling dynamic URLs
- Conditional redirect
- Handling query params in URLs
- Handle 401, 404, and 500 pages

Objectives

After studying this chapter, you will be able to understand how to set up routes in a React application. You will be able to understand the code components provided by the react-router package. You will also know how to handle URL parameters, search parameters, and query parameters.

What is routing?

Routing is a process in which a user is directed to different pages based on their action or request. For example, on the YouTube homepage, you see a grid of video cards, and when you click on any of the cards, it redirects you to the video watch page. This redirection on click is called routing.

When you are building React applications, even if you build a simple application like a Blog App, even then you have to handle routing. You will have multiple screens in the application, such as the homepage, blog page, create/edit blog page, author page, and so on, and each page will have its own URL, which you will need to map with the right components.

Introduction to React-Router

React does not provide an inbuilt API to handle routing. That is why we use a package called react-router to handle routing.

Before we can use React-Router, we need to first install it in our project. Please follow the following steps to install:

1. Open the terminal or command prompt.
2. Navigate to your project folder. Verify that you are at the location where the **package.json** file is located.
3. Run the following command, “**npm i react-router react-router-dom**”, and wait for the installation to be completed.
4. The **react-router** is the package that is the heart of React-Router. It provides all the core functionalities for the **react-router-dom** package. While implementing routing in React, we do not import anything from the **react-router** package. Consider it more like a dependency for the **react-router-dom** package.
5. The **react-router-dom** package gives us access to different components such as **BrowserRouter**, **Routes**, **Route**, and so on. These components will be used to implement routing in our React project.
6. You can verify your installation by opening the **package.json** file. It must be inside the **dependencies** object, as illustrated in [**figure 7.1**](#):

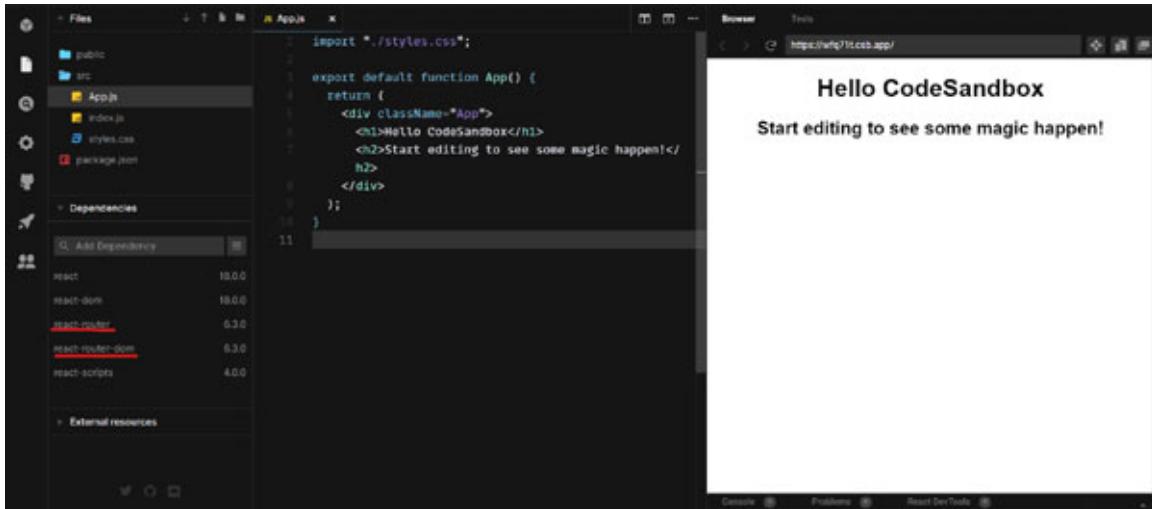


Figure 7.1: Opening the package.json file

[Figure 7.2](#) depicts verifying both package names:

```

{
  "name": "hooks",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^5.16.4",
    "@testing-library/react": "^13.1.1",
    "@testing-library/user-event": "^13.5.0",
    "react": "^18.0.0",
    "react-dom": "^18.0.0",
    "react-router-dom": "^6.3.0",
    "react-scripts": "5.0.1",
    "web-vitals": "^2.1.4"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app"
    ]
  }
}
  
```

Figure 7.2: Verify both package names

[React-Router setup in React app](#)

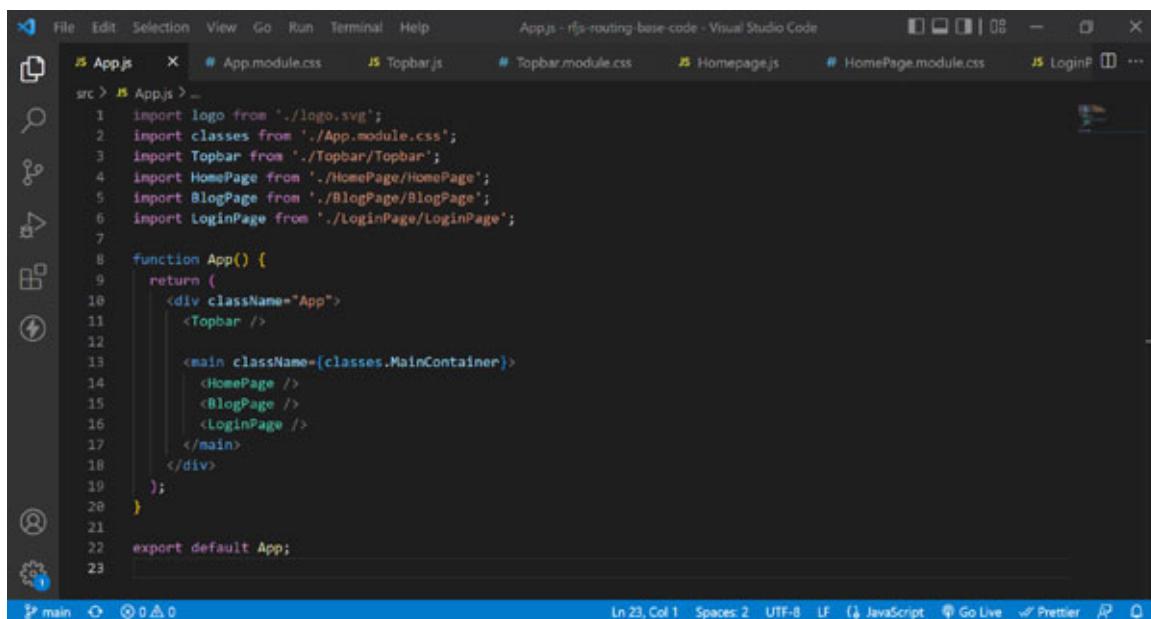
So far, we have installed the required packages, and now it is time to set up React Router to implement routing in our application. Here are a few things to remember to set up a simple route in your React application:

- Instead of an anchor tag, we use a component named `<Link>` provided by `react-router-dom`.

- To map components with URLs, we use a component named `<Route>` provided by react-router-dom.
- Wrap your entire app inside the `<BrowserRouter>` component provided by react-router-dom. The `<BrowserRouter>` uses HTML5 history API to manage the routing. It listens to URL changes and then handles the UI accordingly. It can either be used in an `index.js` or `app.js` file. We will learn how to use it in the upcoming section.

Let us try this by building a simple React blog application. Say we have a React app with three pages—home, blog, and login page. For now, we will not worry about the content on the pages. Follow these steps to set up the base application:

1. Create a new project using `create-react-app` on your local machine.
2. Run the following command, “`npm i react-router react-router-dom`”, and wait for the installation to be completed.
3. Create components for the Home, Blog, and Login pages.
4. Create another component for Topbar. In the topbar, add two menu items —“Home” and “Login”.
5. Your `App.js` should look like as illustrated in [figure 7.3](#):



The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Title Bar:** App.js - rjs-routing-base-code - Visual Studio Code.
- Code Editor:**

```

1  import logo from './logo.svg';
2  import classes from './App.module.css';
3  import Topbar from './Topbar/Topbar';
4  import HomePage from './HomePage/HomePage';
5  import BlogPage from './BlogPage/BlogPage';
6  import LoginPage from './LoginPage/LoginPage';

7
8  function App() {
9    return (
10      <div className="App">
11        <Topbar />
12
13        <main className={classes.MainContainer}>
14          <HomePage />
15          <BlogPage />
16          <LoginPage />
17        </main>
18      </div>
19    );
20  }
21
22  export default App;
23

```
- Bottom Status Bar:** Ln 23, Col 1 | Spaces: 2 | UTF-8 | LF | JavaScript | Go Live | Prettier | R | D

Figure 7.3: App component

6. Alternatively, you can download the preceding starter code from this link as follows:

<https://github.com/qaifikhan/React-for-job-seeker-routing-basecode>

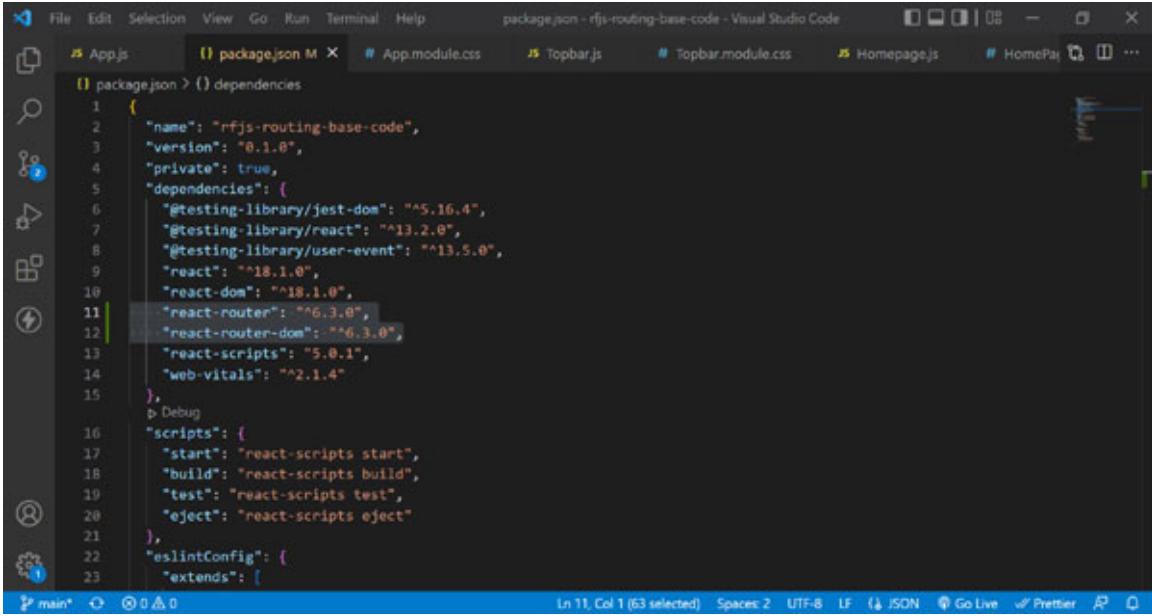
Once you download the code, follow the following steps to set it up on your laptop/desktop:

1. Open the command prompt and navigate to the downloaded folder.
2. Verify if you are in the same folder as the **package.json** file.
3. Run the following command, “**npm install**”. This will read and install your project’s dependencies from the **package.json** file.
4. Wait for dependencies to install.
5. Run the following command to start the application, “**npm start**”.
6. Open the following URL, “**http://localhost:3000/**”, to view the project in your browser.
7. You can see the screen in your browser, as illustrated in [figure 7.4](#):



Figure 7.4: Started code landing page

8. If you check the **package.json** file, you will notice that the routing-related packages are already installed, as illustrated in [figure 7.5](#):



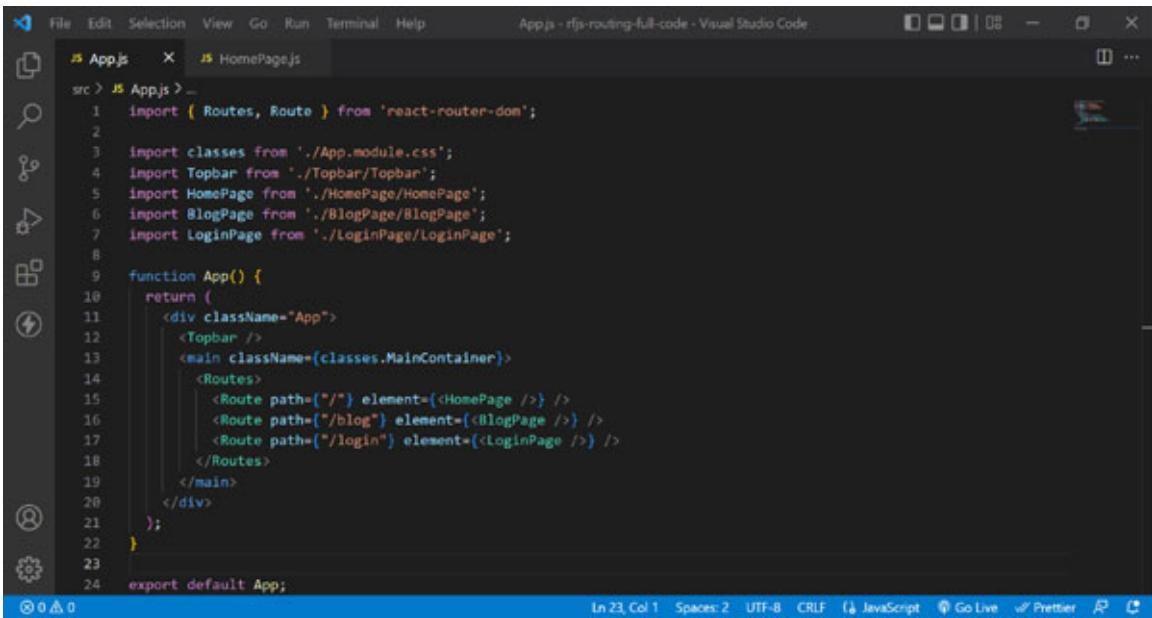
The screenshot shows the Visual Studio Code interface with the package.json file open. The file contains the following code:

```
1  {
2    "name": "rfjs-routing-base-code",
3    "version": "0.1.0",
4    "private": true,
5    "dependencies": {
6      "@testing-library/jest-dom": "^5.16.4",
7      "@testing-library/react": "^13.2.0",
8      "@testing-library/user-event": "^13.5.0",
9      "react": "^18.1.0",
10     "react-dom": "^18.1.0",
11     "react-router": "^6.3.0",
12     "react-router-dom": "^6.3.0",
13     "react-scripts": "5.0.1",
14     "web-vitals": "2.1.4"
15   },
16   "scripts": {
17     "start": "react-scripts start",
18     "build": "react-scripts build",
19     "test": "react-scripts test",
20     "eject": "react-scripts eject"
21   },
22   "eslintConfig": {
23     "extends": [

```

Figure 7.5: Verify installed packages

9. Right now, if you try to open these URLs in your browser **http://localhost:3000/home** or **http://localhost:3000/blog** or **http://localhost:3000/login**, they will all open the same homepage that we saw in the previous screenshot.
10. Now, we just need to set up the React Router to have Home, Blog, and Login on separate pages.
11. First, we need to use **<Routes>** and **<Route>** components to configure URLs and the corresponding components, as illustrated in [figure 7.6](#):



The screenshot shows the Visual Studio Code interface with the App.js file open. The file contains the following code:

```
1  import { Routes, Route } from 'react-router-dom';
2
3  import classes from './App.module.css';
4  import Topbar from './Topbar/Topbar';
5  import HomePage from './HomePage/HomePage';
6  import BlogPage from './BlogPage/BlogPage';
7  import LoginPage from './LoginPage/LoginPage';
8
9  function App() {
10  return (
11    <div className="App">
12      <Topbar />
13      <main className={classes.MainContainer}>
14        <Routes>
15          <Route path="/" element={<HomePage />} />
16          <Route path="/blog" element={<BlogPage />} />
17          <Route path="/login" element={<LoginPage />} />
18        </Routes>
19      </main>
20    </div>
21  );
22}
23
24 export default App;
```

Figure 7.6: Setup routes

12. **<Route>** element maps the component with the URLs. **<Routes>** act as a container/parent for all the individual routes that will be created in our app. Every time the URL is updated in the browser, the **<Routes>** component looks through all its children **<Route>** elements to find the correct match and renders the appropriate components. If you run the preceding code, then it will throw an error, as illustrated in [figure 7.7](#):

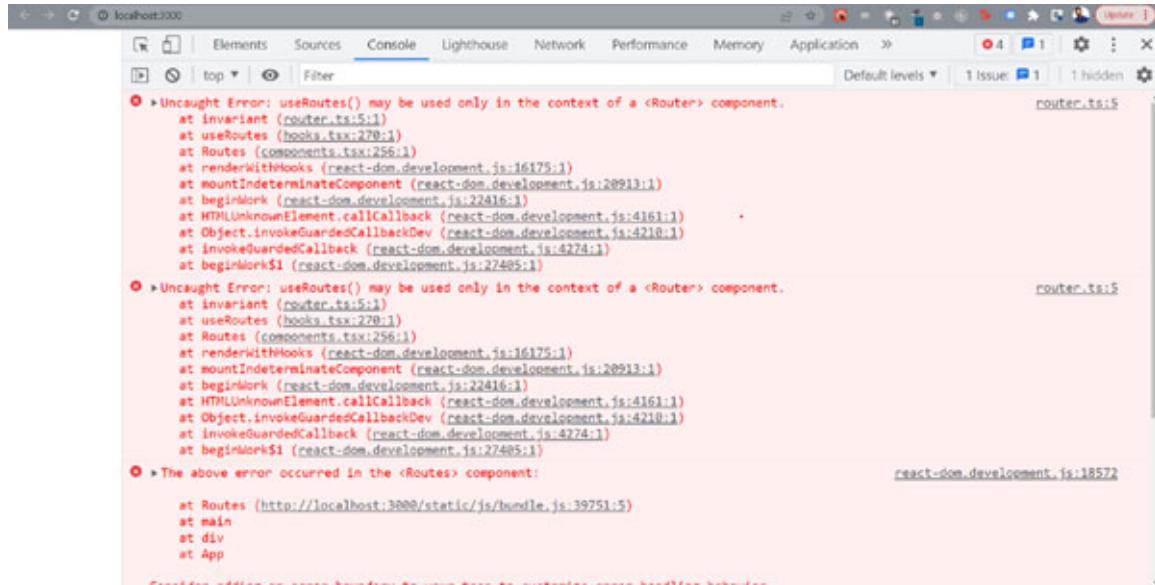
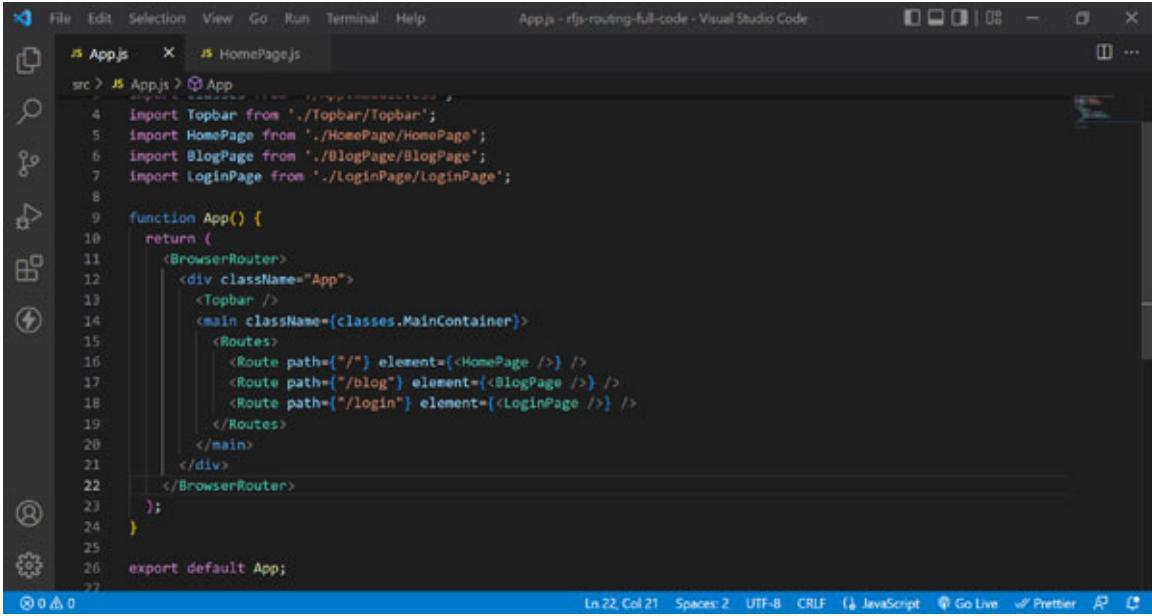


Figure 7.7: Missing browser router error

13. This error is received because we need to wrap our entire application inside **BrowserRouter**. The **BrowserRouter** uses the HTML Route API to facilitate routing in the browser. It stores the current location in the browser's address bar and navigates using the browser's built-in history stack. Either we can wrap the app in the **index.js** file, or we can do it in **app.js**, as illustrated in [figure 7.8](#):



```
App.js
src > App.js > App
4 import Topbar from './Topbar/Topbar';
5 import HomePage from './HomePage/HomePage';
6 import BlogPage from './BlogPage/BlogPage';
7 import LoginPage from './LoginPage/LoginPage';
8
9 function App() {
10   return (
11     <BrowserRouter>
12       <div className="App">
13         <Topbar />
14         <main className={classes.MainContainer}>
15           <Routes>
16             <Route path="/" element={<HomePage />} />
17             <Route path="/blog" element={<BlogPage />} />
18             <Route path="/login" element={<LoginPage />} />
19           </Routes>
20         </main>
21       </div>
22     </BrowserRouter>
23   );
24 }
25
26 export default App;
27
```

Ln 22, Col 21 Spaces: 2 UTF-8 CRLF JavaScript Go Live Prettier

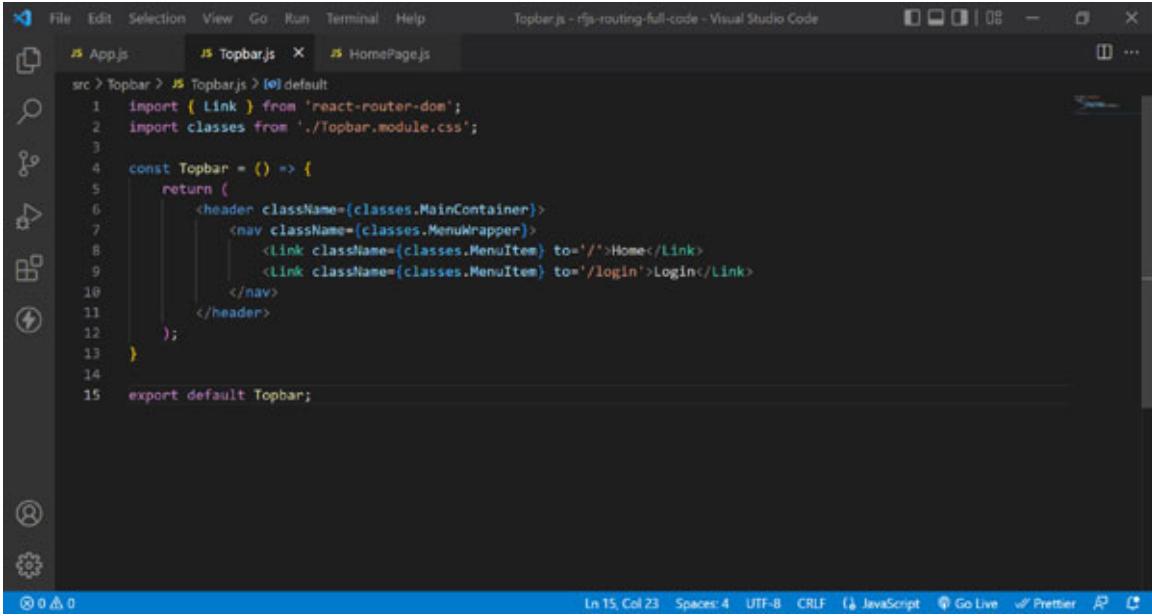
Figure 7.8: Wrap the entire application in the BrowserRouter component

14. Now that we have set up our **<Routes>**, it means when we change URLs in the browser, we will see appropriate components. If you enter **http://localhost:3000/login**, then it will load the **Login** component on the screen, as illustrated in [figure 7.9](#):



Figure 7.9: Login component loaded for “login” path

15. Similarly, if we change the URLs to **http://localhost:3000/**, then the home component will be loaded, and if we load **http://localhost:3000/blog**, then the Blog component will be loaded. But, users will not just type the URLs; they will click on hyperlinks to navigate to different pages. This means we need to setup **<Link>** to create hyperlinks, as illustrated in [figure 7.10](#):



The screenshot shows the Visual Studio Code interface with the 'Topbar.js' file open. The code defines a 'Topbar' component that renders a header with a menu. It uses 'Link' components from 'react-router-dom' to handle internal routing. The code is as follows:

```
src > Topbar > JS Topbar.js > (default)
1 import { Link } from 'react-router-dom';
2 import classes from './Topbar.module.css';
3
4 const Topbar = () => {
5   return (
6     <header className={classes.MainContainer}>
7       <nav className={classes.MenuWrapper}>
8         <Link className={classes.MenuItem} to="/">Home</Link>
9         <Link className={classes.MenuItem} to="/login">Login</Link>
10      </nav>
11    </header>
12  );
13}
14
15 export default Topbar;
```

Figure 7.10: Use Link components to add hyperlinks

16. Notice that we did not use `<a>`. The reason is that if we use `<a>`, then routing is handled by the browser, but we want to handle routing using React Router, which is why we will use `<Link>` whenever we create an internal hyperlink (navigate inside the React application), and we will use `<a>` when we create an external hyperlink (navigate to external websites such as Google or Facebook, and so on).

Thus, we have set up simple routing in our React application.

Handling dynamic URLs

Let us say we want to have a grid of blog cards on the homepage. When a user clicks on any of these cards, they are redirected to the blog details page. We will keep the blog cards simple, and they will have only two information—title and thumbnail, as illustrated in [figure 7.11](#):

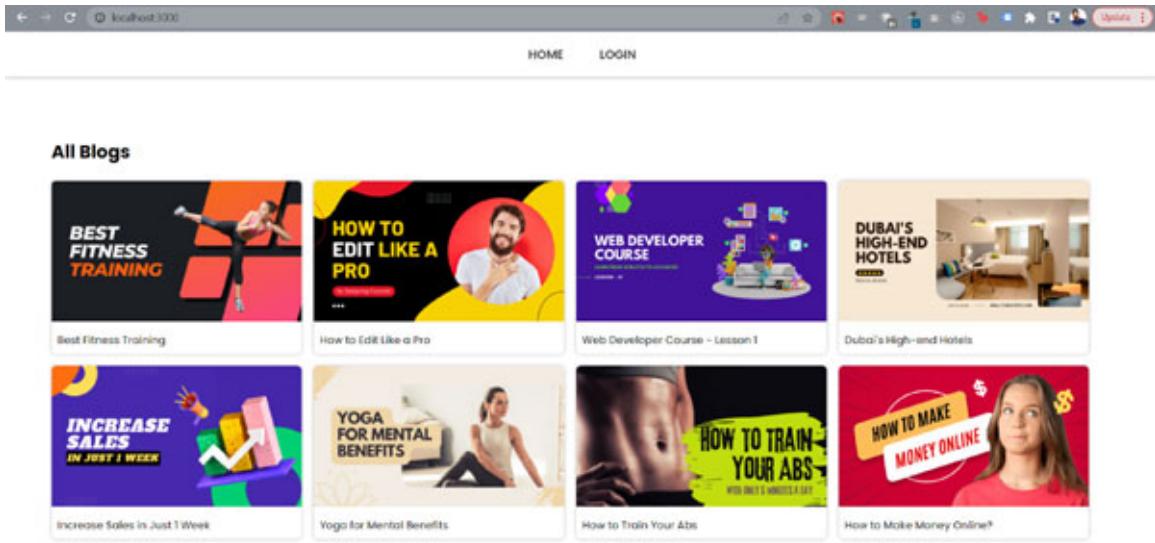


Figure 7.11: Video grid with basic info

You may download the data to populate blog cards from the following URL:

<https://github.com/qaifikhān/React-for-job-seeker-routing-fullcode/blob/main/src/utilities/AppData.js>

To create the grid, we just need to iterate through the preceding data and create the **BlogCard** component. You can design the **BlogCard** component as you want, or you can download the JSX and CSS for **BlogCard** from the following URL:

<https://github.com/qaifikhān/React-for-job-seeker-routing-fullcode/tree/main/src/BlogCard>

Now that we have our grid ready, we need to add the hyperlinks so that when a user clicks on these cards, they are redirected to the blog details page. The URLs should look like “/blog/id”, where “id” is the unique blog id. To implement this, we need to make the following changes to our code.

Update the “to” props for **<Link>** component from `to="#"` to `to={`/blog/${id}`}`, as shown in the following screenshot:

```
src > js App.js < js AppData.js
1 import { Routes, Route, BrowserRouter } from 'react-router-dom';
2
3 import classes from './App.module.css';
4 import Topbar from './Topbar/Topbar';
5 import HomePage from './HomePage/HomePage';
6 import BlogPage from './BlogPage/BlogPage';
7 import LoginPage from './LoginPage/LoginPage';
8
9 function App() {
10   return (
11     <BrowserRouter>
12       <div className="App">
13         <Topbar />
14         <main className={classes.MainContainer}>
15           <Routes>
16             <Route path="/" element={<HomePage />} />
17             <Route path={`/blog/:id`} element={<BlogPage />} />
18             <Route path="/login" element={<LoginPage />} />
19           </Routes>
20         </main>
21       </div>
22     </BrowserRouter>
23   );
24 }
```

Figure 7.12: Setting-up dynamic parameters

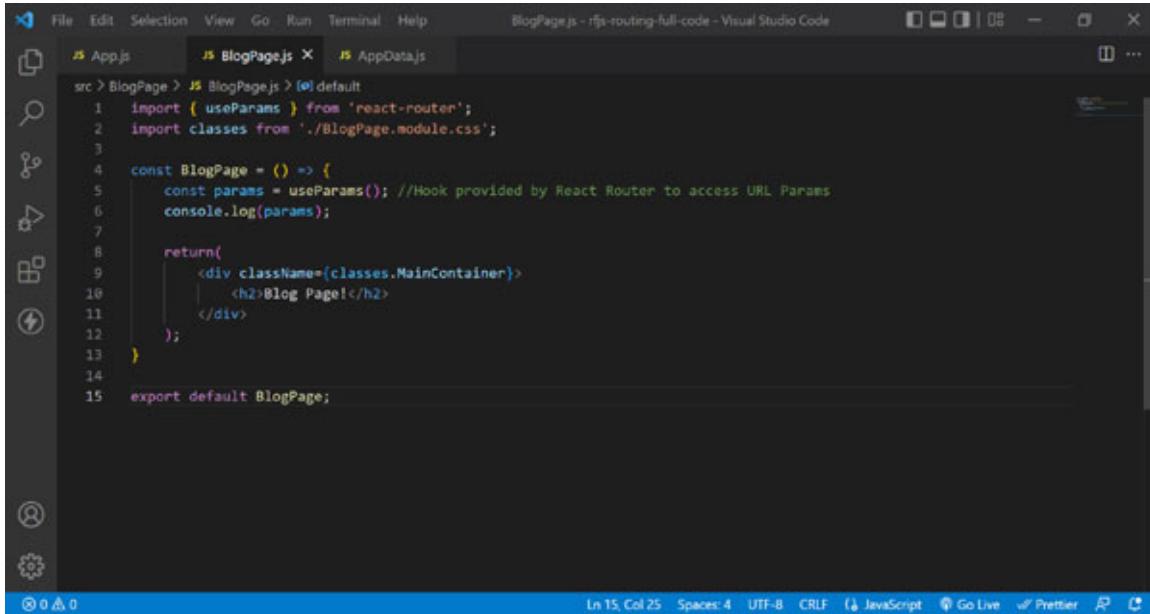
After this change, when a user clicks on the blog card, the URL will contain the id too, as shown in the following screenshot:

```
# Topbar.module.css      js HomePage.js      # HomePage.module.css      js BlogCard.js < # BlogCard.module.css      js AppData.js
src > BlogCard > js BlogCard.js < default
1 import React from 'react';
2 import { Link } from 'react-router-dom';
3
4 import classes from './BlogCard.module.css';
5
6 const BlogCard = ({id, thumbnail, title}) => {
7   return (
8     <div className={classes.BlogCard}>
9       <Link to={`/blog/${id}`}>
10         <img className={classes.BlogThumbnail} src={thumbnail} alt={title} />
11         <p className={classes.BlogTitle}>{title}</p>
12       </Link>
13     </div>
14   );
15 }
16
17 export default BlogCard;
```

Figure 7.13: Update BlogCard to handle redirect

We need to show the right data on the blog details page. Can you guess how will you know which blog you need to show on the details page? URL contains the unique ID for the blog to be loaded. We can access the URL params by

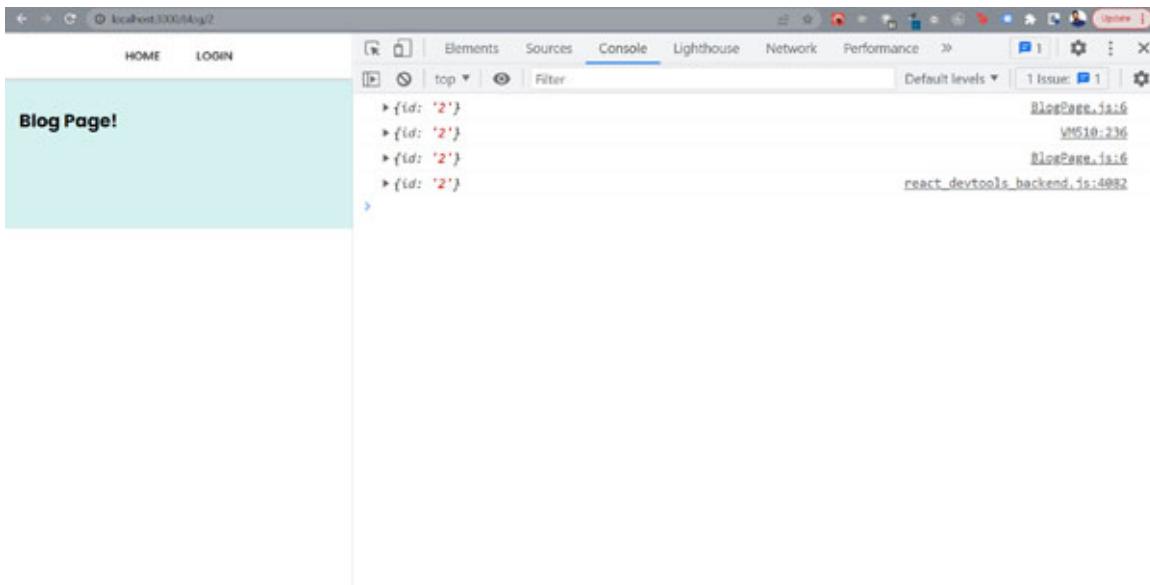
using the `useParams()` hook in the Blog details page as shown in the following:



```
File Edit Selection View Go Terminal Help
App.js  JS BlogPage.js  AppData.js
src > BlogPage > JS BlogPage.js > default
1 import { useParams } from 'react-router';
2 import classes from './BlogPage.module.css';
3
4 const BlogPage = () => {
5   const params = useParams(); //Hook provided by React Router to access URL Parameters
6   console.log(params);
7
8   return(
9     <div className={classes.MainContainer}>
10      <h2>Blog Page!</h2>
11    </div>
12  );
13}
14
15 export default BlogPage;
```

Figure 7.14: useParams() hook to access URL parameters

If you print the props in the `Blog` component, then you will see the URL params as shown in the screenshot that follows:

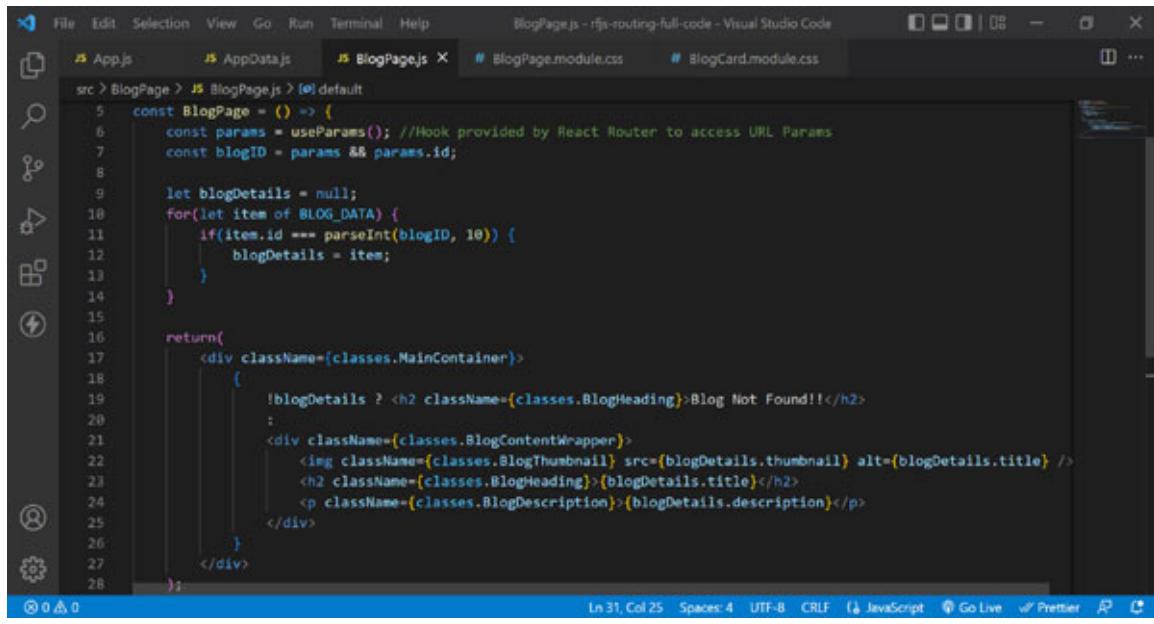


```
localhost:3001/blog/2
HOME LOGIN
Blog Page!
Console
Default levels 1 Issue: 1
BlogPage.js:6
BlogPage.js:6
BlogPage.js:6
react_devtools_backend.js:4082
```

Figure 7.15: Print URL parameters in the console

The `params` object contains the key with the name “`id`” because we have configured the Route path as `path={"/blog/:id"}`. If we change it to say

`path="/blog/:blogId"`, then we will get the following object from the `useParams()` hook: `{blogId: "2"}`. As shown in [figure 7.14](#), we can access the Params to use them for different purposes. For example, we can use the id provided by the URL params to fetch details about the current blog, as shown in the following screenshot:



The screenshot shows a Visual Studio Code interface with several tabs open. The active tab is `BlogPage.js`, which contains the following code:

```
src > BlogPage > BlogPage.js > default
5  const BlogPage = () => {
6    const params = useParams(); //Hook provided by React Router to access URL Params
7    const blogID = params && params.id;
8
9    let blogDetails = null;
10   for(let item of BLOG_DATA) {
11     if(item.id === parseInt(blogID, 10)) {
12       blogDetails = item;
13     }
14   }
15
16   return(
17     <div className={classes.MainContainer}>
18       {
19         !blogDetails ? <h2 className={classes.BlogHeading}>Blog Not Found!!</h2>
20         :
21         <div className={classes.BlogContentWrapper}>
22           <img className={classes.BlogThumbnail} src={blogDetails.thumbnail} alt={blogDetails.title} />
23           <h2 className={classes.BlogHeading}>{blogDetails.title}</h2>
24           <p className={classes.BlogDescription}>{blogDetails.description}</p>
25         </div>
26       }
27     </div>
28   );

```

The code uses the `useParams()` hook to extract the `blogID` from the URL. It then loops through a `BLOG_DATA` array to find the blog entry with the matching ID. Finally, it returns a `div` element containing either a `h2` for "Blog Not Found!!" or the blog's title and description.

Figure 7.16: Fetching blog details

Hooks provided by React Router

Earlier, we used the `useParams()` hook provided by React router. There are many more hooks provided by React Router, as shown in the following list. We will use the most common ones in this chapter and the upcoming chapters:

- `useLocation()`
- `useMatch()`
- `useNavigate()`
- `useParams()`
- `useSearchParams()`

Conditional redirect

Let us try to implement a simple functionality that, on the Login Menu Click, we redirect the user to the login page. On the login page, we will just have a

button to login into the application. We will implement forms in the upcoming chapter, [Chapter 8, “Controlled and Uncontrolled Components”](#). As shown in the following screenshot, when the user clicks on this button, then we will sign in the user automatically and store a flag that the user is logged in.



Figure 7.17: Login page

On the “Login” menu item click, we need to manage the login status flag; let us call it “`isUserLoggedIn`”. When the login menu item is clicked, we can set “`isUserLoggedIn`” to true in the state. Now, the question is whether we manage the “`isUserLoggedIn`” state in the `Topbar` component or somewhere else? Simply ask yourself this question, “*Does this value affect multiple components or just this one?*”. If the answer is that the value affects only one component, then manage the state in that component itself. If the value is used in multiple components, then manage it in the closest parent component. For example, let us say we want to add a condition that only logged-in users can view the blog details page, and when a logged-out user clicks on the blog card, they are redirected to the login page. This example clearly affects multiple components, so it needs to have the state in the nearest parent component (or you can use Redux, which we will learn in the upcoming chapter).

The login button in the `topbar` component will change the “`isUserLoggedIn`” value. This value will be required in the `BlogDetails` component to decide whether to open the blog details page or redirect to the `LoginPage` component. As `Topbar` and `BlogDetails` components have the `App` component as a common parent so we can manage the “`isUserLoggedIn`” state value in the `App` component and pass it down to the Topbar along with a function to update it when required, as shown in the following screenshot:

The screenshot shows the Visual Studio Code interface with the 'App.js' file open. The code handles the logged-in state of a user. It uses React's useState hook to manage the state and provides a function to update it. The component then renders a BrowserRouter, which contains a div with the class 'App'. Inside this div, there is a Topbar component and a main container. The main container holds a Routes component with three defined routes: one for the home page, one for blog posts, and one for the login page. The login page conditionally renders based on the user's logged-in status.

```
9
10    function App() {
11        const [isUserLoggedIn, setIsLoginStatus] = React.useState(false);
12
13        const updateLoggedInStatus = (status) => {
14            setIsLoginStatus(status);
15        }
16
17        return (
18            <BrowserRouter>
19                <div className="App">
20                    <Topbar isLoggedIn={isUserLoggedIn} setLoginStatus={updateLoggedInStatus} />
21                    <main className={classes.MainContainer}>
22                        <Routes>
23                            <Route path="/" element={<HomePage />} />
24                            <Route path="/blog/:id" element={<BlogPage isLoggedIn={isUserLoggedIn} />} />
25                            <Route path="/login" element={<LoginPage isLoggedIn={isUserLoggedIn} setLoginStatus={updateLoggedInStatus} />} />
26                        </Routes>
27                    </main>
28                </div>
29            </BrowserRouter>
30        );
31    }
32
```

Figure 7.18: Handling logged-in state in the App component

The state and function to update the state can be accessed in the **Topbar** component as shown in the following:

The screenshot shows the Visual Studio Code interface with the 'Topbar.js' file open. This component is a functional component that takes props. It returns a header element containing a nav component. The nav component has links for 'Home' and 'Logout'. The 'Logout' link is only present if the user is logged in, as indicated by the 'props.isUserLoggedIn' prop. The component also includes a 'Login' link. The code uses the 'Link' component from 'react-router-dom' and styles from 'Topbar.module.css'.

```
1  import { Link } from 'react-router-dom';
2  import classes from './Topbar.module.css';
3
4  const Topbar = (props) => {
5      return (
6          <header className={classes.MainContainer}>
7              <nav className={classes.MenuWrapper}>
8                  <Link className={classes.MenuItem} to='/'>Home</Link>
9                  {
10                      props.isUserLoggedIn ?
11                          <p className={classes.MenuItem} onClick={() => props.setLoginStatus(false)}>Logout</p>
12                          :
13                          <Link className={classes.MenuItem} to='/login'>Login</Link>
14                  }
15              </nav>
16          </header>
17      );
18  }
19
20  export default Topbar;
```

Figure 7.19: Using props to manage app state changes

As shown in [figure 7.20](#), we have access to the “**isUserLoggedIn**” state value in the **BlogDetails** component as a prop. We can add a condition to redirect to the **LoginPage** based on the logged-in status of the user, as shown in the following screenshot:

```
const BlogPage = (props) => {
  const params = useParams(); // Hook provided by React Router to access URL Params
  const blogID = params && params.id;

  let blogDetails = null;
  for(let item of BLOG_DATA) {
    if(item.id === parseInt(blogID, 10)) {
      blogDetails = item;
    }
  }

  const navigate = useNavigate();
  useEffect(() => {
    if (!props.isUserLoggedIn) {
      return navigate("/login");
    }
  }, [props.isUserLoggedIn]);

  return(
    <div className={classes.MainContainer}>
      {blogDetails ? <h2 className={classes.BlogHeading}>Blog Not Found!!</h2>
      :
      <h1>{blogDetails.title}</h1>
      <p>{blogDetails.content}</p>
      <button onClick={()=>navigate('/edit/' + blogID)}>Edit</button>
    </div>
  );
}
```

Figure 7.20: Conditional redirection

Similarly, you can conditionally navigate to any URL inside your application or even to an external URL.

Handling query params in URLs

You might have noticed that sometimes the URLs have a question mark and some ampersand symbols, as shown in the following examples:

<https://randomsite.com?search=hello>

<https://randomsite.com?fName=React&lName=Book>

These extra values in the URLs are called **search parameters**. It is pretty common to have search parameters in the URLs, and they hold simple data that needs to be passed from one page to another. For example, let us talk about search functionality. When the users type in the search box and hit enter or click on the search button, they are redirected to the search page. Now the question is how does the search page know what was searched by the user. Simply, we pass the search text in the search parameters, and on the search page, we can access the search parameter value to send a call to the backend to get the search results. We can use the inbuilt hook `useSearchParams()` provided by react-router to access the search parameters.

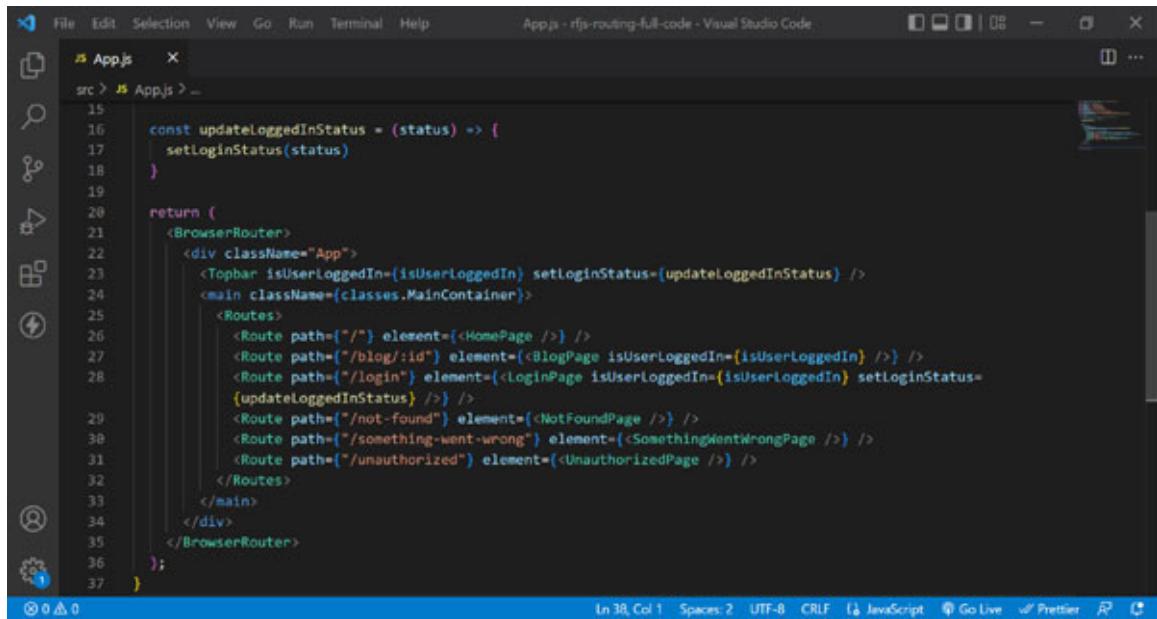
Let us try to build a simple search functionality. We will need to do the following:

1. Create a search page that will show the search string.
2. Add a form with an input box in the topbar to enable users to type the search string.
3. Listen to the form submit button to redirect to a search page.
4. Pass search parameters on redirection.
5. Read the search parameters on the search page and show them in UI.

We will implement this functionality when we learn how to implement forms in [Chapter: 8, “Controlled and Uncontrolled Components”](#).

Handling 401, 404, and 500 pages

For page not found (404), something went wrong (500), and unauthorized (401) pages, we need to create a different component for each and set up the route component in the **App** component, as shown in the following screenshot:



The screenshot shows the Visual Studio Code interface with the file 'App.js' open. The code defines a functional component 'App' that returns a 'BrowserRouter' component. Inside it, there is a 'div' with the class name 'App'. This div contains a 'Topbar' component with the prop 'isUserLoggedIn' and a 'setLoginStatus' function. Below the Topbar is a 'main' component with the class name 'MainContainer'. Inside the main container, there is a 'Routes' component. The 'Routes' component contains several 'Route' components for different paths: '/', '/blog/:id', '/login', '/something-went-wrong', and '/unauthorized'. Each 'Route' component has a corresponding 'element' prop containing a component like 'HomePage', 'BlogPage', 'LoginPage', 'SomethingWentWrongPage', or 'UnauthorizedPage'. The code also includes a 'setLoginStatus' function that takes a 'status' parameter and calls the 'setLoginStatus' prop on the 'Topbar' component.

```

File Edit Selection View Go Run Terminal Help App.js - Visual Studio Code
src > App.js ...
15
16   const updateLoggedInStatus = (status) => {
17     setLoginStatus(status)
18   }
19
20   return (
21     <BrowserRouter>
22       <div className="App">
23         <Topbar isUserLoggedIn={isUserLoggedIn} setLoginStatus={updateLoggedInStatus}>/>
24         <main className={classes.MainContainer}>
25           <Routes>
26             <Route path="/" element={<HomePage />} />
27             <Route path="/blog/:id" element={<BlogPage isUserLoggedIn={isUserLoggedIn} />} />
28             <Route path="/login" element={<LoginPage isUserLoggedIn={isUserLoggedIn} setLoginStatus={updateLoggedInStatus} />} />
29             <Route path="/something-went-wrong" element={<SomethingWentWrongPage />} />
30             <Route path="/unauthorized" element={<UnauthorizedPage />} />
31           </Routes>
32           </main>
33         </div>
34       </BrowserRouter>
35     );
36   }
37 }

```

Figure 7.21: Handling 401, 404, and 500 error pages

Conclusion

React does not provide an inbuilt mechanism to handle routing. To implement routing, we use a package called React-Router. It provides inbuilt components to create routes for different components. It provides hooks to access

properties and methods related to location, history, search parameters, URL parameters, and so on.

In the upcoming chapter, we will learn about the controlled and uncontrolled implementation of forms in React applications.

Questions

1. What is the purpose of React Router? How do you configure it?
2. What is the use of BrowserRouter, Routes, Route, and Link components provided by React Router?
3. How do you conditionally redirect to a URL?
4. How do you access the search parameters?
5. What are query parameters? How do you configure it?

Multiple choice questions with answers

1. **Which component is used to map URLs to components?**
 - a. Link
 - b. BrowserRouter
 - c. Route
 - d. Routes
2. **Which is a valid way to configure URL parameters?**
 - a. path={"/watch/:id"}
 - b. path={"/watch?id"}
 - c. path={"/watch#id"}
 - d. path={"/watch+id"}
3. **What is a 404 page?**
 - a. It is loaded when there is an error on the application.
 - b. It is loaded when a user request for something which does not exist.
 - c. It is loaded when a user searches for “/404” in the browser.
 - d. It is loaded when API returns 401 as status code.
4. **Which hook is used to change URL programmatically?**

- a. useLocation
 - b. useMatch
 - c. useNavigate
 - d. useParams
5. Is this a valid way to create Route? `<Route to={"/profile"} element={<UserProfile />} />`
- a. True
 - b. False

Answers

Question	Correct Answer
1	c. Route
2	a. path={"/watch/:id"}
3	b. It is loaded when a user request for something which does not exists.
4	c. useNavigate
5	b. False

CHAPTER 8

Controlled and Uncontrolled Components

In this chapter, we will learn about controlled and uncontrolled components. We will learn how to access DOM objects of elements using Ref. We will create our first form and also learn how to generate form elements dynamically.

Structure

In this chapter, we will discuss the following topics:

- Controlled versus uncontrolled components
- Forms in ReactJS
- Intro to Ref
- Dynamically rendered forms
- Callbacks and callback hell
- What are promises
- Creating promises from Scratch
- then() and catch() methods
- Create asynchronous functions using Async and Await
- Restructuring our existing code using Async and Await

Objectives

After studying this chapter, you will be able to create forms and handle data by both uncontrolled and controlled approaches. You will also know how to create async functions and promises.

Controlled versus uncontrolled components

When the component state is managed by the browser, it is known as **Uncontrolled Component**. When the component state is managed by the React application, then it is called a **Controlled Component**.

When we implement forms, there are two ways. We can either implement form elements as controlled components or uncontrolled components. It is preferred to create forms using controlled components, but if your form is really simple where you do not require custom errors or any custom behavior, then you can also implement forms as uncontrolled components. Let us try using both approaches.

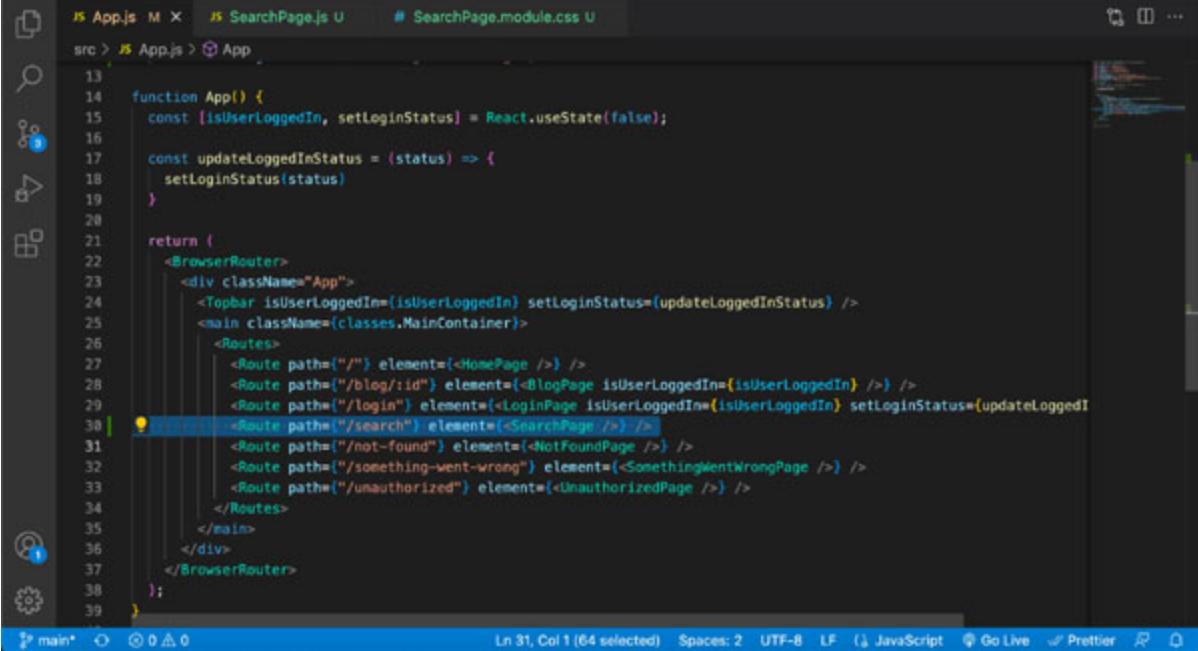
Forms using uncontrolled components

Let us try an example from our previous chapter. We will implement the search functionality. Here is a quick recap, when the users type in the search box and hit enter or click on the search button, they are redirected to the search page. To build this search functionality, we will need to do the following:

1. Create a search page that will show the search string.
2. Add a form with an input box in the topbar to enable users to type the search string.
3. Listen to the form and submit an event to redirect to a search page.
4. Pass search parameters on redirection.
5. Read the search parameters on the search page and show them in UI.

Let us implement it as per the steps mentioned earlier.

Create a search page and add it to the routes as shown in [figure 8.1](#):



```
JS App.js M X JS SearchPage.js U # SearchPage.module.css U
src > JS App.js > App
13
14  function App() {
15    const [isUserLoggedIn, setLoginStatus] = React.useState(false);
16
17    const updateLoggedInStatus = (status) => {
18      setLoginStatus(status)
19    }
20
21    return (
22      <BrowserRouter>
23        <div className="App">
24          <Topbar isUserLoggedIn={isUserLoggedIn} setLoginStatus={updateLoggedInStatus} />
25          <main className={classes.MainContainer}>
26            <Routes>
27              <Route path="/" element={<HomePage />} />
28              <Route path="/Blog/:id" element={<BlogPage isUserLoggedIn={isUserLoggedIn} />} />
29              <Route path="/Login" element={<LoginPage isUserLoggedIn={isUserLoggedIn} setLoginStatus={updateLoggedInStatus} />} />
30              <Route path="/search" element={<SearchPage />} />
31              <Route path="/not-found" element={<NotFoundPage />} />
32              <Route path="/something-went-wrong" element={<SomethingWentWrongPage />} />
33              <Route path="/unauthorized" element={<UnauthorizedPage />} />
34            </Routes>
35          </main>
36        </div>
37      </BrowserRouter>
38    );
  
```

Figure 8.1: Add search component and route

We can add a search box in the topbar, as shown in [figure 8.2](#). This search box will be used by the users to type and search for blogs.

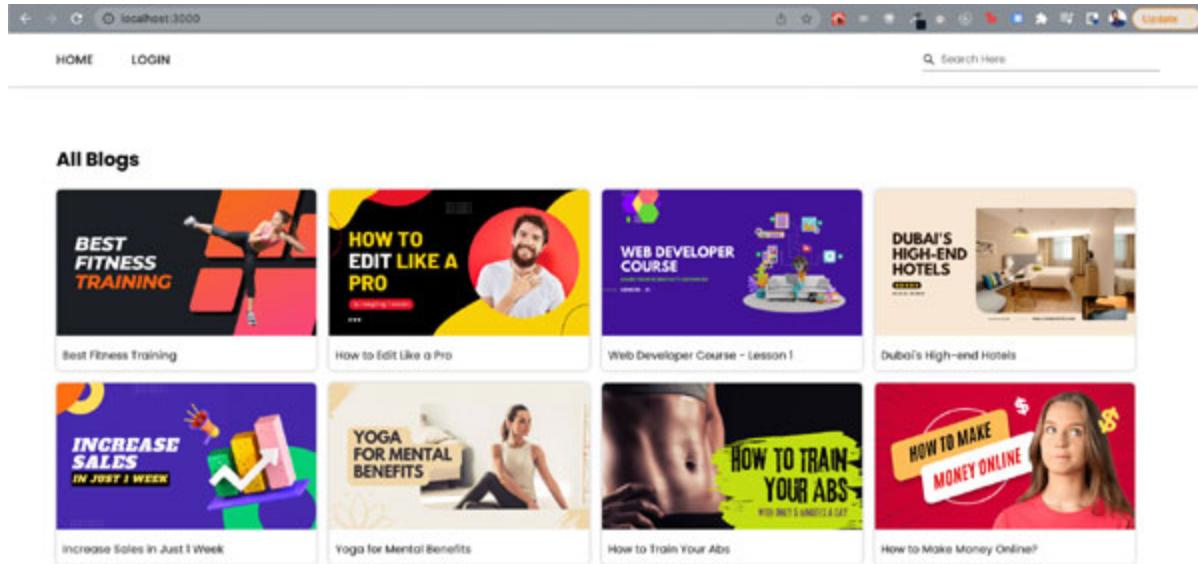
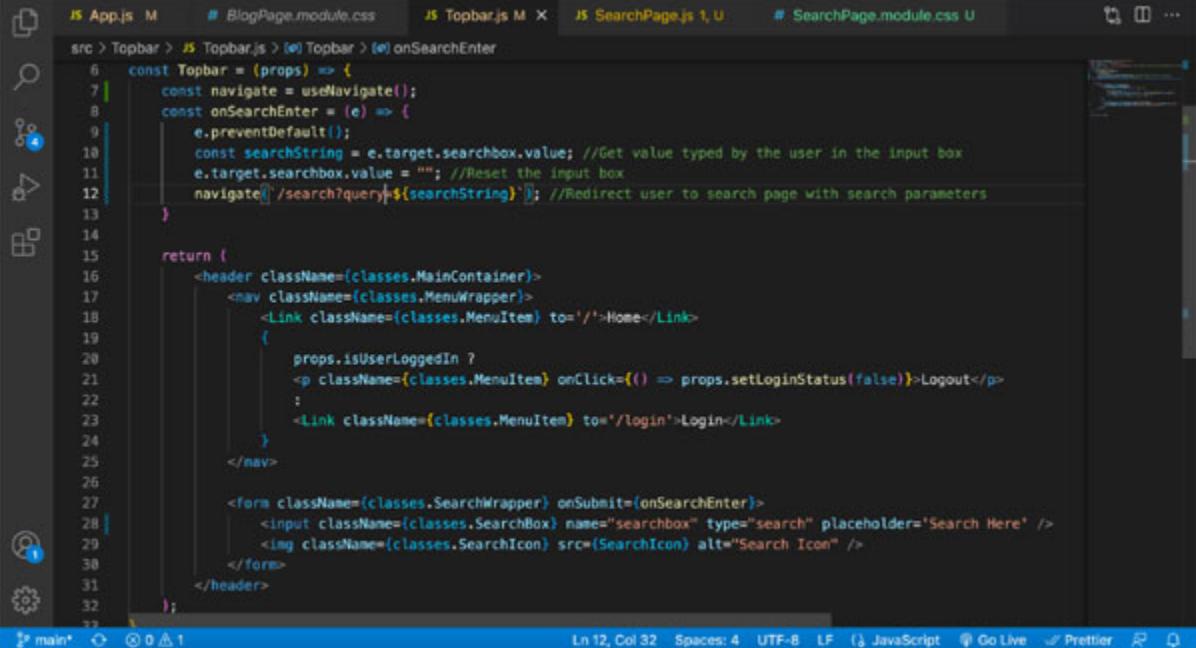


Figure 8.2: Add search box in topbar

We need to add the form and handle the submit event to redirect the user to the search page, as shown in [figure 8.3](#):



```
JS App.js M # BlogPage.module.css JS Topbar.js M X JS SearchPage.js 1, U # SearchPage.module.css U
src > Topbar > JS Topbar.js > (e) Topbar > (e)onSearchEnter
6  const Topbar = (props) => {
7    const navigate = useNavigate();
8    const onSearchEnter = (e) => {
9      e.preventDefault();
10     const searchString = e.target.searchbox.value; //Get value typed by the user in the input box
11     e.target.searchbox.value = ""; //Reset the input box
12     navigate('/search?query=${searchString}'); //Redirect user to search page with search parameters
13   }
14
15   return (
16     <header className={classes.MainContainer}>
17       <nav className={classes.MenuWrapper}>
18         <Link className={classes.MenuItem} to='/>Home</Link>
19         {
20           props.isUserLoggedIn ?
21             <p className={classes.MenuItem} onClick={() => props.setLoginStatus(false)}>Logout</p>
22             :
23             <Link className={classes.MenuItem} to="/login">Login</Link>
24         }
25       </nav>
26
27       <form className={classes.SearchWrapper} onSubmit={onSearchEnter}>
28         <input className={classes.SearchBox} name="searchbox" type="search" placeholder='Search Here' />
29         <img className={classes.SearchIcon} src={SearchIcon} alt="Search Icon" />
30       </form>
31     </header>
32   );
33 }
```

Figure 8.3: Handle form submission

By performing these steps, our users will be redirected to the search page, but they will not see the search string on the UI because we have not written that code yet. So, let us read the search parameters on the search page and show the search string on the user's screen.

In vanilla JS, we would use **URLSearchParams** to manage search parameters. It provides inbuilt methods to perform create, read, update and delete (CRUD) operations on search parameters. If you are unaware of the **URLSearchParams** interface, then read the following code snippet, else you can skip to the next paragraph.

```
const paramsString = "?vId=12122&cId=123499&time=324&uId=449";
//Sample query parameter string you might find in the URL.
const searchParams = new URLSearchParams(paramsString)
//URLSearchParams is used to parse the paramsString.
searchParams.has("vId") //It'll return true. It checks if the
search string is available or not.
searchParams.has("courseId") //false
searchParams.get("vId") //It'll return 12122. It returns the
search parameter's value.
searchParams.get("courseId") //null. If search parameter is
unavailable then it returns null.
```

```

searchParams.append("courseId", 9276) //It will add a new key-value pair to the search parameters.
sp.toString(); //It will return the latest search parameter string. "vId=12122&cId=123499&time=324&uId=449&courseId=9276"
searchParams.set("vId", "565757") //It will update the value of key-value pair in the search parameters.
sp.toString(); //
"vId=565757&cId=123499&time=324&uId=449&courseId=9276"
searchParams.set("commentId", "88722") //It will add a new key-value pair in the search parameters if the key does not exist.
sp.toString(); //
"vId=565757&cId=123499&time=324&uId=449&courseId=9276&commentId=88722"
searchParams.delete("cId"); //It'll delete the key-value pair for the specified key in the search parameters.
sp.toString(); //
"vId=565757&time=324&uId=449&courseId=9276&commentId=88722"

```

We can use an inbuilt hook provided by React router called **useSearchParams**. It wraps around the **URLSearchParams** API, which means it has the same functionality as that of **URLSearchParams**. It is used to perform CRUD operations on search parameters in the URL for the current location. It returns an array with two elements: the first is the current location's search params, and the second is a function that can be used to update the search parameters for the current location, as shown in the following code snippet:

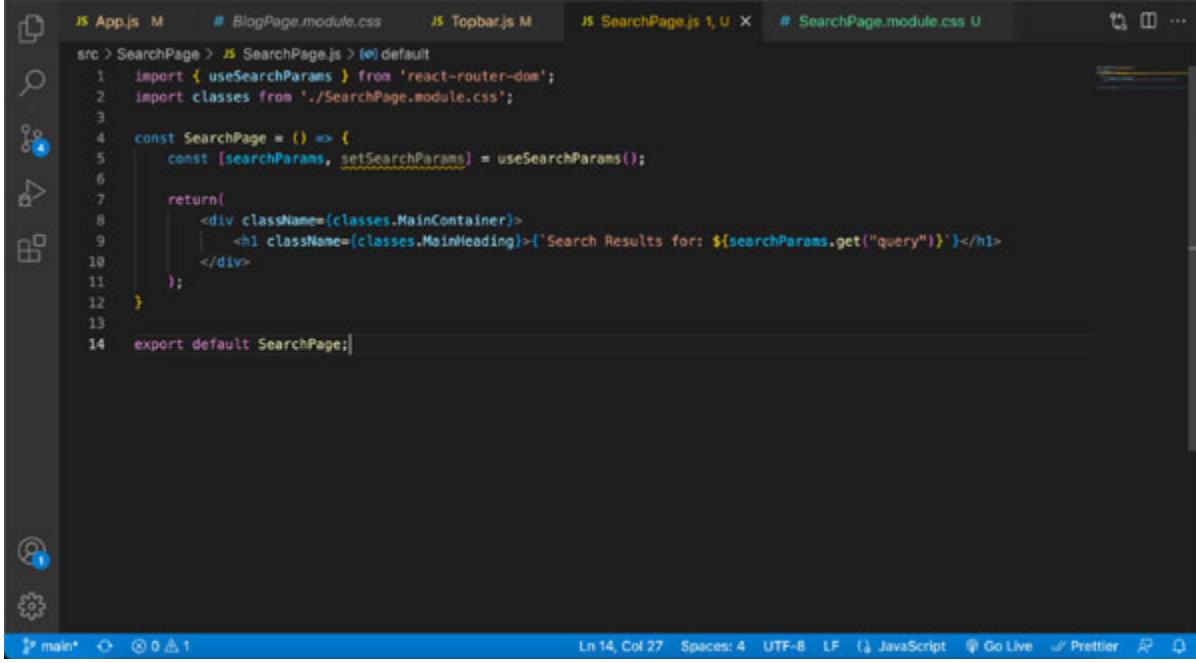
```

import { useSearchParams } from 'react-router-dom'; //To import the hook
const [searchParams, setSearchParams] = useSearchParams();
//Used just like useState

```

This **searchParams** provides us with all the methods required to manage search parameters. It provides us access to all the methods used previously —**has()**, **get()**, **append()**, **set()**, **delete()**, and **toString()**. These functions work exactly the same way they work with the **URLSearchParams** API.

Now that we know how to use the `useSearchParams` hook, we can read the “query” search parameters as shown in [figure 8.4](#):



```
JS App.js M # BlogPage.module.css JS Topbar.js M JS SearchPage.js 1, U X # SearchPage.module.css U
src > SearchPage > JS SearchPage.js > [e] default
1 import { useSearchParams } from 'react-router-dom';
2 import classes from './SearchPage.module.css';
3
4 const SearchPage = () => {
5   const [searchParams, setSearchParams] = useSearchParams();
6
7   return(
8     <div className={classes.MainContainer}>
9       <h1 className={classes.MainHeading}>['Search Results for: ${searchParams.get("query")}]</h1>
10      </div>
11    );
12  }
13
14 export default SearchPage;
```

Figure 8.4: Get search parameters

Hopefully, this gave you some clarity on search parameters and how to implement basic search functionality.

Moving onto a bit more complex form using uncontrolled components. Let us try to implement a login form. The form will have an input field for username and another input field for password. It will also have a login button to allow users to submit the form, as shown in the screenshot given in [figure 8.5](#):

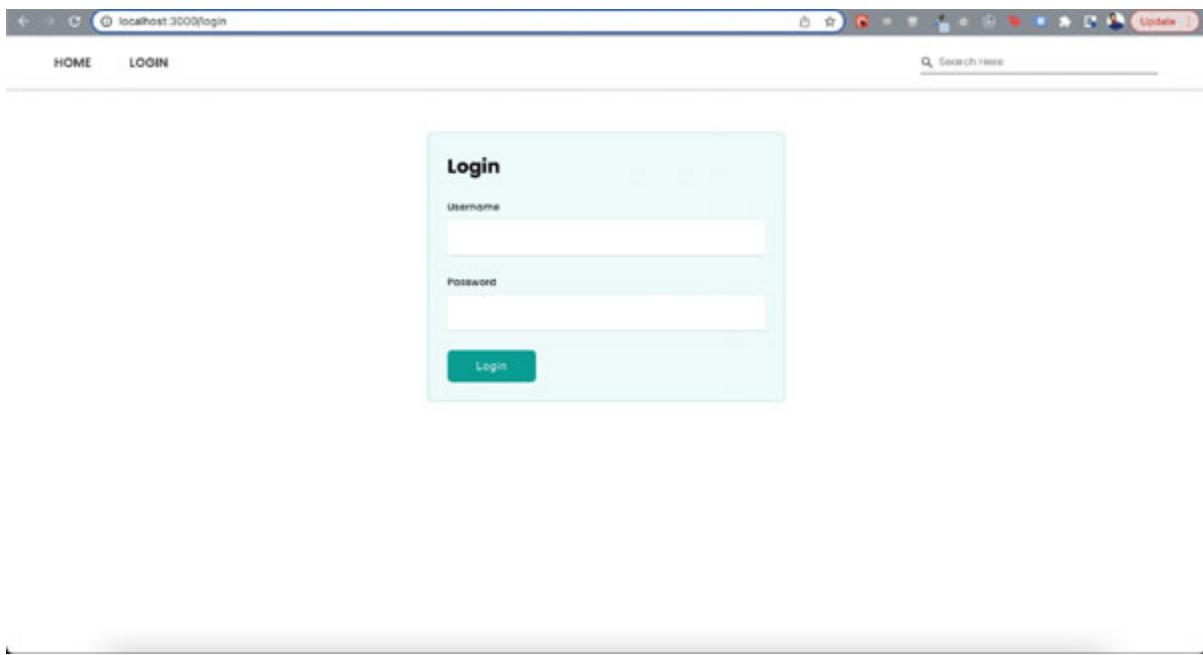


Figure 8.5: Login page

We need to follow these steps to implement form submit functionality:

1. Create an empty callback function that will be called when the form submit event is triggered.
2. This callback function will get the event object as an argument.
3. Using the event object, we can access the form element because the form element triggered the submit event.
4. Using the form element, we can access the input elements inside the form using the name property.
5. Once we have access to username and password input elements, we can access the value entered by the user.
6. We can use these values to trigger API requests to a possible backend. In our case, we do not have an actual backend, so we can use a dummy backend to simulate API requests.

It is advisable that you try writing the code for the preceding steps yourself. After following these steps, your Topbar component's return function should look like as shown in the screenshot provided in [*figure 8.6*](#):

A screenshot of the Visual Studio Code interface. The left sidebar shows a tree view with files: App.js, LoginPage.js, and LoginPage.module.css. The main editor area displays the following code:

```
JS App.js M JS LoginPage.js M # LoginPage.module.css M
src > LoginPage > JS LoginPage.js > [el] default
37     return (
38         <div className={classes.MainContainer}>
39             <form className={classes.LoginForm} onSubmit={onLoginClick}>
40                 <h2 className={classes.MainHeading}>Login</h2>
41                 <label className={classes.InputLabel}>Username</label>
42                 <input className={classes.InputBox} type="text" name='uname' required />
43                 <label className={classes.InputLabel}>Password</label>
44                 <input className={classes.InputBox} type="password" name='pwd' required />
45                 <input type="submit" className={classes.LoginButton} value="Login" />
46             </form>
47         );
48     );
49 }
50 export default LoginPage;
```

The status bar at the bottom shows: Line 51, Col 26, Spaces: 4, UTF-8, LF, JavaScript, Go Live, Prettier.

Figure 8.6: LoginPage component—return statement

The submit event handler for your form will look like the screenshot given in [figure 8.7](#):

A screenshot of the Visual Studio Code interface. The left sidebar shows a tree view with files: App.js, LoginPage.js, and LoginPage.module.css. The main editor area displays the following code:

```
JS App.js M JS LoginPage.js M # LoginPage.module.css M
src > LoginPage > JS LoginPage.js > [el] default
12     );
13     }, [navigate, props.isUserLoggedIn]);
14
15     const onLoginClick = (e) => {
16         e.preventDefault();
17         const formElement = e.target;
18         const usernameElement = formElement.uname;
19         const passwordElement = formElement.pwd;
20
21         const requestData = {
22             email: usernameElement.value,
23             password: passwordElement.value
24         }
25
26         // You can login ONLY with the email addresses available in the
27         // response of this API request "https://reqres.in/api/users?page=1". You
28         // can open the URL in the browser. You can enter any string as a password
29         // for login API.
30
31         axios.post("https://reqres.in/api/login", requestData)
32             .then(successResponse => {
33                 alert("Login Successful!");
34                 props.setLoginStatus(true);
35                 navigate("/", {replace: true})
36             })
37             .catch(error => console.log(error));
38     );
39 }
40
41 export default LoginPage;
```

The status bar at the bottom shows: Line 51, Col 26, Spaces: 4, UTF-8, LF, JavaScript, Go Live, Prettier.

Figure 8.7: LoginPage component—form submit handler

If you notice in [figure 8.7](#), we are using a dummy backend called "<https://reqres.in/>". We are using the API provided by this backend to

simulate the login functionality, but we can only login to the users available in that dummy backend. To get a list of users available in the dummy backend, you can search the following URL in your browser: <https://reqres.in/api/users?page=1>.

Intro to Ref

If you remember, in Vanilla JS, we used to add IDs to our HTML elements and used to access them using the `getElementById` method, but we do not access HTML elements that way in React. Even in the previous implementation, we accessed the input elements through the form element. There is another way to access form elements in uncontrolled elements called Refs. Refs are not specific to just form and form elements. Refs provide a way to access DOM nodes or React elements created in the render method.

Now remember, as this is very important —Refs is not a replacement for ID attribute. We use Ref only when we cannot perform an action declaratively.

To create Refs in a classbased component, we use the `createRef()` method provided by React as shown in the code snippet that follows:

```
class MyComponent extends React.Component {  
  constructor(props) {  
    super(props);  
    this.inputRef = React.createRef(); //We create a new ref  
    called inputRef. "inputRef" is just a variable name, you  
    can call it anything you want.  
  }  
  render() {  
    return <input ref={this.inputRef} />; //We assign  
    inputRef to the ref attribute of JSX element.  
  }  
}
```

When a Ref is passed to an element in render, a reference to the node becomes accessible at the current attribute of the Ref, as shown in the code snippet as follows:

```
const node = this.inputRef.current; //current property gives us access to the reference node. In this example, it'll give us access to the DOM node of the input element.
```

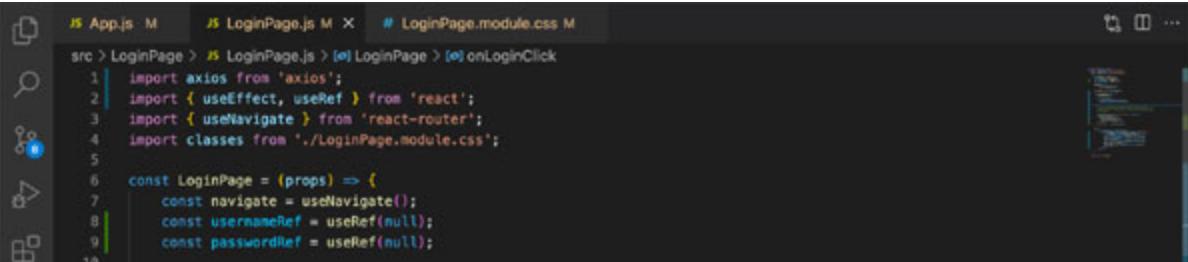
The value of the Ref differs depending on the type of the node:

- When the Ref attribute is used on an HTML element, the Ref is created in the constructor with `React.createRef()`. It receives the underlying DOM element node as its current property.
- When the `ref` attribute is used on a custom class component, the `ref` object receives the mounted instance of the component as its current property. For example, `<CustomClassComponent ref={this.mRef}>`
- You cannot use the `ref` attribute on function components because they do not have instances. For example, `<CustomFunctionalComponent ref={this.mRef}>`, this is not allowed.

To create refs in a function-based component, we use the inbuilt `useRef()` hook provided by React. The `useRef()` hook returns a mutable `ref` object whose current property is initialized to the passed argument (`initialValue`), as shown in the following code snippet. The returned object will persist for the full lifetime of the component.

```
import { useRef } from 'react'; //Import useRef hook
const inputRef = useRef(null); //Create ref using useRef()
hook
<input ref={inputRef} /> //Attach ref to React element
console.log(inputRef.current.value) //Access the element node
using current.
```

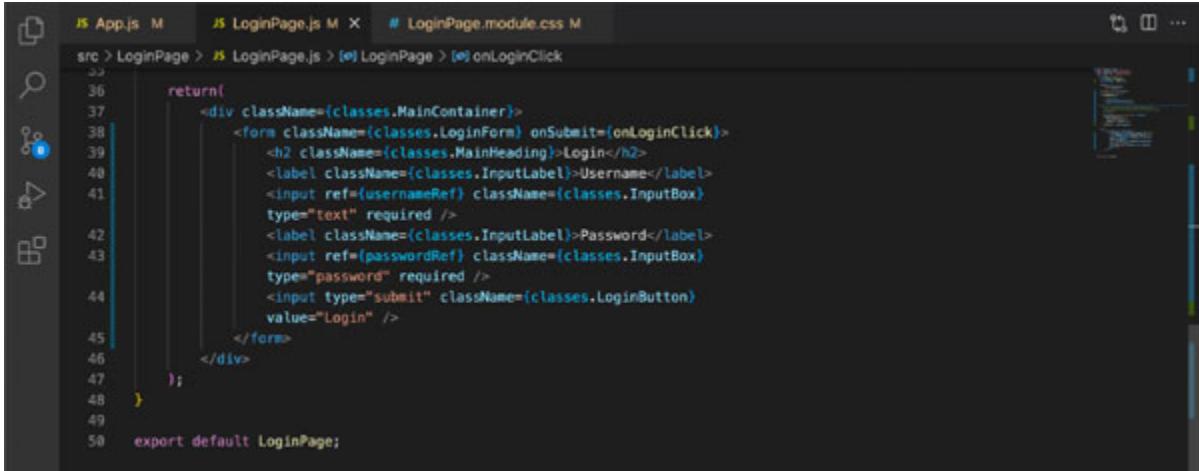
Let us implement the login form using Refs instead of the name property. We will create two Refs for the username and password input field, as shown in the screenshot given in [figure 8.8](#):



```
JS App.js M JS LoginPage.js M # LoginPage.module.css M
src > LoginPage > JS LoginPage.js > LoginPage > onLoginClick
1 import axios from 'axios';
2 import { useEffect, useRef } from 'react';
3 import { useNavigate } from 'react-router';
4 import classes from './LoginPage.module.css';
5
6 const LoginPage = (props) => {
7   const navigate = useNavigate();
8   const usernameRef = useRef(null);
9   const passwordRef = useRef(null);
```

Figure 8.8: Create Refs

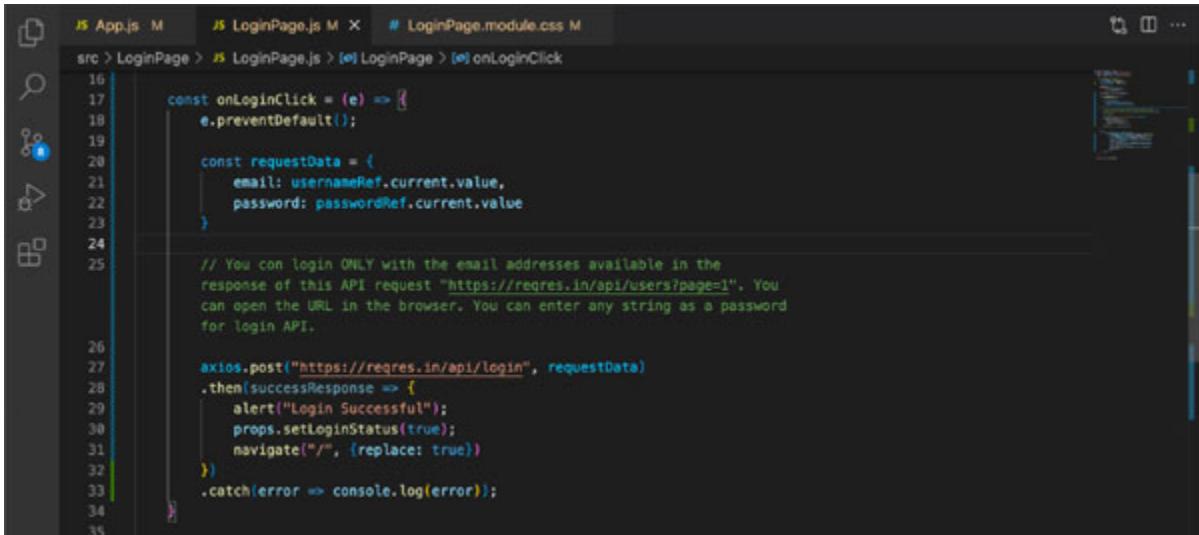
We will attach refs to the form input elements as shown in [figure 8.9](#) :



```
JS App.js M JS LoginPage.js M # LoginPage.module.css M
src > LoginPage > JS LoginPage.js > [e] LoginPage > [e] onLoginClick
35
36     return(
37         <div className={classes.MainContainer}>
38             <form className={classes.LoginForm} onSubmit={onLoginClick}>
39                 <h2 className={classes.MainHeading}>Login</h2>
40                 <label className={classes.InputLabel}>Username</label>
41                 <input ref={usernameRef} className={classes.InputBox}
42                     type="text" required />
43                 <label className={classes.InputLabel}>Password</label>
44                 <input ref={passwordRef} className={classes.InputBox}
45                     type="password" required />
46                 <input type="submit" className={classes.LoginButton}
47                     value="Login" />
48             </form>
49         </div>
50     );
51
52     export default LoginPage;
```

Figure 8.9: Attach Refs to input elements

We will access the input element nodes to access the value attribute, as shown in the screenshot provided in [figure 8.10](#):



```
JS App.js M JS LoginPage.js M # LoginPage.module.css M
src > LoginPage > JS LoginPage.js > [e] LoginPage > [e] onLoginClick
16
17     const onLoginClick = (e) => {
18         e.preventDefault();
19
20         const requestData = {
21             email: usernameRef.current.value,
22             password: passwordRef.current.value
23         }
24
25         // You can login ONLY with the email addresses available in the
26         // response of this API request "https://reqres.in/api/users?page=1". You
27         // can open the URL in the browser. You can enter any string as a password
28         // for login API.
29
30         axios.post("https://reqres.in/api/login", requestData)
31             .then(successResponse => {
32                 alert("Login Successful!");
33                 props.setLoginStatus(true);
34                 navigate("/", {replace: true})
35             })
36             .catch(error => console.log(error));
37     }
38
39     export default LoginPage;
```

Figure 8.10: Access input element nodes

This was another uncontrolled component implementation because the data is stored and managed by the DOM and not React state. You must have got some clarity on how to use refs to access element nodes and their attributes.

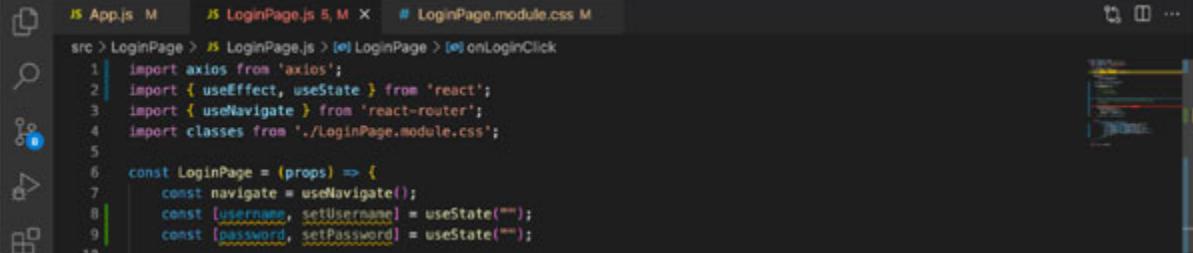
Forms using controlled components

Now, let us try to implement forms using controlled components that means we are going to manage form data in React's state.

We need to follow these steps to implement form as controlled components:

1. Add and attach state variables for individual input items.
2. Add event handlers for input value change. On value change, the state should get updated; when the state is updated, it will update the value for input items.
3. Form submit should take values from state variables to send to the backend.

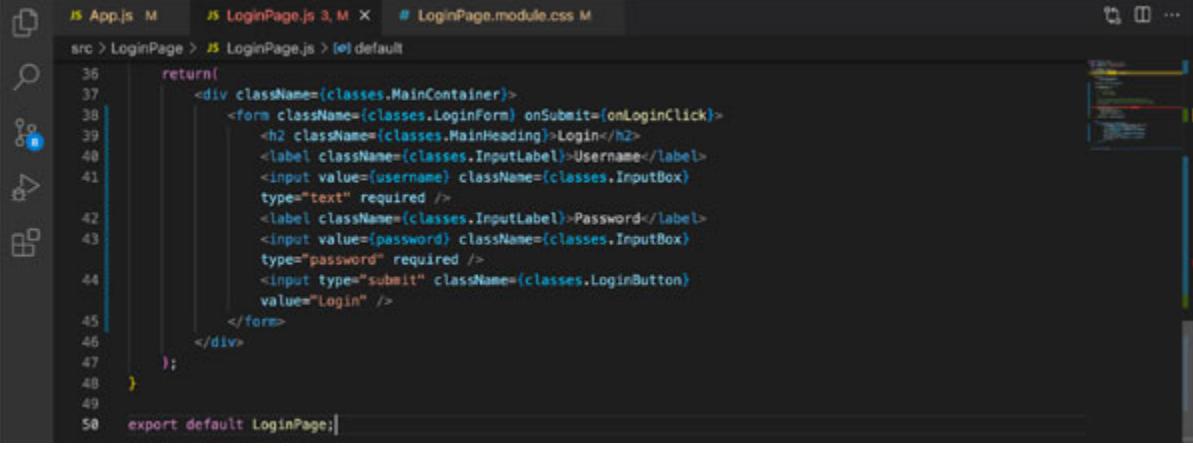
We will create two state variables, one for the username and another for the password, as shown in the screenshot given in [figure 8.11](#):



```
JS App.js M JS LoginPage.js 5, M X # LoginPage.module.css M
src > LoginPage > JS LoginPage.js > [e] LoginPage > [e] onLoginClick
1 import axios from 'axios';
2 import { useEffect, useState } from 'react';
3 import { useNavigate } from 'react-router';
4 import classes from './LoginPage.module.css';
5
6 const LoginPage = (props) => {
7   const navigate = useNavigate();
8   const [username, setUsername] = useState("");
9   const [password, setPassword] = useState("");
```

Figure 8.11: Create state variables

Use these state variables to get input element values as shown in the screenshot provided (refer to [figure 8.12](#)):

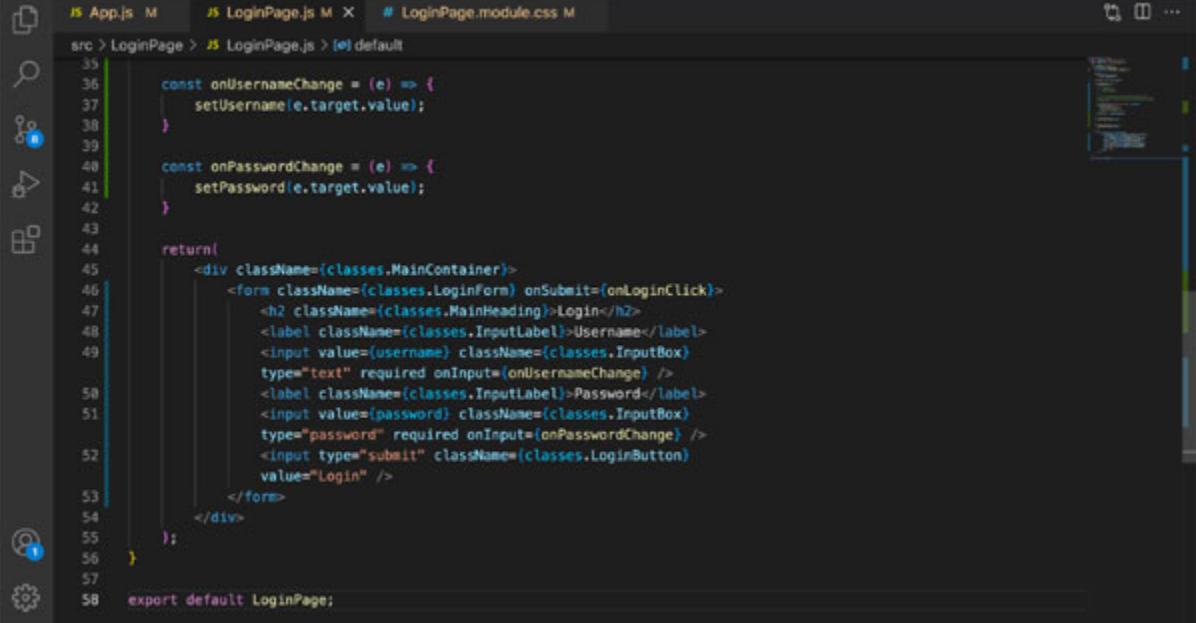


```
JS App.js M JS LoginPage.js 3, M X # LoginPage.module.css M
src > LoginPage > JS LoginPage.js > [e] default
36   return(
37     <div className={classes.MainContainer}>
38       <form className={classes.LoginForm} onSubmit={onLoginClick}>
39         <h2 className={classes.MainHeading}>Login</h2>
40         <label className={classes.InputLabel}>Username</label>
41         <input value={username} className={classes.InputBox}
42           type="text" required />
43         <label className={classes.InputLabel}>Password</label>
44         <input value={password} className={classes.InputBox}
45           type="password" required />
46         <input type="submit" className={classes.LoginButton}
47           value="Login" />
48       </form>
49     </div>
50   );
51
52   export default LoginPage;
```

Figure 8.12: Dynamic values from state variables

Now, if you try to type something in the input items, then nothing will happen. You will not see the text being typed in the input boxes because the

data/values are coming from state variables, and we are not updating the state variables on input value change. So, let us update the state on the input event, as shown in [figure 8.13](#):



```
JS App.js M JS LoginPage.js M # LoginPage.module.css M
src > LoginPage > JS LoginPage.js > [e] default
35
36     const onUsernameChange = (e) => {
37         setUsername(e.target.value);
38     }
39
40     const onPasswordChange = (e) => {
41         setPassword(e.target.value);
42     }
43
44     return (
45         <div className={classes.MainContainer}>
46             <form className={classes.LoginForm} onSubmit={onLoginClick}>
47                 <h2 className={classes.MainHeading}>Login</h2>
48                 <label className={classes.InputLabel}>Username</label>
49                 <input value={username} className={classes.InputBox}
50                     type="text" required onInput={onUsernameChange} />
51                 <label className={classes.InputLabel}>Password</label>
52                 <input value={password} className={classes.InputBox}
53                     type="password" required onInput={onPasswordChange} />
54                 <input type="submit" className={classes.LoginButton}
55                     value="Login" />
56             </form>
57         </div>
58     );
59
60     export default LoginPage;
```

Figure 8.13: Add an input event handler

The preceding change will enable the form to reflect whatever you type in the input elements. Basically, when you try to type something in an input element, it will trigger the input event. Once the event is triggered, the control flow will go to the event handler (**onUsernameChange** or **onPasswordChange** in the preceding example). The event handler will update the state. When the state is updated, it will render the **LoginPage** component with the updated state values. If now you notice everything that you type in the input elements is being stored in the state variables, this means it is a controlled component.

Right now, we have a different event handler function for each input element. Imagine if we had 20 different input elements, then we would have had to create 20 different functions to handle events. Let us make the code a bit better. We will use only a single event handler for all the input elements, as shown in the screenshot provided in [figure 8.14](#):

The screenshot shows a code editor with three tabs: App.js, LoginPage.js, and LoginPage.module.css. The LoginPage.js tab is active, displaying the following code:

```
src > LoginPage > JS LoginPage.js > LoginPage
  36 const onInputChange = (e, htmlFor) => {
  37   switch(htmlFor) {
  38     case "username":
  39       return setUsername(e.target.value);
  40     case "password":
  41       return setPassword(e.target.value);
  42     default:
  43       return;
  44   }
  45 }
  46
  47 return(
  48   <div className={classes.MainContainer}>
  49     <form className={classes.LoginForm} onSubmit={onLoginClick}>
  50       <h2 className={classes.MainHeading}>Login</h2>
  51       <label className={classes.InputLabel}>Username</label>
  52       <input value={username} className={classes.InputBox}
  53         type="text" required onChange={(e) => onInputChange(e,
  54           "username") } />
  55       <label className={classes.InputLabel}>Password</label>
  56       <input value={password} className={classes.InputBox}
  57         type="password" required onChange={(e) => onInputChange(e,
  58           "password") } />
  59       <input type="submit" className={classes.LoginButton}
  60         value="Login" />
  61     </form>
  62   </div>
  63 )
```

Figure 8.14: Code refactor to use a single event handler

We are storing the input values in the component's state. We need to access the values from the state to send to the backend on form submit, as shown in the screenshot that follows (refer to [figure 8.15](#)):

The screenshot shows the same code editor with the LoginPage.js tab active. The code has been updated to include an axios POST request:

```
src > LoginPage > JS LoginPage.js > LoginPage
  17 const onLoginClick = (e) => {
  18   e.preventDefault();
  19
  20   const requestData = {
  21     email: username,
  22     password: password
  23   }
  24
  25   // You can login ONLY with the email addresses available in the
  26   // response of this API request "https://reqres.in/api/users?page=1". You
  27   // can open the URL in the browser. You can enter any string as a password
  28   // for login API.
  29
  30   axios.post("https://reqres.in/api/login", requestData)
  31     .then(successResponse => {
  32       alert("Login Successful!");
  33       props.setloginStatus(true);
  34       navigate("/", {replace: true});
  35     })
  36     .catch(error => console.log(error));
  37 }
```

Figure 8.15: Update API request data

That was the implementation of a basic form using controlled components. It can be cumbersome to work with controlled components for pages with a large number of input elements. Remember, in most cases, we will handle

the forms as controlled components. It is recommended to use controlled components.

Keeping the users logged in

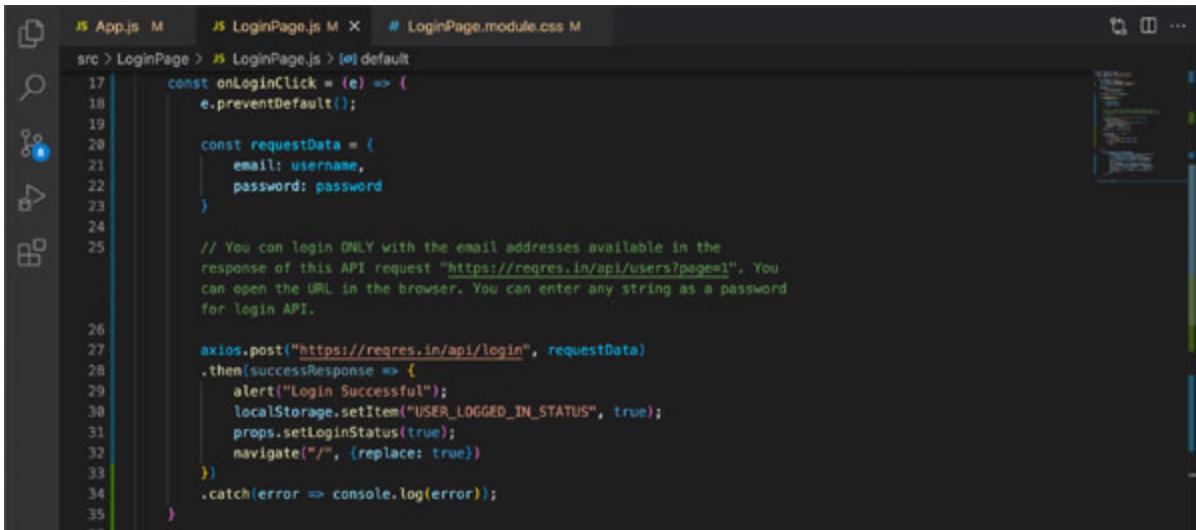
So far, in our application, we are storing the logged-in status in the application state. When you refresh the page, all the data stored in the application's state gets reset. Let us try to retain the logged-in state even when a user closes the browser.

The question is, “*How do I store the logged-in state in the browser?*”. Well, it is simple, we can use local storage provided by HTML5.

We need to follow these steps to implement this functionality:

1. Update the local storage when the user is logged-in.
2. Fetch logged-in status from local storage and initialize the app state variable.
3. Update the local storage when the user is logged-out.

When the login call is successful, then we need to store the logged-in status as true in local storage, as shown in Line 30 of the screenshot in [figure 8.16](#):



```
JS App.js M JS LoginPage.js M # LoginPage.module.css M
src > LoginPage > JS LoginPage.js > [e] default
17 const onLoginClick = (e) => {
18   e.preventDefault();
19
20   const requestData = {
21     email: username,
22     password: password
23   }
24
25   // You can login ONLY with the email addresses available in the
26   // response of this API request "https://reqres.in/api/users?page=1". You
27   // can open the URL in the browser. You can enter any string as a password
28   // for login API.
29
30   axios.post("https://reqres.in/api/login", requestData)
31     .then(successResponse => {
32       alert("Login Successful!");
33       localStorage.setItem("USER_LOGGED_IN_STATUS", true);
34       props.setLoginStatus(true);
35       navigate("/", {replace: true})
36     })
37     .catch(error => console.log(error));
38 }
```

Figure 8.16: Store logged-in status in local storage

If we login to the application now, it will store the “**USER_LOGGED_IN_STATUS**” as true in local storage. We can verify this by

opening the application tab of the developer tools, as shown in the screenshot given in [figure 8.17](#):

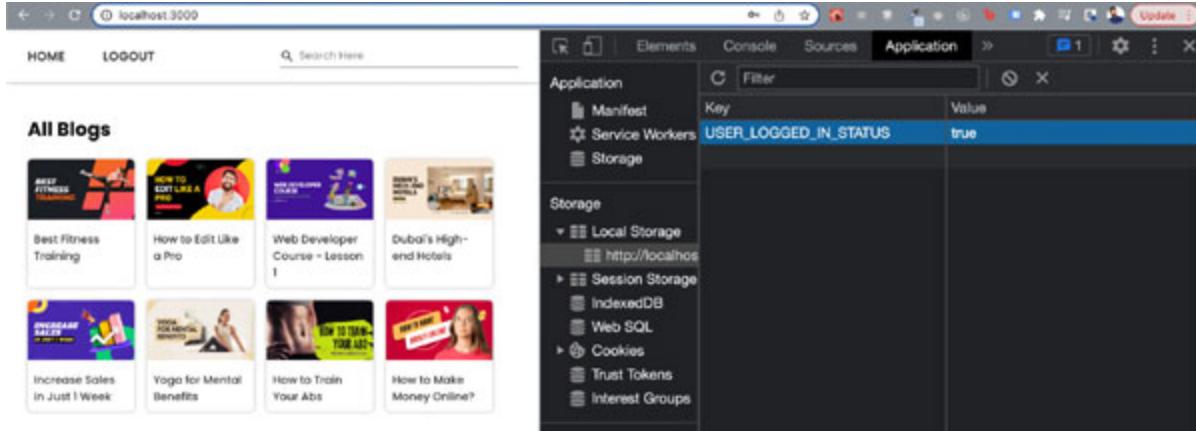


Figure 8.17: Verify data in local storage

The preceding step will only store the logged-in status, but the application is not using that value. If we refresh the page, then it will show the user as logged-out even though we have stored logged-in status as true in local storage. To fix this issue, we need to fetch the “**USER_LOGGED_IN_STATUS**” from local storage and set it to the **isUserLoggedIn** state variable on app load as shown in Lines 15 and 16 of the screenshot ([figure 8.18](#)):

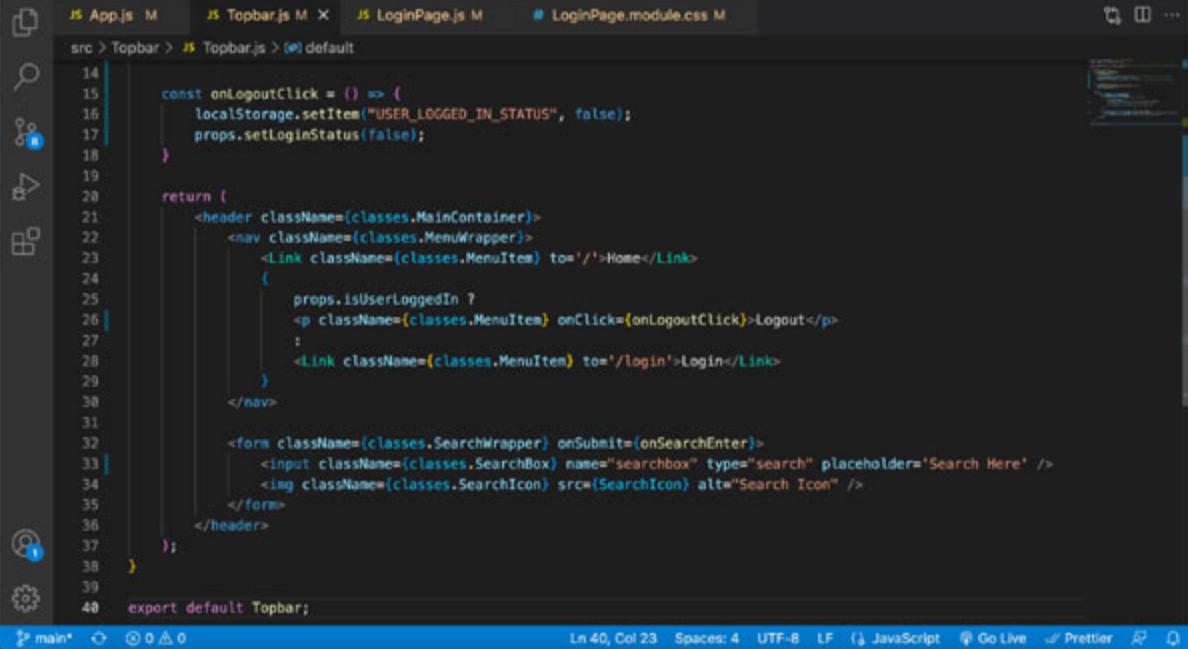
```

14 function App() {
15   const loggedInStatusFromStorage = localStorage.getItem("USER_LOGGED_IN_STATUS") === "true";
16   const [isUserLoggedIn, setIsLoginStatus] = React.useState(loggedInStatusFromStorage);
17
18   const updateLoggedInStatus = (status) => {
19     setIsLoginStatus(status);
20   }
21
22   return (
23     <BrowserRouter>
24       <div className="App">
25         <Topbar isLoggedIn={isUserLoggedIn} setLoginStatus={updateLoggedInStatus}>/>
26         <main className={classes.MainContainer}>
27           <Routes>
28             <Route path="/" element={<HomePage />} />
29             <Route path="/blog/:id" element={<BlogPage isLoggedIn={isUserLoggedIn} />} />
30             <Route path="/login" element={<LoginPage isLoggedIn={isUserLoggedIn} setLoginStatus={updateLoggedInStatus} />} />
31             <Route path="/search" element={<SearchPage />} />
32             <Route path="/not-found" element={<NotFoundPage />} />
33             <Route path="/something-went-wrong" element={<SomethingWentWrongPage />} />
34             <Route path="/unauthorized" element={<UnauthorizedPage />} />
35           </Routes>
36         </main>
37       </div>
38     </BrowserRouter>
39   );
}

```

Figure 8.18: Fetch initial data from local storage

If we click on the logout menu item and refresh the page, it will show the user as logged-in. This is happening because we are not setting the logged-in status as false in the local storage. So, let us do that on logout click as shown in [figure 8.19](#):



```
JS App.js M JS Topbar.js M X JS LoginPage.js M # LoginPage.module.css M
src > Topbar > JS Topbar.js > default
14 const onLogoutClick = () => {
15   localStorage.setItem("USER_LOGGED_IN_STATUS", false);
16   props.setLoginStatus(false);
17 }
18
19 return (
20   <header className={classes.MainContainer}>
21     <nav className={classes.MenuWrapper}>
22       <Link className={classes.MenuItem} to="/">Home</Link>
23       {
24         props.isUserLoggedIn ?
25           <p className={classes.MenuItem} onClick={onLogoutClick}>Logout</p>
26           :
27           <a href="/login" className={classes.MenuItem}>Login</a>
28       }
29     </nav>
30
31     <form className={classes.SearchWrapper} onSubmit={onSearchEnter}>
32       <input className={classes.SearchBox} name="searchbox" type="search" placeholder="Search Here" />
33       <img className={classes.SearchIcon} src={SearchIcon} alt="Search Icon" />
34     </form>
35   </header>
36 )
37
38
39
40 export default Topbar;
```

Figure 8.19: Update local storage on user logout

That was all about implementing forms as controlled components. Now, you can implement forms as both controlled and uncontrolled components.

Callback functions and callback hell

In JavaScript, the code is executed line-by-line in a sequence, so when we run a parallel operation or asynchronous operation like fetching data from the backend, JavaScript does not wait for the response; it simply executes the next line of code. So, we give the asynchronous operation a function to call when it is completed. This function is called a callback function.

For example,

```
$.get('some-url', () => {
  // call back function
})
```

This **get** method provided by jQuery is an asynchronous function. It runs in parallel to the main program flow. It sends a call to the backend, and

whenever the response is received, it runs the Callback Function and works perfectly fine.

Now, let us say the call returns a list of objects with IDs, and you need to send another call to the backend to get details for a specific ID.

How would you do it?

You cannot write it outside, as shown in the following code snippet:

```
const id = 0;
$.get('https://api-url-1', (response) => {
    id = response.data[0].id
})
$.get(`https://api-url-2/${id}`, (response) => {
    //callback function
})
```

You have no way of knowing when the response is received for the first API request, and your second API will be triggered with id=0. So, your only option is to put the second API call inside the first callback function, as shown in the following code snippet:

```
$.get('https://api-url-1', (response) => {
    const id = response.data[0].id
    $.get(`https://api-url-2/${id}`, (response) => {
        //callback function
    })
})
```

Now say, once the second API response is received, you need to send another call to the backend. Again, you have no choice but to put the get call inside the second callback function, as shown in the following code snippet:

```
$.get('https://api-url-1', (response) => {
    const id = response.data[0].id
    $.get(`https://api-url-2/${id}`, (response) => {
        const data = response.data
        $.get(`https://api-url-3/${data.id}`, (response) => {
            //callback function
        })
    })
})
```

Of course, you will also have to handle the failure scenarios. Let us add them too, as shown in the following code snippet:

```
$ .get('https://api-url-1', (response) => {
  const id = response.data[0].id
  $ .get(`https://api-url-2/${id}`, (response) => {
    const data = response.data
    $ .get(`https://api-url-3/${data.id}`, (response) => {
      //callback function
    })
    .fail(err => {
      console.log(err)
    })
  })
  .fail(err => {
    console.log(err)
  })
})
.fail(err => {
  console.log(err)
})
```

You must have started noticing that when you have a callback inside a callback like this, the code starts to get messier and less readable. This situation is known as **Callback Hell**.

Now, to avoid this callback hell, JavaScript introduced something called **Promise**.

Introduction to promises

A promise is used to handle the asynchronous result of an operation. It defers the execution of a code block until an asynchronous request is completed. This way, other operations can keep running without interruption.

A promise can have the following three states:

- **Pending:** This means the operation is ongoing.
- **Fulfilled:** This means the operation was completed.

- **Rejected:** The operation did not complete, and an error can be thrown.

Create a promise

Follow this syntax that follows to create a **Promise** object:

```
const mPromise = new Promise((resolve, reject) => {
    // Promise body
    //Call resolve() when the operation is successful.
    //Call reject() when the operation is failed.
})
```

To create a **Promise** object, we use the **Promise** class with **new** keyword. Let us try to create a Promise for the AJAX call we saw in the previous section.

```
const promiseForApi = (url) => new Promise((resolve, reject)
=> {
    $.get(url, (data) => {
        resolve(data);
    }).fail(err => {
        reject(new Error(`API call failed with following error:
        ${err}`)))
    })
})
```

Now, this does not trigger the API call because we have not called the **promiseForApi1** function. If you check the networks tab in the browser developer tools; there will be no request sent from our code.

To call the **promiseForApi1** function, we use parenthesis just like any other function. The function will return a promise, as shown in the subsequent code snippet:

```
promiseForApi("https://api-url-1"); //This will return a
promise object
```

After writing the preceding code, our API is triggered, but how do we handle the response? To handle both success and failure responses, we chain **then()** and **catch()** methods. These methods are available by default for our promise object. The **then()** method is called when the **resolve()** is executed and **then()** method receives the data passed in the **resolve()** method. Similarly, **catch()** method is called when the **reject()** method is executed, and it receives the data passed in the **reject()** method.

Let us update our code and handle the response as shown in the following code snippet:

```
promiseForApi("https://api-url-1")
  .then(successResponse => {
    const id = successResponse.data[0].id;
  })
  .catch(error => {
    console.log("Error", error);
  })
```

As you can see in the preceding code snippet, both **then()** and **catch()** accept callback functions as an argument, which are called based on the success or failure of the Promise.

[Chaining multiple promises](#)

In the previous section, we saw how to create a promise and trigger the API request using a Promise. So, coming back to the problem of callback hell. How does Promise avoid the problem of callback hell? Simply, Promise allows us to chain multiple promises. So instead of nested functions, we get a chain of functions that is much easier to read, understand, and debug.

In the following code snippet, we have chained three promises to fix the callback hell situation we faced earlier.

```
promiseForApi("https://api-url-1")
  .then(response => {
    const id = successResponse.data[0].id;
    return promiseForApi(`https://api-url-2/${id}`)
  })
  .then(response => {
    const data = response.data
    return promiseForApi(`https://api-url-3/${data.id}`)
  })
  .then(response => {
    console.log("Success response of third API")
  })
  .catch(error => { //Single catch to handle errors for all the promises
    console.log("Error", error);
  })
```

```
})
```

Create asynchronous functions using `async` and `await`

JavaScript introduced “`async`” and “`await`” keywords in ES6 to create asynchronous functions. When we append the “`async`” keyword to the function, it returns a resolved promise by default on execution, as shown in the screenshot that follows:

```
> async function greetings() {
  return "Hello"
}
console.log(greetings())
▼Promise {<fulfilled>: 'Hello'} VM9899:4
▶ [[Prototype]]: Promise
[[PromiseState]]: "fulfilled"
[[PromiseResult]]: "Hello"
```

Figure 8.20: Async function

In the preceding example, the returned value is a promise, so we can chain `then()` and `catch()` methods, as shown in the screenshot in [figure 8.21](#):

```
> async function greetings() {
  return "Hello"
}

greetings()
.then(res => console.log(res))
.catch(err => console.log(err))
Hello VM10525:6
```

Figure 8.21: then() and catch() methods

If we do not return any value, then a resolved promise is returned with an undefined value, as shown in the screenshot given in [figure 8.22](#):

```
> async function greetings() {
}

greetings()
.then(res => console.log(res))
.catch(err => console.log(err))
undefined VM11107:6
```

Figure 8.22: Undefined returned by default

Normally, a promise is executed in the background, which means JavaScript does not wait for the Promise to resolve; it moves on to execute the next line of code, as shown in the screenshot provided in [figure 8.23](#):

```
> const greetings = () => new Promise(resolve => {
  setTimeout(() => {
    resolve("Hello after 5 seconds")
  }, 5000)
})

greetings()
.then(res => console.log(res))
.catch(err => console.log(err))
console.log("Console after greetings")
Console after greetings
VM12901:10
< undefined
Hello after 5 seconds
VM12901:8
```

Figure 8.23: Promise does not block execution

Let us say we want to make JavaScript wait for the Promise to resolve before it executes the next line of code. We can achieve this by adding “`await`” before a Promise, as you can see in the screenshot given in [figure 8.24](#):

```
> const greetings = () => new Promise(resolve => {
  setTimeout(() => {
    resolve("Hello after 2 seconds")
  }, 2000)
})

await greetings()
.then(res => console.log(res))
.catch(err => console.log(err))
console.log("Console after greetings promise")
Hello after 2 seconds
VM58557:8
Console after greetings promise
VM58557:10
< undefined
> |
```

Figure 8.24: await keyword

Restructuring our existing code using `async` and `await`

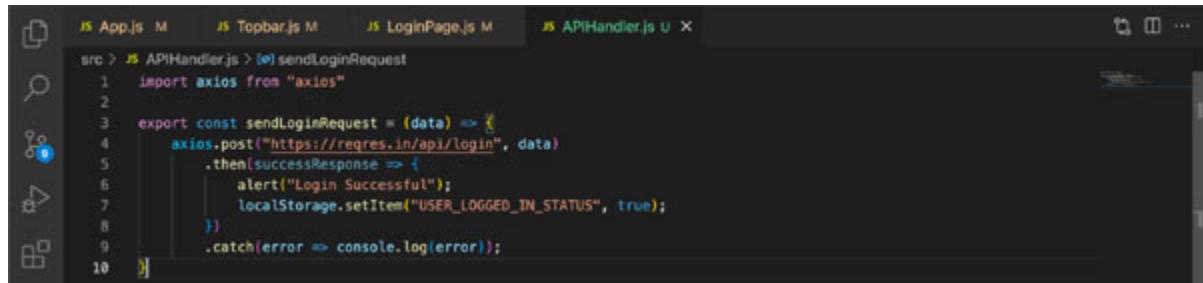
It is a good practice to write API request logic in separate files rather than in the component itself. This helps in the following two ways:

- We can reuse the API request functions across the applications. It follows the DRY principle.
- Side effects are handled separately from the components, which makes it easier to test components using unit-testing libraries such as Mocha, Jest, and so on.

We just have one API request in our blog application, which is the Login API. Let us move that to a separate file. We need to follow these steps to refactor our code:

1. Create a new file, “**APIHandler.js**”.
2. Add a new function, “**sendLoginRequest**” and move the API request-related code to the function.
3. Call **sendLoginRequest()** function from **LoginPage** component and handle the response.

Add the API request-related code as shown in the screenshot in [figure 8.25](#):



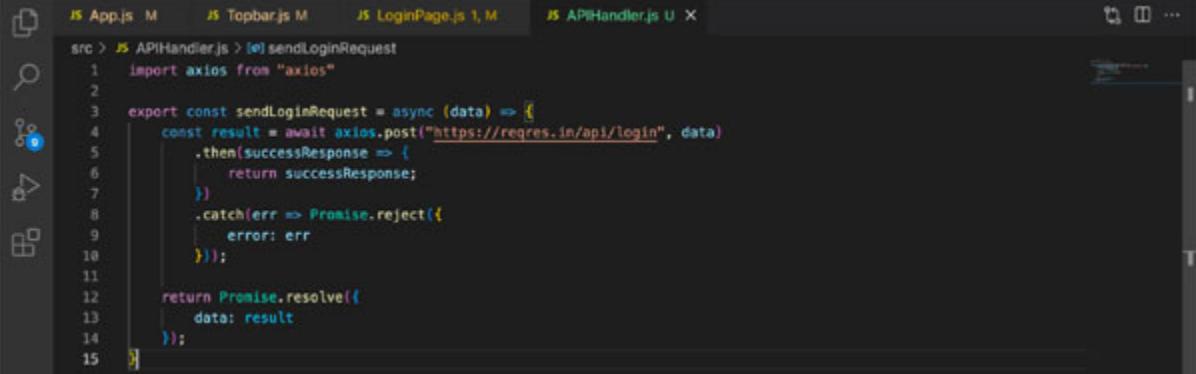
```

JS App.js M JS Topbar.js M JS LoginPage.js M JS APIHandler.js U X
src > JS APIHandler.js > [e] sendLoginRequest
1 import axios from "axios"
2
3 export const sendLoginRequest = (data) => {
4   axios.post("https://regres.in/api/login", data)
5     .then(successResponse => {
6       alert("Login Successful");
7       localStorage.setItem("USER_LOGGED_IN_STATUS", true);
8     })
9     .catch(error => console.log(error));
10 }

```

Figure 8.25: Code refactor to separate network calls

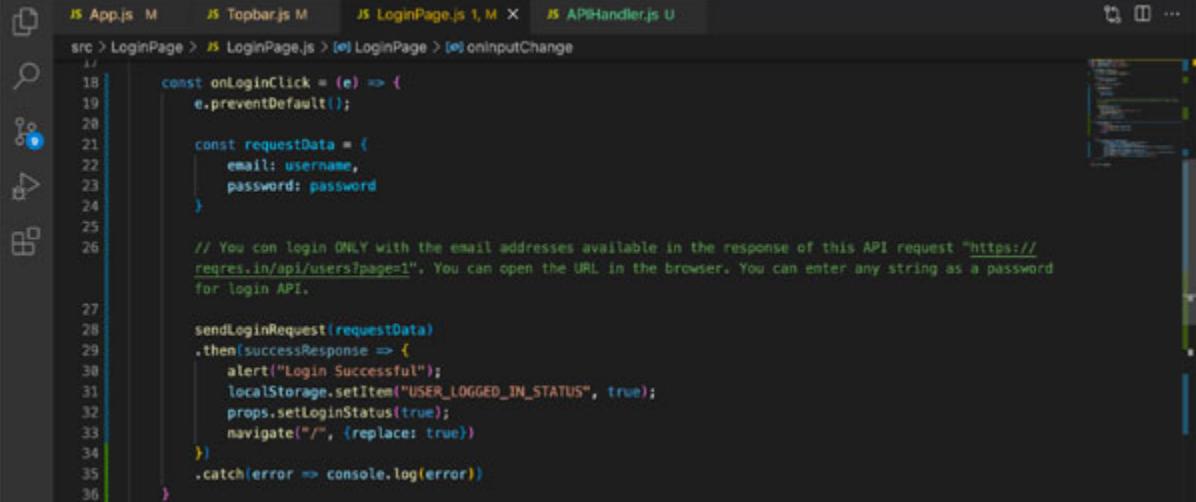
The problem with this code is that when you call it from the **LoginPage** component, it will not return the response immediately. How do we fix this problem? We can convert this function into an async function, as shown in the screenshot as follows:



```
src > JS APIHandler.js > [e] sendLoginRequest
1 import axios from "axios"
2
3 export const sendLoginRequest = async (data) => {
4     const result = await axios.post("https://resges.in/api/login", data)
5     .then(successResponse => {
6         return successResponse;
7     })
8     .catch(err => Promise.reject({
9         error: err
10    }));
11
12    return Promise.resolve({
13        data: result
14    });
15}
```

Figure 8.26: Adding `async` functions

As you can see in the preceding screenshot, `sendLoginRequest()` is an `async` function that returns a promise in both success and failure scenarios. Because this is returning a Promise, we can chain `then()` and `catch()` function in `LoginPage` component to handle the response as shown in the screenshot given in [figure 8.27](#):



```
src > LoginPage > JS LoginPage.js > [e] LoginPage > [e] onInputChange
18
19 const onLoginClick = (e) => {
20     e.preventDefault();
21
22     const requestData = {
23         email: username,
24         password: password
25     }
26
27     // You can login ONLY with the email addresses available in the response of this API request "https://resges.in/api/users?page=1". You can open the URL in the browser. You can enter any string as a password for login API.
28
29     sendLoginRequest(requestData)
30     .then(successResponse => {
31         alert("Login Successful!");
32         localStorage.setItem("USER_LOGGED_IN_STATUS", true);
33         props.setLoginStatus(true);
34         navigate("/", {replace: true})
35     })
36     .catch(error => console.log(error))
}
```

Figure 8.27: Handle API response using `then()` and `catch()`

Refactor the project structure

So far, we have been adding all the components and other files directly in the `src` folder, as shown in the screenshot given in [figure 8.28](#):

Figure 8.28: Flat project structure

Developers follow different approaches to structure their applications. One of such way is to create two main folders—**containers** and **components**. All the components which represent a page are placed inside the **containers** folder. All the components which are part of the page are placed inside the **components** folder, as shown in the following screenshot:

Figure 8.29: Project restructuring

Apart from containers and components, you might have files that contain utility functions like Date formatters, Data transformations, or any other functions, which are used across the application. You can add those files inside a “**utilities**” or “**utils**” folder.

You can add all the icons, illustrations, background images, and so on, inside the “**assets**” folder.

You might also have files related to the API requests. These files can be added inside a “**webServices**” or “**NetworkServices**” folder, as shown in the screenshot given in [*figure 8.30*](#):

Figure 8.30: Adding API-related files

Conclusion

We can create forms and handle their data by both uncontrolled and controlled approaches. We can access element nodes using Ref. We can use Refs in both class-based and function-based components. JavaScript introduced Promises to solve the problem of callback hell. We can use `async-await` keywords to make a function asynchronous.

In the next chapter, we will learn about state management using Redux. We will learn about the core concepts of Redux, and how it can function with and without React. We will also learn about different hooks provided by the redux packages.

Questions

1. What are controlled and uncontrolled components?

2. Implement forms using the controlled element approach.
3. What is callback hell? How do you resolve this issue?
4. What is a Ref? How do you create refs in function-based components?
5. What is async and await?

Multiple choice questions with answers

1. A controlled component is when?

- a. There is no state.
- b. The state is managed by React.
- c. The state is managed by Browser.
- d. None of the above.

2. An uncontrolled component is when?

- a. There is no state.
- b. The state is managed by React.
- c. The state is managed by Browser.
- d. None of the above.

3. Which keyword is used to make JavaScript wait for the execution to complete?

- a. Async
- b. Await
- c. Promise
- d. Ref

4. Which method is used to handle the success of a promise?

- a. then()
- b. catch()
- c. useRef()
- d. async()

5. Which is not a valid promise state?

- a. Pending
- b. Fulfilled
- c. Rejected
- d. Completed

Answers

Question	Correct Answer
1	b. The state is managed by React
2	c. The state is managed by Browser
3	b. Await
4	a. then()
5	d. Completed

CHAPTER 9

State Management Using Redux

In this chapter, we will learn about Redux and how it works behind the scenes. We will understand how to install and set up Redux in a React app. We will create a global store and its reducer to manage state changes. We will learn how to dispatch actions and handle the state changes accordingly.

Structure

In this chapter, we will discuss the following topics:

- Introduction to Redux
- Redux installation
- Introduction and setup of React-Redux
- Configure Redux in a React app
- Handle multiple reducers
- Implement Redux using Hooks
- Async actions using Middleware

Objectives

After studying this chapter, you will be able to understand the core concepts of Redux, its installation, and its implementation. You will also learn about Redux-thunk and creating asynchronous actions.

Introduction to Redux

Redux is an open-source JavaScript library for managing the application state. It follows the flux pattern. It is independent of React, which means it can also be used with Angular or Vue, or any other framework for state management.

Let us talk about Redux outside React for a bit to understand how it works. Redux revolves around the following four main concepts:

1. Global Store
2. Actions
3. Reducer
4. Subscriptions

Global store

This is the single central place that is used to manage the state of an application. A store is a JavaScript object with a few special functions and abilities that make it different than a plain global object:

- We should not directly modify or change the state stored in the global store.
- The only way to update the state is to create an action object which we dispatch to the store to tell it which part of the global store needs to be updated.
- When an action is dispatched, it goes to the reducer, which is a function. The reducer contains all the logic to create a new state based on the old one. This new state replaces the old state.
- Once the global store is updated, it notifies all the subscribers of the global state variables that the state has been updated so the UI can be updated with the new data.

Actions

An action is a normal JavaScript object that has a **type** property in it. Based on the type value present in action, the reducer knows how to update the global store. The type should be a meaningful string. For example:

```
const incrementAction = {
  type: "counter/increment"
}
const decrementAction = {
  type: "counter/decrement"
}
```

We can also add additional properties to hold some custom data, as shown in the following code snippet:

```
const incrementBy50Action = {
  type: "counter/customIncrement",
```

```

        incrementBy: 50
    }
const incrementBy100Action = {
    type: "counter/customIncrement",
    incrementBy: 100
}
const decrementBy75Action = {
    type: "counter/customDecrement",
    decrementBy: 75
}

```

These additional properties can be accessed inside the reducer to update the global state with these values.

Reducer

This is a synchronous function. It contains all the logic to update the global store. Only the reducer can update the global store. If it receives multiple requests, then they are executed one by one. It never updates the global store directly. It gets access to the old state as an argument along with the action object. It updates the global state immutably, and it copies the old state to create a new copy of the state, which then updates the global store. You can see the following sample reducer function:

```

const sampleReducer = (state, action) => {
    switch(action.type) {
        case "counter/increment":
            return { ...state, counter: state.counter + 1 }
        default:
            return state
    }
}

```

As you can see in the code snippet, the reducer returns a newly created object, which is used to update the global store. Once the store is updated, it notifies the subscribers that the state has been updated and sends out the new data to update the UI with the latest data.

Let us summarize the entire Redux flow:

1. It all starts with an action. To trigger an action, we use the dispatch method provided by the Redux store.

2. Once the action is dispatched, the action object goes to the reducer.
3. Reducer checks if it knows how to update the state based on the action type and updates the state accordingly.
4. Once the global store is updated, it broadcasts the updated data to all the subscribers.

You can see the Redux flow in the following figure:

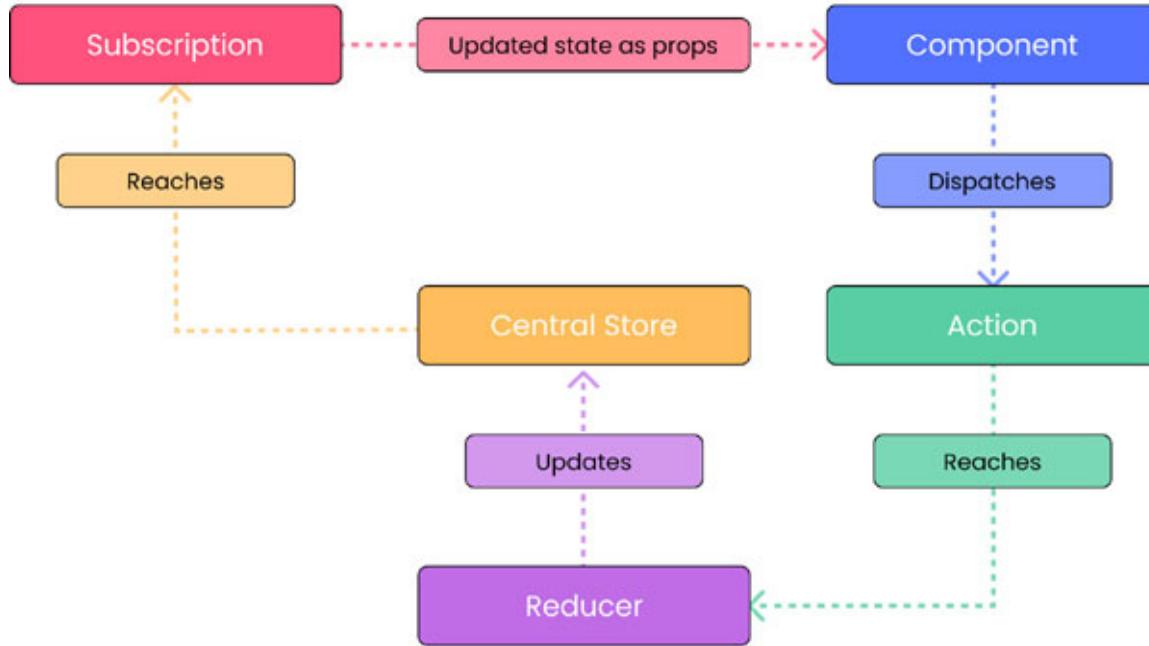


Figure 9.1: Redux flow

Redux installation

As we learned earlier, Redux is independent of React, so first, we are going to try Redux without React. Redux is an npm package, so we can use it in a node environment. Do not worry, we are not going to learn node for this.

You need to follow these steps:

1. Create a new folder.
2. Open terminal/command prompt.
3. Navigate to the folder you created in Step 1.
4. Run the following command “`npm init`”. It will give the message shown in the screenshot as follows:

```
qaifi@Qaifis-MacBook-Pro redux-without-react % npm init
This will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See 'npm help init' for definitive documentation on these fields
and exactly what they do.

Use 'npm install <pkg>' afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (redux-without-react) 
```

Figure 9.2: Initialize project

5. Once you see the prompt to enter the package name, please press *Ctrl + C* or *Command + C* to exit the command prompt. This will set up the **package.json** file for us.
6. Run the command “**npm i redux**” to install the latest version of redux. It will be automatically added as a dependency in the **package.json** file.
7. In VS Code, open the folder we created in Step 1.
8. You should see the following files and folders. You can even verify that Redux was installed for the project, as shown in the screenshot given in [figure 9.3](#):

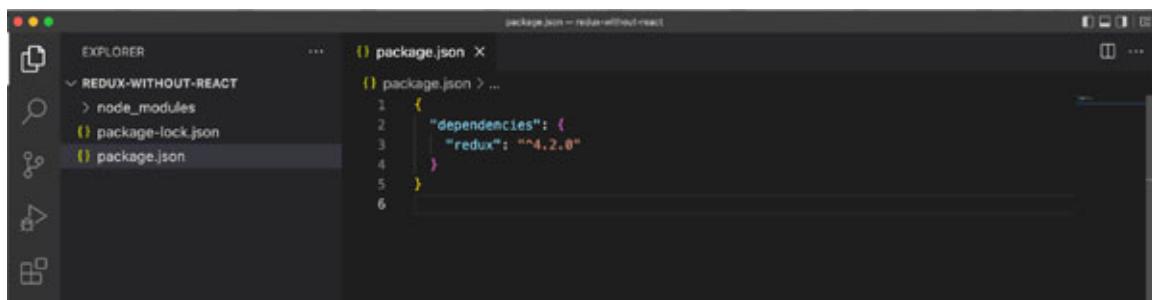


Figure 9.3: Check the package.json file

By following the preceding steps, we have set up a simple node project. Now, let us create a JavaScript file to try out Redux. We can call it **index.js**.

In order to use Redux in the file, we need to import it first, as shown in the screenshot that follows:

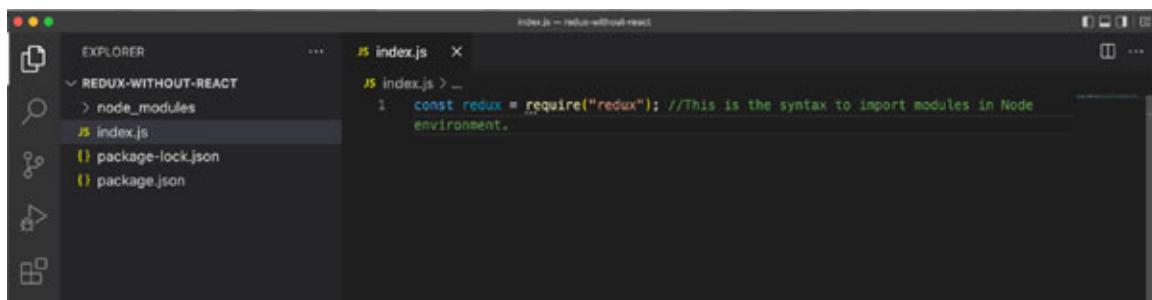
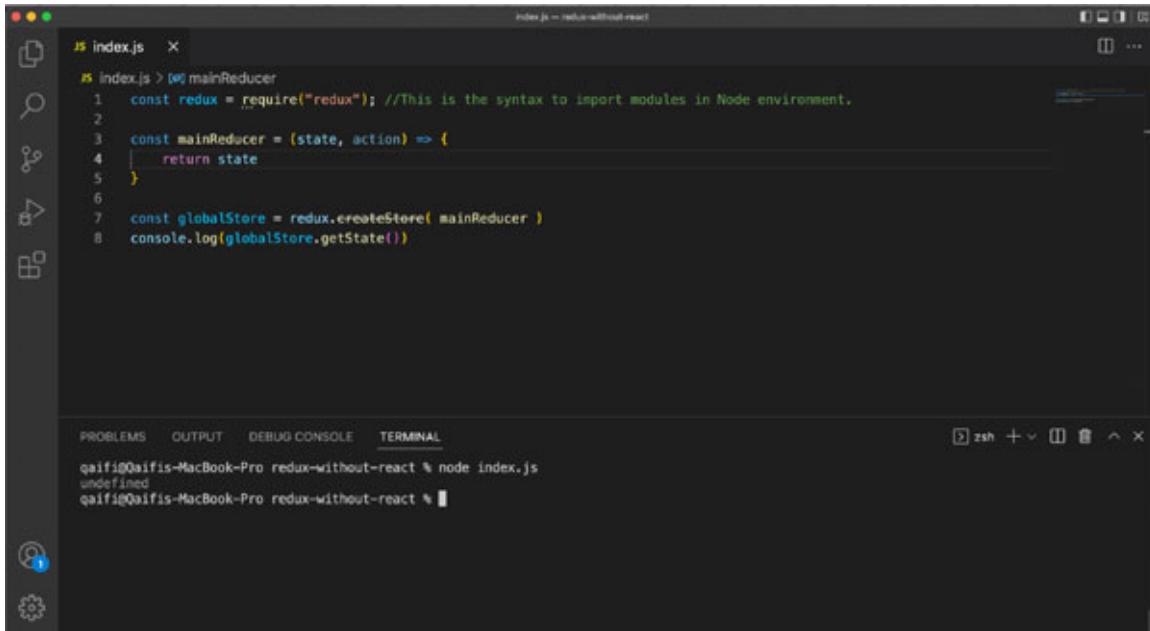


Figure 9.4: Import Redux

To create the global store, Redux has provided an inbuilt function called “**createStore()**”. The **createStore()** method requires a reducer function so let us create both, as shown in [figure 9.5](#):



The screenshot shows a VS Code interface with a dark theme. On the left, there's a file tree with 'index.js' selected. The main editor area contains the following code:

```
JS index.js  X
index.js > [0] mainReducer
1 const redux = require("redux"); //This is the syntax to import modules in Node environment.
2
3 const mainReducer = (state, action) => {
4   return state
5 }
6
7 const globalStore = redux.createStore( mainReducer )
8 console.log(globalStore.getState())
```

Below the editor is a terminal window showing the command 'node index.js' being run and the output 'undefined'. The terminal tab bar includes 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL'. The status bar at the bottom right shows 'zsh +v'.

Figure 9.5: Create a global store

As shown in the preceding screenshot (refer to [figure 9.5](#)), to run this JavaScript file, we simply do “**node index.js**”. When this script is executed, it prints undefined. This is because we are trying to get the global state object but initially, there is no state object because we have not added anything to the global store.

We can make a small change to our reducer function. If there is no old state available in the arguments, then set it to default or initial state as shown in the screenshot given in [figure 9.6](#):

The screenshot shows a VS Code interface with a dark theme. The left sidebar has icons for file, search, and other files. The main editor window contains the following code:

```
index.js  X
index.js > ...
1 const redux = require("redux"); //This is the syntax to import modules in Node environment.
2
3 const initialState = {
4   counter: 0
5 }
6
7 const mainReducer = (state = initialState, action) => {
8   return state
9 }
10
11 const globalStore = redux.createStore( mainReducer )
12 console.log(globalStore.getState())
```

The terminal at the bottom shows the output of running the script:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
zsh +^[[0m
qaifi@Oaifis-MacBook-Pro redux-without-react % node index.js
{ counter: 0 }
qaifi@Oaifis-MacBook-Pro redux-without-react %
```

Figure 9.6: Set initial state

Notice that we have created an object for the initial state on Line 3. We are using it to set a default value to the state argument in the reducer on like 7. Now that we have set up an initial state `getState()` returns that initial state as the global store.

Remember to trigger actions we have to the dispatch method provided by the global store. Let us try to dispatch an action, as shown in the following screenshot ([figure 9.7](#)):

The screenshot shows a VS Code interface with a dark theme. The left sidebar has icons for file, search, and other files. The main editor window contains the following code:

```
index.js  X
index.js > ...
1 const redux = require("redux"); //This is the syntax to import modules in Node environment.
2
3 const initialState = {
4   counter: 0
5 }
6
7 const mainReducer = (state = initialState, action) => {
8   return state
9 }
10
11 const globalStore = redux.createStore( mainReducer )
12 console.log(globalStore.getState())
13
14 globalStore.dispatch({ type: "counter/increment" });
15 console.log(globalStore.getState())
```

The terminal at the bottom shows the output of running the script:

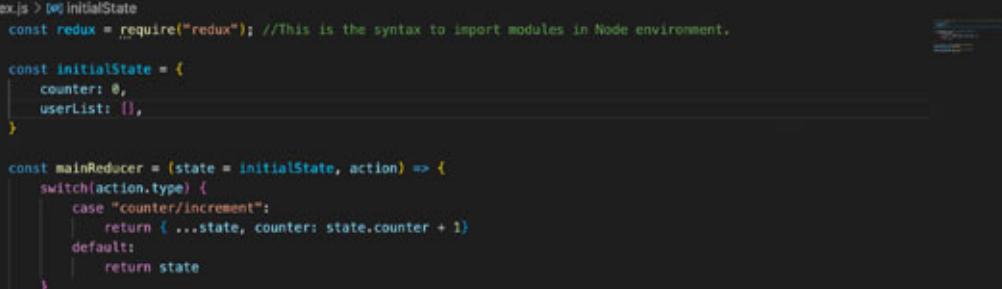
```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
zsh +^[[0m
qaifi@Oaifis-MacBook-Pro redux-without-react % node index.js
{ counter: 0 }
{ counter: 0 }
qaifi@Oaifis-MacBook-Pro redux-without-react %
```

Figure 9.7: Dispatch actions

As we can see in the screenshot in [figure 9.7](#), we have dispatched an action on Line 14, and then we tried to print the global store, but nothing changed in the global store as printed in the terminal.

Can you guess what the problem is?

Well, we also need to handle the state update in the reducer for this specific action, as shown in the following screenshot:



The screenshot shows a code editor window with the file `index.js` open. The code implements a simple counter application using the Redux library. It defines an initial state object with `counter` and `userList` properties. A reducer function, `mainReducer`, handles actions like `counter/increment` by returning a new state where the `counter` value is incremented. Finally, it creates a global store and dispatches an increment action.

```
index.js
1 const redux = require("redux"); //This is the syntax to import modules in Node environment.
2
3 const initialState = {
4   counter: 0,
5   userList: []
6 }
7
8 const mainReducer = (state = initialState, action) => {
9   switch(action.type) {
10     case "counter/increment":
11       return { ...state, counter: state.counter + 1}
12     default:
13       return state
14   }
15 }
16
17 const globalStore = redux.createStore( mainReducer )
18 console.log(globalStore.getState())
19
20 globalStore.dispatch({ type: "counter/increment" })
```

Figure 9.8: Handle actions in reducer

As we can see on Line 11, we are using the spread operator to create a copy of the old state. In the console, we can see that the initial state is printed before the action is dispatched, but after the action is dispatched, the counter is updated to 1 in the global store.

Let us try to dispatch multiple actions and see how the values are updating, as shown in the screenshot as follows:

The screenshot shows a VS Code interface with the file 'index.js' open. The code defines a global store and dispatches four 'counter/increment' actions. The terminal below shows the state object being updated after each dispatch.

```
index.js - redux-without-react
16  const globalStore = redux.createStore( mainReducer )
17  console.log(globalStore.getState())
18
19  globalStore.dispatch({ type: "counter/increment" });
20  console.log(globalStore.getState())
21  globalStore.dispatch({ type: "counter/increment" });
22  console.log(globalStore.getState())
23  globalStore.dispatch({ type: "counter/increment" });
24  console.log(globalStore.getState())
25  globalStore.dispatch({ type: "counter/increment" });
26  console.log(globalStore.getState())
27

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
zsh + × ⓘ ⌘ ⌘ ×

qaifis@Oaifis-MacBook-Pro redux-without-react % node index.js
{ counter: 0, userList: [] }
{ counter: 1, userList: [] }
{ counter: 2, userList: [] }
{ counter: 3, userList: [] }
{ counter: 4, userList: [] }
qaifis@Oaifis-MacBook-Pro redux-without-react %
```

Figure 9.9: Dispatch multiple actions

As you can see in the console, after every “counter/increment” type action, the counter value is increased by 1 in the global store.

Also, if you notice, we have been adding a `console.log()` statement after every dispatch. If you remember, there was something called a **subscription**; let us subscribe to the global store to get automatically notified of the changes in the global state, as shown in the screenshot given in [figure 9.10](#):

The screenshot shows a VS Code interface with the file 'index.js' open. The code defines a main reducer, creates a global store, and subscribes to it to log the state. Four 'counter/increment' actions are dispatched, and the state is logged each time.

```
index.js - redux-without-react
8  const mainReducer = (state = initialState, action) => {
9    switch(action.type) {
10      case "counter/increment":
11        return { ...state, counter: state.counter + 1}
12      default:
13        return state
14    }
15
16  const globalStore = redux.createStore( mainReducer )
17
18  globalStore.subscribe(() => {
19    console.log(globalStore.getState());
20  })
21
22
23  globalStore.dispatch({ type: "counter/increment" });
24  globalStore.dispatch({ type: "counter/increment" });
25  globalStore.dispatch({ type: "counter/increment" });
26  globalStore.dispatch({ type: "counter/increment" });

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
zsh + × ⓘ ⌘ ⌘ ×

qaifis@Oaifis-MacBook-Pro redux-without-react % node index.js
{ counter: 1, userList: [] }
{ counter: 2, userList: [] }
{ counter: 3, userList: [] }
{ counter: 4, userList: [] }
qaifis@Oaifis-MacBook-Pro redux-without-react %
```

Figure 9.10: Subscribe to actions

Now, if you see, we have removed all the `console.log()` statements and only added one inside the subscribe method. Every time the global store is updated, the subscribe method is called, which is printing the global state object in the console.

You can access the above code at the following URL:

<https://github.com/qaifikhan/RFJS-redux-without-react/tree/main>

I hope this gave you some sense of Redux, its 4 main concepts and how it works behind the scenes.

Configure Redux in a React app

Let us try to set up Redux in our React application. To connect Redux with React, we need two packages—Redux and React-Redux. The Redux package contains all the redux-related features. React-Redux contains all the tools and features required to connect Redux to React applications. We will connect Redux to a simple counter app. You can access the code at the URL below:

<https://github.com/qaifikhan/RFJS-redux-counter-base-code>

When you install dependencies and run the application, it will load the page shown in the following screenshot:

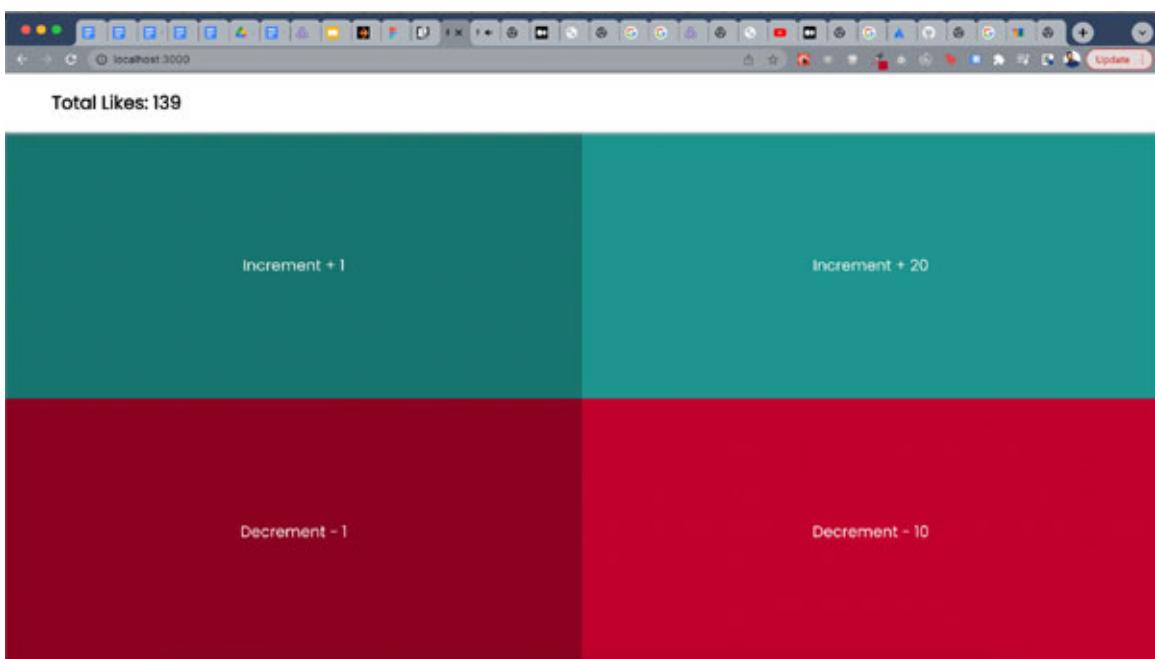
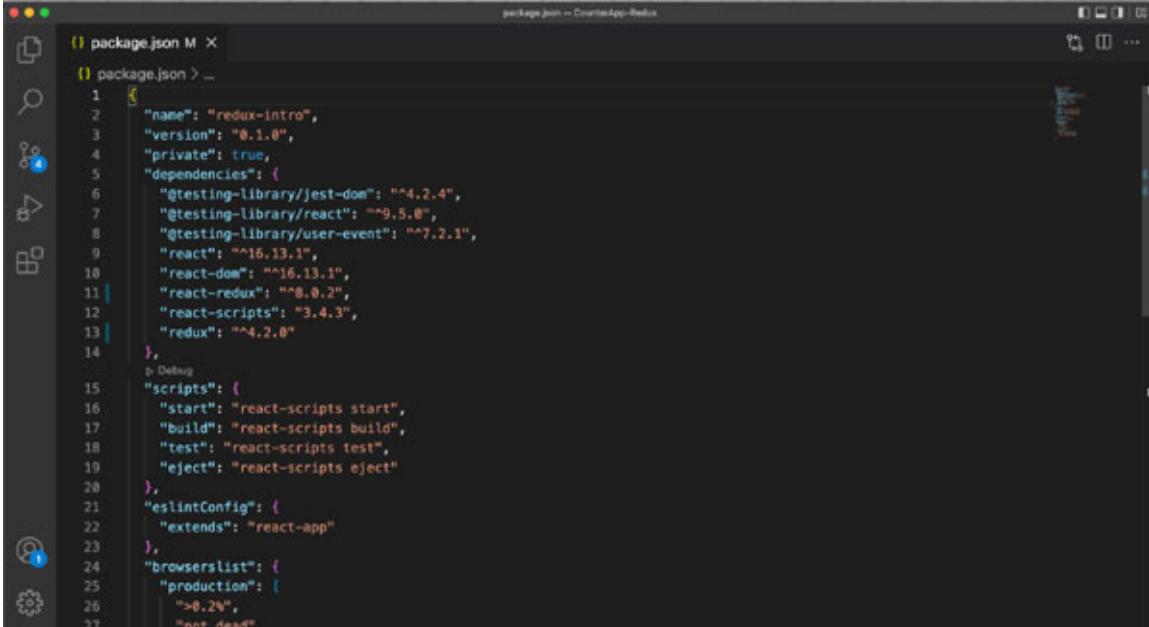


Figure 9.11: Application landing page

Please follow these steps to install Redux and React-Redux in a React app:

1. Open your terminal or command prompt.
2. Navigate to the React project folder and run the following command “**npm i redux react-redux**”.
3. Wait for the installation to complete.
4. Open the **package.json** file to verify if both packages were installed, as shown in the following screenshot:



```

{
  "name": "redux-intro",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^4.2.4",
    "@testing-library/react": "^9.5.0",
    "@testing-library/user-event": "^7.2.1",
    "react": "^16.13.1",
    "react-dom": "^16.13.1",
    "react-redux": "^8.0.2",
    "react-scripts": "3.4.3",
    "redux": "^4.2.0"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": "react-app"
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead"
    ]
  }
}

```

Figure 9.12: Verify package.json file

5. Start your app.

Create a global store

First, let us create a global store and connect it to our React app. To create the global store, we can use the **createStore()** method provided by the redux package, as shown in the following code snippet:

```

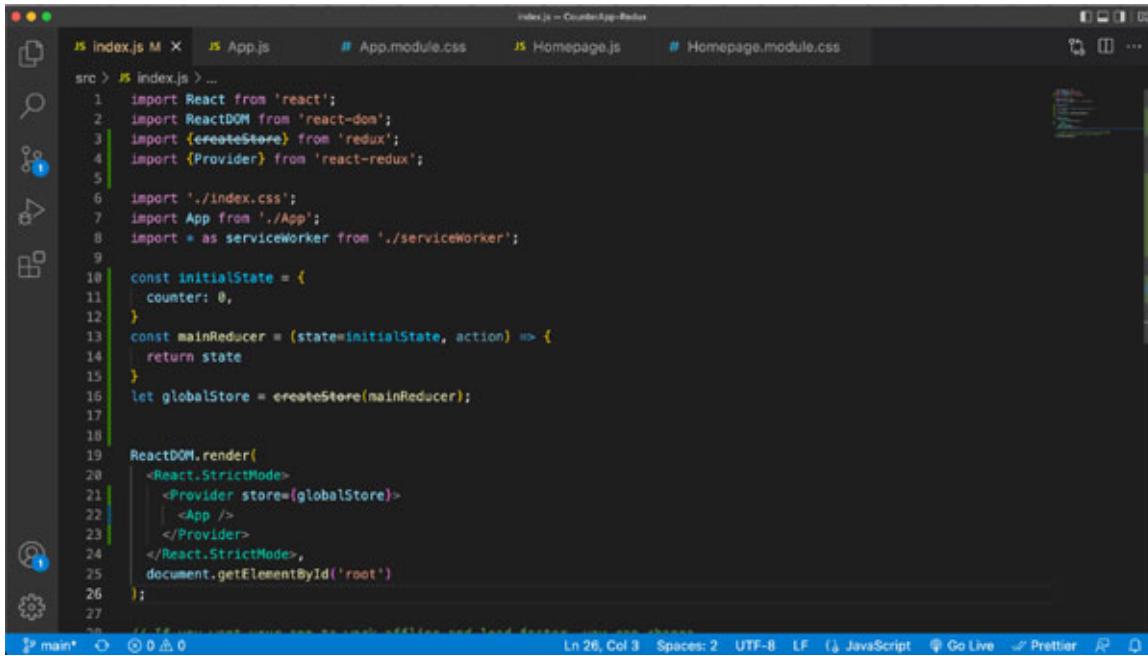
import {createStore} from 'redux';
const initialState = {
  counter: 0,
}
const mainReducer = (state=initialState, action) => {
  return state
}
let globalStore = createStore(mainReducer);

```

The preceding code should look familiar to you. We did the same thing earlier when we introduced Redux without React. Once we create the global store, we need to connect it to the React app. This is where the React-Redux package comes into play. It gives us a component called `<Provider>` that is used to connect our React app with redux. We must wrap our entire React app in the `<Provider>` component, so the best place to do it is in the `index.js` file, as shown in the following code snippet:

```
import {Provider} from 'react-redux';
ReactDOM.render(
  <React.StrictMode>
    <Provider store={globalStore}>
      <App />
    </Provider>
  </React.StrictMode>,
  document.getElementById('root')
);
```

After you make changes in the `index.js` file, it should look something like as shown in the screenshot that follows:

A screenshot of a code editor showing the `index.js` file for a React application. The code defines a global store using the `Provider` component from the `react-redux` library. The `index.js` file imports React, ReactDOM, createStore, and Provider. It sets up an initial state with a counter, defines a main reducer, creates a global store, and then renders the entire application wrapped in a `Provider` component into the root element of the DOM.

```
index.js M X JS App.js # App.module.css JS Homepage.js # Homepage.module.css
src > JS index.js > ...
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3 import {createStore} from 'redux';
4 import {Provider} from 'react-redux';
5
6 import './index.css';
7 import App from './App';
8 import * as serviceWorker from './serviceWorker';
9
10 const initialState = {
11   counter: 0,
12 }
13 const mainReducer = (state=initialState, action) => {
14   return state
15 }
16 let globalStore = createStore(mainReducer);
17
18
19 ReactDOM.render(
20   <React.StrictMode>
21     <Provider store={globalStore}>
22       <App />
23     </Provider>
24   </React.StrictMode>,
25   document.getElementById('root')
26 );
27
```

Figure 9.13: Setup global store in React application

Now that we have connected Redux with our React app. We can access values from the global store inside our components.

Fetch state from the global store in components

To subscribe and access the state variables from the global store, we use **connect()**, which is an unbuilt function made available by the react-redux package. The **connect()** is a curried function.

The first parenthesis can accept two arguments—a callback function to select the state variables from the global state and another callback function to set up dispatch functions to update the global state.

The callback function to select state variables from the global store gets the global state as an argument. This callback function needs to return an object. This object will contain the state variables we select from the global store, as shown in the following code snippet:

```
const mapGlobalStatetoProps = (globalState) => {
  return {
    totalLikes: globalState.totalLikes,
    totalDislikes: globalState.totalDislikes,
    isUserLoggedIn: globalState.userLoggedInStatus
  }
}
```

The second parenthesis accepts the component as an argument, which allows **connect()** method to pass the state variables and dispatch actions as props to the component, as you can see in the following code snippet:

```
import {connect} from 'react-redux';
//.....some code
<p>Total Likes: {props.totalLikes}</p>
<p>Total Dislikes: {props.totalDislikes}</p>
//.....some code
export default connect(mapGlobalStatetoProps)(Topbar);
```

This is what the current application looks like. We are managing the state in the **App** component and passing the state values to **Topbar** as props, as shown in the screenshot given in [figure 9.14](#):

```
App.js
src > JS App.js > (el) default
1 import React from 'react';
2 import Topbar from './Topbar/Topbar';
3 import Homepage from './Homepage/Homepage';
4
5 import classes from './App.module.css';
6
7 class App extends React.Component {
8   state = {
9     totalLikes: 0,
10   }
11
12   updateLikes = (incrementValue) => {
13     const updatedVal = this.state.totalLikes +
14       incrementValue;
15     this.setState({totalLikes: updatedVal})
16   }
17
18   render() {
19     return (
20       <div className={classes.App}>
21         <Topbar likesCount={this.state.totalLikes}>
22         />
23         <Homepage updateLikes={this.updateLikes} />
24       </div>
25     );
26   }
27 }
28
29 export default App;
```

```
Topbar.js
src > Topbar > JS Topbar.js > (el) Topbar
1 import React from 'react';
2 import classes from './Topbar.module.css';
3
4 const Topbar = (props) => {
5   return (
6     <div className={classes.Topbar}>
7       <h2>Total Likes: {props.likesCount}</h2>
8     </div>
9   );
10 }
11
12 export default Topbar;
```

Figure 9.14: Passing state variables as props

Now that we are using Redux, we can remove the state management from the **App** component, as shown in the screenshot given in [figure 9.15](#):

```
App.js
src > JS App.js > (el) default
1 import React from 'react';
2 import Topbar from './Topbar/Topbar';
3 import Homepage from './Homepage/Homepage';
4
5 import classes from './App.module.css';
6
7 class App extends React.Component {
8   render() {
9     return (
10       <div className={classes.App}>
11         <Topbar />
12         <Homepage />
13       </div>
14     );
15   }
16 }
17
18 export default App;
```

Figure 9.15: Remove state variables from the App component

Let us update the **Topbar** component to access the likes data from the global state, as shown in the screenshot as follows:

```
index.js M X
src > JS index.js > ...
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3 import {createStore} from 'redux';
4 import {Provider} from 'react-redux';
5
6 import './index.css';
7 import App from './App';
8 import * as serviceWorker from './serviceWorker';
9
10 const initialState = {
11   likesCounter: 99,
12 }
13 const mainReducer = (state=initialState, action)
14 => {
15   return state;
16 }
17 let globalStore = createStore(mainReducer);
18
19 ReactDOM.render(
20   <React.StrictMode>
21     <Provider store={globalStore}>
22       <App />
23     </Provider>
24   </React.StrictMode>,
25   document.getElementById('root')
26 );
27
```

```
Topbar.js M X
src > Topbar > JS Topbar.js > Topbar
1 import React from 'react';
2 import { connect } from 'react-redux';
3 import classes from './Topbar.module.css';
4
5 const Topbar = (props) => {
6   return (
7     <div className={classes.Topbar}>
8       <h2>Total Likes: {props.totalLikes}</h2>
9     </div>
10   );
11 }
12
13 const mapGlobalStateToProps = (globalState) => {
14   return {
15     totalLikes: globalState.likesCounter
16   }
17 }
18
19 export default connect(mapGlobalStateToProps)
(Topbar);
```

Figure 9.16: Access state values from the global store

As we can see in the preceding screenshot, we are using the `connect()` method to access the required global state variable, which in this case, is the `likesCounter`. The `mapGlobalStateToProps` function returns an object that contains the values from the global state. The key-value pairs from this object are passed to the component as a prop. In the preceding example, we are returning the object with “`totalLikes`” as a key, which is available as a prop to be used in the component.

Normally, we would move the reducer to a separate file, as shown in the following screenshot:

```

JS index.js M ...
src > JS index.js > ...
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3 import {createStore} from 'redux';
4 import {Provider} from 'react-redux';
5
6 import './index.css';
7 import App from './App';
8 import * as serviceWorker from './serviceWorker';
9 import { mainReducer } from './reducer';
10
11 let globalStore = createStore(mainReducer);
12
13 ReactDOM.render(
14   <React.StrictMode>
15     <Provider store={globalStore}>
16       <App />
17     </Provider>
18   </React.StrictMode>,
19   document.getElementById('root')
20 );
21
22 // If you want your app to work offline and load
23 // faster, you can change
24 // unregister() to register() below. Note this
25 // comes with some pitfalls.
26 // Learn more about service workers: https://bit.
27 ly/CRA-PWA
...
JS reducer.js U ...
src > JS reducer.js > [0] initialState > likesCounter
1 const initialState = [
2   likesCounter: 20,
3 ]
4
5 export const mainReducer = (state=initialState,
action) => {
6   return state
7 }

```

Ln 2, Col 21 Spaces: 4 UTF-8 LF (JavaScript Go Live Prettier)

Figure 9.17: Create a separate file for the reducer

Update global store from components

The last part is pending in our counter application. We need to implement the functionality where on button click the `likesCounter` gets updated in the global store.

If you remember, we read earlier that the `connect()` function can accept two arguments. The second argument is used to create action dispatchers. The second argument is supposed to be a callback function that gets access to the `dispatch()` method. This callback function should also return an object, which will contain methods to dispatch actions. These action dispatcher methods are passed to the component as a prop, as shown in the following code snippet:

```

<button onClick={() => props.onIncrementLike(1)}>Increment +
1</button>
<button onClick={() => props.onIncrementLike(10)}>Increment +
10</button>
<button onClick={props.onDecrementLike}>Decrement - 1</button>
const mapDispatchToProps = (dispatch) => {
  return {
    onIncrementLike: (value) => dispatch({type:
ACTION_INCREMENT_LIKE, increaseBy: value}),

```

```

        onDecrementLike: () => dispatch({type:
          ACTION_DECREMENT_LIKE}),
    }
}

export default connect(mapStateToProps, mapDispatchToProps)
(HomePage);

```

As you can see, we have passed the action objects inside the dispatch actions just like we did earlier in the chapter.

Let us update our **HomePage** component to handle the **likeCounter** updates using dispatch, as shown in the screenshot as follows:

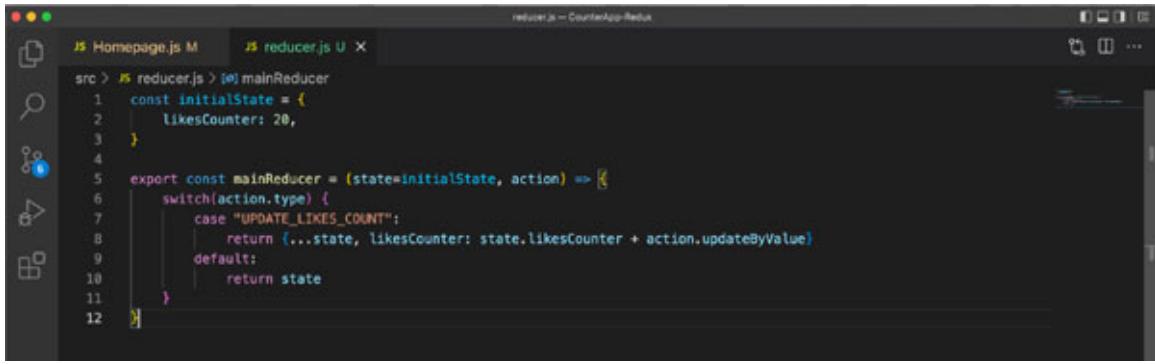
```

JS Homepage.js M X JS reducer.js U
src > HomePage > JS Homepage.js > [e] mapStateToProps
3   import classes from './Homepage.module.css';
4
5   class Homepage extends React.Component {
6     render() {
7       return(
8         <div className={classes.MainContainer}>
9           <section className={classes.First} onClick={() => this.props.updateLikesCount(1)}>Increment +1</
10          section>
11          <section className={classes.Second} onClick={() => this.props.updateLikesCount(20)}>Increment +
12            20</section>
13          <section className={classes.Third} onClick={() => this.props.updateLikesCount(-1)}>Decrement -1</
14            section>
15          <section className={classes.Fourth} onClick={() => this.props.updateLikesCount(-10)}>Decrement -
16            10</section>
17        </div>
18      );
19    }
20
21    const mapDispatchToProps = (dispatch) => {
22      return {
23        updateLikesCount: (value) => dispatch({ type: "UPDATE_LIKES_COUNT", updateByValue: value })
24      };
25    }
26
27    export default connect(null, mapDispatchToProps)(Homepage); //We'll set 1st argument as null because we do not
28    need to access any global state variables in this component.

```

Figure 9.18: Dispatch actions from components

Now when we click on the update buttons, they will dispatch actions to reducer, so we also need to handle the update logic for the “**UPDATE_LIKES_COUNT**” action type, as shown in the screenshot given in [figure 9.19](#):



```
JS Homepage.js M JS reducer.js U X
src > JS reducer.js > [e] mainReducer
  1  const initialState = {
  2    likesCounter: 20,
  3  }
  4
  5  export const mainReducer = (state=initialState, action) => [
  6    switch(action.type) {
  7      case "UPDATE_LIKES_COUNT":
  8        return {...state, likesCounter: state.likesCounter + action.updateByValue}
  9      default:
10        return state
11    }
12 ]
```

Figure 9.19: Handle actions in reducer

Handle multiple reducers

Right now, our application is pretty simple, but when you are working with real production-level applications, you will need to handle a lot of data in the global state, which results in pretty complex and large reducers. To avoid this, we split reducers into smaller and simpler functions.

Let us give it a try, and we will add another counter for dislikes in the current reducer, topbar, and two buttons to increment and decrement the dislikes count, as shown in the following screenshot:

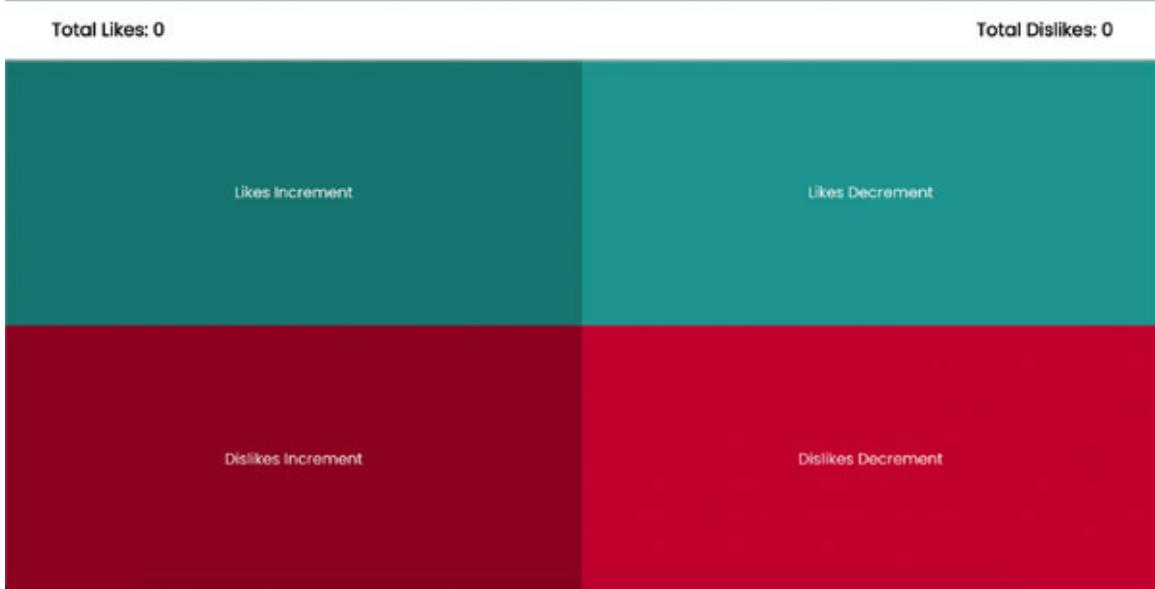
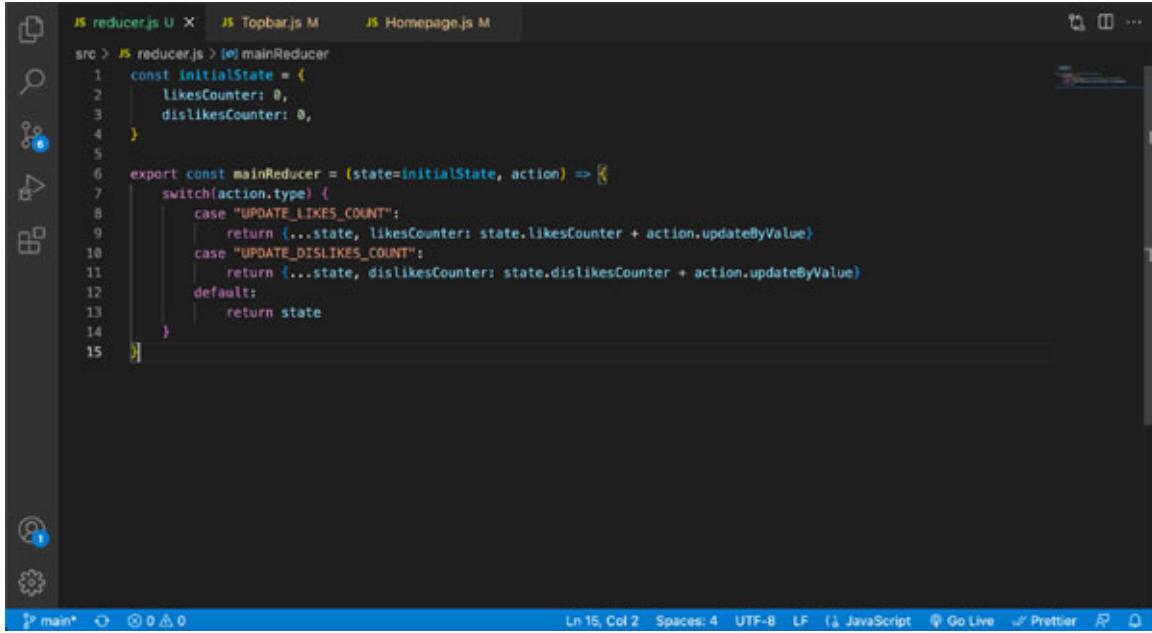


Figure 9.20: Dislike counter

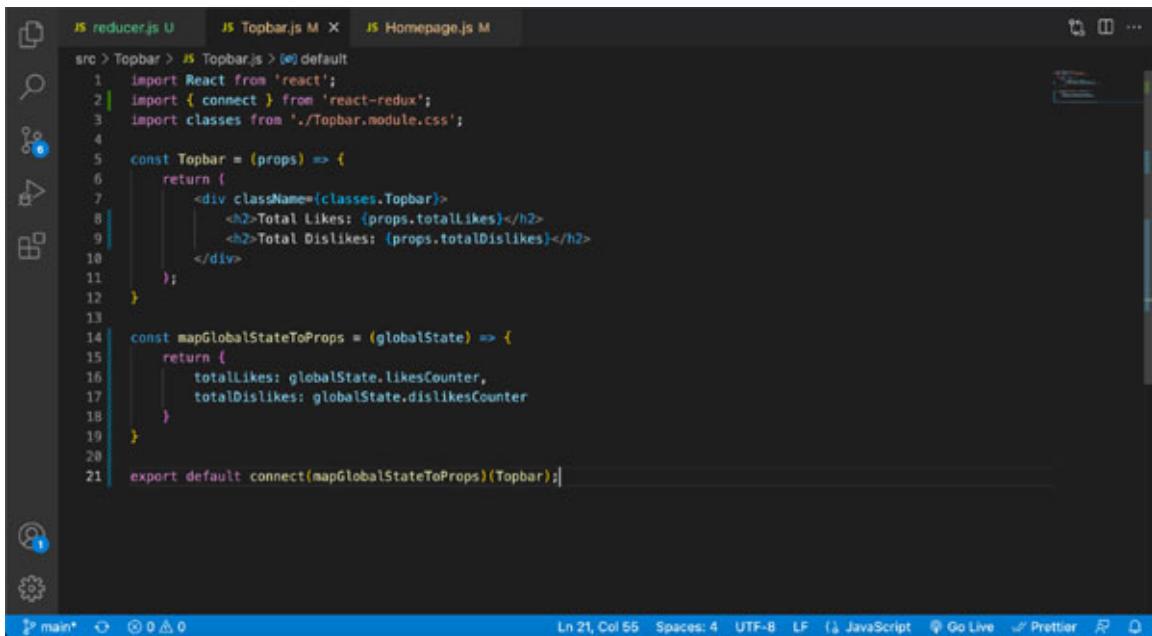
We will need to update the reducer to also manage the `dislikesCounter` in the global store, as shown in the screenshot (refer to [figure 9.21](#)):



```
JS reducer.js U X JS Topbar.js M JS Homepage.js M
src > JS reducer.js > (E) mainReducer
1 const initialState = {
2   likesCounter: 0,
3   dislikesCounter: 0,
4 }
5
6 export const mainReducer = (state=initialState, action) => [
7   switch(action.type) {
8     case "UPDATE_LIKES_COUNT":
9       return {...state, likesCounter: state.likesCounter + action.updateByValue}
10    case "UPDATE_DISLIKES_COUNT":
11      return {...state, dislikesCounter: state.dislikesCounter + action.updateByValue}
12    default:
13      return state
14  }
15]
```

Figure 9.21: Handle dislike action in reducer

We need to update the topbar component to read the dislike count from the global store, as shown in the screenshot as follows:



```
JS reducer.js U JS Topbar.js M JS Homepage.js M
src > Topbar > JS Topbar.js > (E) default
1 import React from 'react';
2 import { connect } from 'react-redux';
3 import classes from './Topbar.module.css';
4
5 const Topbar = (props) => {
6   return (
7     <div className={classes.Topbar}>
8       <h2>Total Likes: {props.totalLikes}</h2>
9       <h2>Total Dislikes: {props.totalDislikes}</h2>
10    </div>
11  );
12
13 const mapGlobalStateToProps = (globalState) => {
14   return {
15     totalLikes: globalState.likesCounter,
16     totalDislikes: globalState.dislikesCounter
17   }
18 }
19
20 export default connect(mapGlobalStateToProps)(Topbar);
```

Figure 9.22: Read dislike state variable from the global store

We also need to update the **Homepage** component to dispatch actions to update the **dislikesCounter** in the global store, as shown in the screenshot given in [figure 9.23](#):

```
JS reducer.js U JS Topbar.js M JS Homepage.js M X
src > Homepage > JS Homepage.js > ...
5  class Homepage extends React.Component {
6    render() {
7      return(
8        <div className={classes.MainContainer}>
9          <section className={classes.First} onClick={() => this.props.updateLikesCount(1)}>Likes Increment</
10         section>
11         <section className={classes.Second} onClick={() => this.props.updateLikesCount(-1)}>Likes
12         Decrement</section>
13         <section className={classes.Third} onClick={() => this.props.updateDislikesCount(1)}>Dislikes
14         Increment</section>
15         <section className={classes.Fourth} onClick={() => this.props.updateDislikesCount(-1)}>Dislikes
16         Decrement</section>
17       </div>
18     );
19   }
20
21   const mapDispatchToProps = (dispatch) => {
22     return {
23       updateLikesCount: (value) => dispatch({ type: "UPDATE_LIKES_COUNT", updateByValue: value }),
24       updateDislikesCount: (value) => dispatch({ type: "UPDATE_DISLIKES_COUNT", updateByValue: value })
25     }
26   }
27
28   export default connect(null, mapDispatchToProps)(Homepage); //We'll set 1st argument as null because we do not
29   need to access any global state variables in this component.

```

Figure 9.23: Dispatch dislike actions from component

Now, let us handle like and dislike state variables in separate reducers.

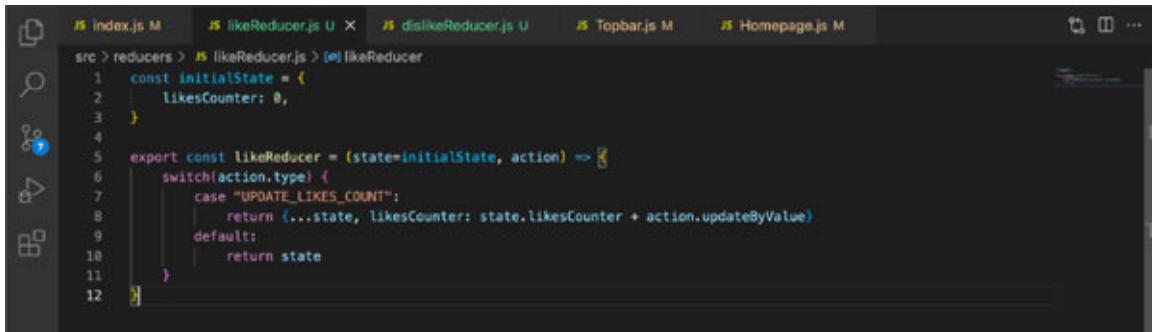
First, let us create a new reducer to handle the dislike state variable, as shown in the screenshot that follows:

```
JS index.js M JS likeReducer.js U JS dislikeReducer.js U X JS Topbar.js M JS Homepage.js M X
src > reducers > JS dislikeReducer.js > dislikeReducer
1  const initialState = {
2   dislikesModuleCounter: 0,
3 }
4
5  export const dislikeReducer = (state=initialState, action) => {
6    switch(action.type) {
7      case "UPDATE_DISLIKES_COUNT":
8        return {...state, dislikesModuleCounter: state.dislikesModuleCounter + action.updateByValue}
9      default:
10        return state
11    }
12  }

```

Figure 9.24: Dislike reducer

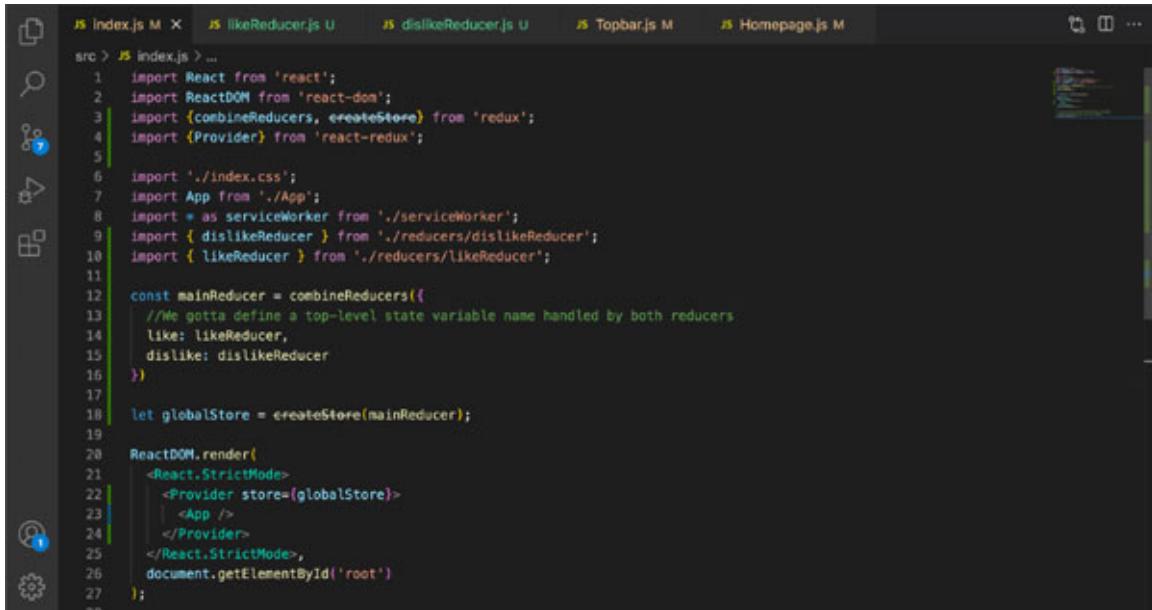
We also need to refactor the current reducer to only handle the likesCounter state variable, as shown in the following screenshot:



```
src > reducers > likeReducer.js > likeReducer
1 const initialState = {
2   likesCounter: 0,
3 }
4
5 export const likeReducer = (state=initialState, action) => [
6   switch(action.type) {
7     case "UPDATE_LIKES_COUNT":
8       return {...state, likesCounter: state.likesCounter + action.updateByValue}
9     default:
10       return state
11   }
12 ]
```

Figure 9.25: Like reducer

Now that we have two different reducers, we need to combine them and then pass them to the global store, as shown in the screenshot ([figure 9.26](#)):



```
src > index.js > ...
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3 import {combineReducers, createStore} from 'redux';
4 import {Provider} from 'react-redux';
5
6 import './index.css';
7 import App from './App';
8 import * as serviceWorker from './serviceWorker';
9 import { dislikeReducer } from './reducers/dislikeReducer';
10 import { likeReducer } from './reducers/likeReducer';
11
12 const mainReducer = combineReducers({
13   //We gotta define a top-level state variable name handled by both reducers
14   like: likeReducer,
15   dislike: dislikeReducer
16 });
17
18 let globalStore = createStore(mainReducer);
19
20 ReactDOM.render(
21   <React.StrictMode>
22     <Provider store={globalStore}>
23       <App />
24     </Provider>
25   </React.StrictMode>,
26   document.getElementById('root')
27 );
```

Figure 9.26: Combine multiple reducers

We use the `combineReducers()` method provided by redux to combine two or more reducers. It accepts an object where we pass individual reducers. This changes the overall structure of the global store, as shown in the following code snippet:

Previous global store:

```
{
  "likesCounter": 0,
  "dislikesCounter": 0
}
```

Updated global store:

```
{
  like: {
    "likesCounter": 0
  },
  dislike: {
    "dislikesCounter": 0
  }
}
```

This updated structure will affect all the components which were subscribed (accessing the global state variables) to the global store. In our example, it is only the topbar component so let's refactor it as shown on Lines 16 and 17 in the following screenshot:

```

src > Topbar > JS Topbar.js > [e] default
  1 import React from 'react';
  2 | import { connect } from 'react-redux';
  3 | import classes from './Topbar.module.css';
  4 |
  5 const Topbar = (props) => {
  6   return (
  7     <div className={classes.Topbar}>
  8       <h2>Total Likes: {props.totalLikes}</h2>
  9       <h2>Total Dislikes: {props.totalDislikes}</h2>
 10     </div>
 11   );
 12 }
 13
 14 const mapStateToProps = (globalState) => {
 15   return {
 16     totalLikes: globalState.like.likesCounter,
 17     totalDislikes: globalState.dislike.dislikesCounter
 18   }
 19 }
 20
 21 export default connect(mapStateToProps)(Topbar);

```

Figure 9.27: Handle state after combining reducers

We do not have to make any changes to the action dispatches, and it remains the same.

Implementing Redux using Hooks

With the release of React Hooks, the latest version of React-redux has provided us with two inbuilt hooks to access the global store and to dispatch actions—**useSelector()** and **useDispatch()**.

useSelector() is very similar to **mapStateToProps**. It gets access to the global state, and we can access the global state value we need. Here is a

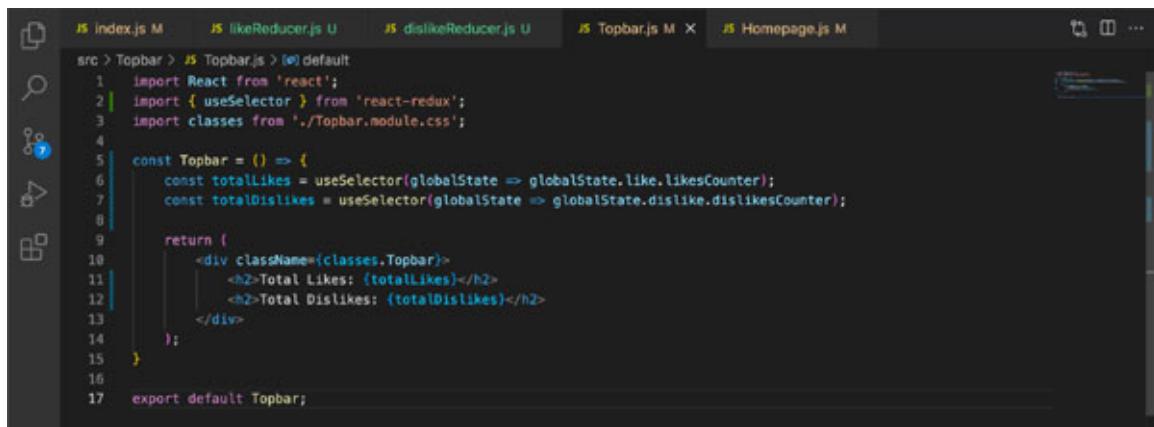
code snippet to help you understand better:

```
import { useSelector } from "react-redux";
const totalLikes = useSelector(globalState =>
globalState.like.likesCounter);
const totalDislikes = useSelector(globalState =>
globalState.dislike.dislikesCounter);
```

useDispatch() is a replacement for **mapDispatchToProps**. It directly returns your store's dispatch method so you can manually dispatch actions. This change is highly likeable, as binding action creators can be a little confusing to newcomers to React Redux. Here is a code snippet for you:

```
import { useDispatch } from "react-redux";
const dispatch = useDispatch();
<someElement someEvent={() => dispatch({type:
"UPDATE_LIKES_COUNT", value: someValue})}>
```

Let us try to update our **Topbar** component by using these Hooks, as shown in the screenshot that follows:

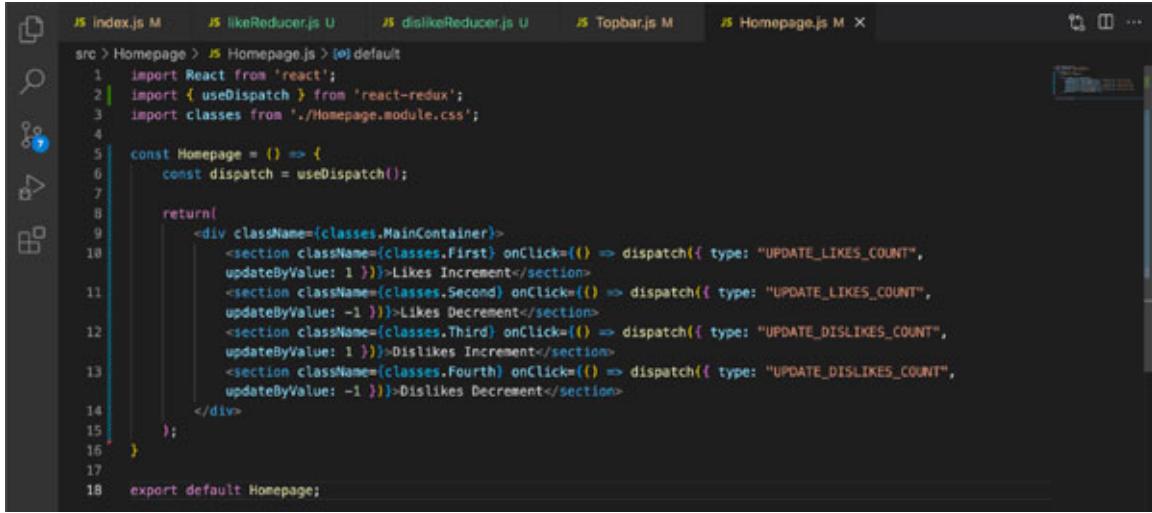


```
JS index.js M JS likeReducer.js U JS dislikeReducer.js U JS Topbar.js M X JS Homepage.js M
src > Topbar > JS Topbar.js > (e) default
1 import React from 'react';
2 import { useSelector } from 'react-redux';
3 import classes from './Topbar.module.css';
4
5 const Topbar = () => {
6   const totalLikes = useSelector(globalState => globalState.like.likesCounter);
7   const totalDislikes = useSelector(globalState => globalState.dislike.dislikesCounter);
8
9   return (
10     <div className={classes.Topbar}>
11       <h2>Total Likes: {totalLikes}</h2>
12       <h2>Total Dislikes: {totalDislikes}</h2>
13     </div>
14   );
15 }
16
17 export default Topbar;
```

Figure 9.28: Redux using Hooks

Notice we do not use props anymore. We can directly use the values because they are variables.

Let us also update the **Homepage** component to use hooks as shown in the screenshot provided in [figure 9.29](#):



```
JS index.js M JS likeReducer.js U JS dislikeReducer.js U JS Topbar.js M JS Homepage.js M X
src > Homepage > JS Homepage.js > [0] default
  1 import React from 'react';
  2 import { useDispatch } from 'react-redux';
  3 import classes from './Homepage.module.css';
  4
  5 const Homepage = () => {
  6   const dispatch = useDispatch();
  7
  8   return (
  9     <div className={classes.MainContainer}>
 10       <section className={classes.First} onClick={() => dispatch({ type: "UPDATE_LIKES_COUNT", updateByValue: 1 })}>Likes Increment</section>
 11       <section className={classes.Second} onClick={() => dispatch({ type: "UPDATE_LIKES_COUNT", updateByValue: -1 })}>Likes Decrement</section>
 12       <section className={classes.Third} onClick={() => dispatch({ type: "UPDATE_DISLIKES_COUNT", updateByValue: 1 })}>Dislikes Increment</section>
 13       <section className={classes.Fourth} onClick={() => dispatch({ type: "UPDATE_DISLIKES_COUNT", updateByValue: -1 })}>Dislikes Decrement</section>
 14     </div>
 15   );
 16 }
 17
 18 export default Homepage;
```

Figure 9.29: Dispatch actions using hooks

This must have given you some clarity on how we can use redux to manage the application state in the global store.

Action creators

When we create production-level applications, we do not call dispatch methods and pass action objects throughout the codebase. We create something called action creators. Action creator is just a function that returns an action, as shown in the following code snippet:

Old Code:

```
const dispatch = useDispatch();
<section className={classes.First} onClick={() => dispatch({
  type: "UPDATE_LIKES_COUNT", updateByValue: 1 })}>Likes
  Increment</section>
```

New Code:

```
//Create a new folder "actions" add an index.js file inside it.
In a new file where we will have the action creators
const updateLikeCount = (value) => {
  return {
    type: "UPDATE_LIKES_COUNT",
    updateByValue: value
  }
}
import { updateLikeCount } from '../some-path/actions/';
const dispatch = useDispatch();
```

```
<section className={classes.First} onClick={() =>  
  dispatch(updateLikeCount(1))}>Likes Increment</section>
```

The biggest benefit of using action creators is that all the action-related logic is in a single file, and you can call the functions from across your application.

Async actions using middleware

So far, we have learned that redux only works for synchronous actions. The reducer is a function that updates the global store in a synchronous fashion.

But what about handling API responses in the global store? Is that possible if the reducer only handles synchronous actions? The answer is no and yes. It does not handle asynchronous requests out-of-the-box. But, we can add a middleware like Redux-thunk or Redux-saga to make it possible.

The middleware acts as a middle layer between your application and redux. Normally, we would call action creators, and they would return the actions, which would directly go to the reducer, and the state would get updated. But, in the case of API requests (or other side-effects), there is some unknown delay time before the actions can be passed to the reducer. This is where the middleware does magic. It allows action creators to return a function instead of actions. Every time an action is dispatched, the middleware checks to see if it is an object or a function. If it is an object, then it is directly passed on to the reducer. If it is a function, then the middleware calls the function, performs the side-effect, and then dispatches actions to the reducer. If the action is a function, then Redux will pass two arguments—**dispatch()** and **getState()**. The **dispatch()** so that they can dispatch new actions if they need to, and **getState()**, so they can access the current state.

When we add a Middleware, the redux gets updated as shown in the diagram given in [figure 9.30](#):

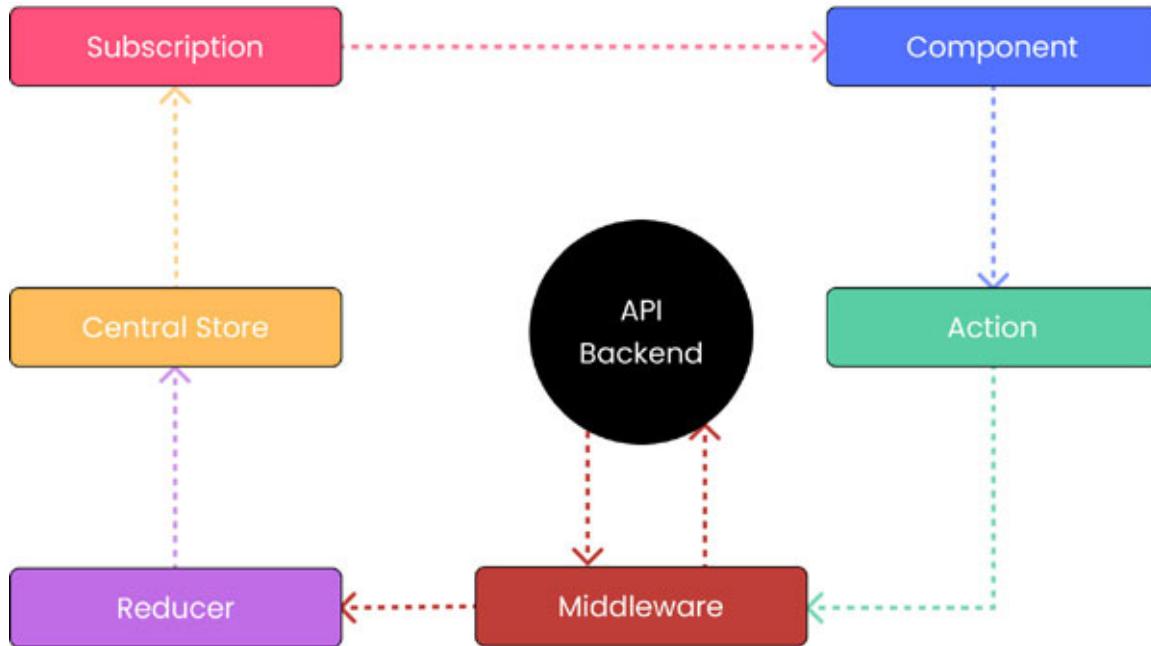
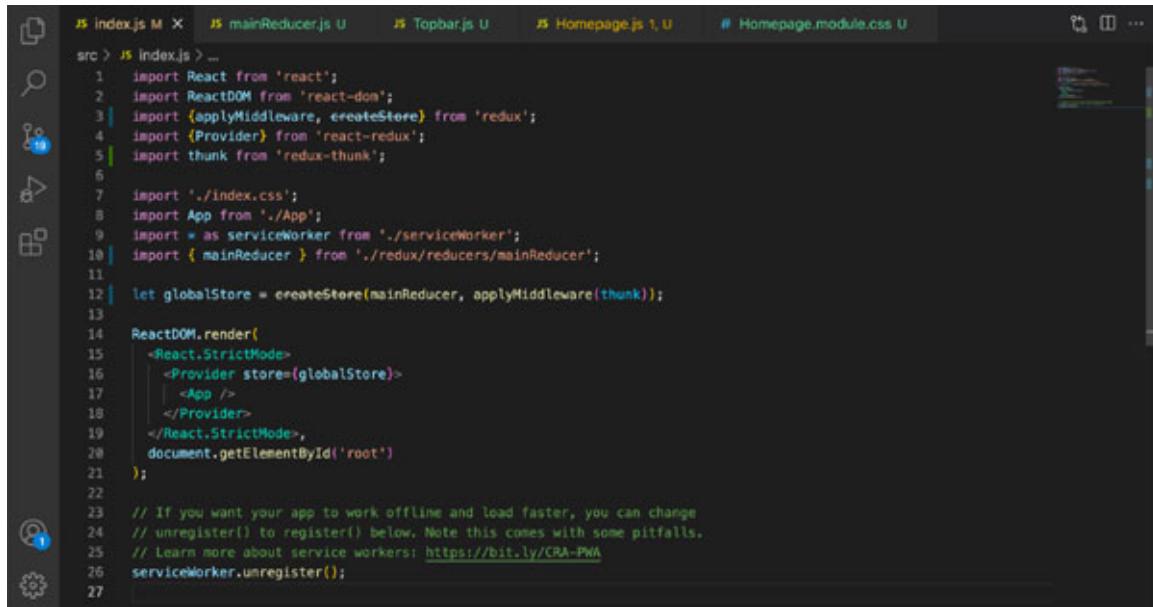


Figure 9.30: Redux flow with Middleware

Let us give this a try. We will use Redux Thunk for this demonstration, but first, we have to install it. Please follow these steps to install and configure the react-thunk package:

1. Open the terminal or command prompt.
2. Navigate to the project folder.
3. Run the following command, “`npm i redux-thunk`”.
4. Wait for the installation to complete.
5. Open `package.json` to verify if it is installed.
6. We need to pass thunk as a middleware to the `createStore()` method, as shown in the screenshot in [figure 9.31](#):



```
src > JS index.js < ...  
1 import React from 'react';  
2 import ReactDOM from 'react-dom';  
3 import { applyMiddleware, createStore } from 'redux';  
4 import { Provider } from 'react-redux';  
5 import thunk from 'redux-thunk';  
6  
7 import './index.css';  
8 import App from './App';  
9 import * as serviceWorker from './serviceWorker';  
10 import { mainReducer } from './redux/reducers/mainReducer';  
11  
12 let globalStore = createStore(mainReducer, applyMiddleware(thunk));  
13  
14 ReactDOM.render(  
15   <React.StrictMode>  
16     <Provider store={globalStore}>  
17       <App />  
18     </Provider>  
19   </React.StrictMode>,  
20   document.getElementById('root')  
21 );  
22  
23 // If you want your app to work offline and load faster, you can change  
24 // unregister() to register() below. Note this comes with some pitfalls.  
25 // Learn more about service workers: https://bit.ly/CRA-PWA  
26 serviceWorker.unregister();  
27
```

Figure 9.31: Initialize Middleware

We have installed and configured the Redux-thunk package. Now, let us update our **actionCreators** to send an API request.

We will create a simple application where our **Homepage** component will load a list of users and topbar to show the count of users fetched from the backend, as shown in the screenshot as follows:

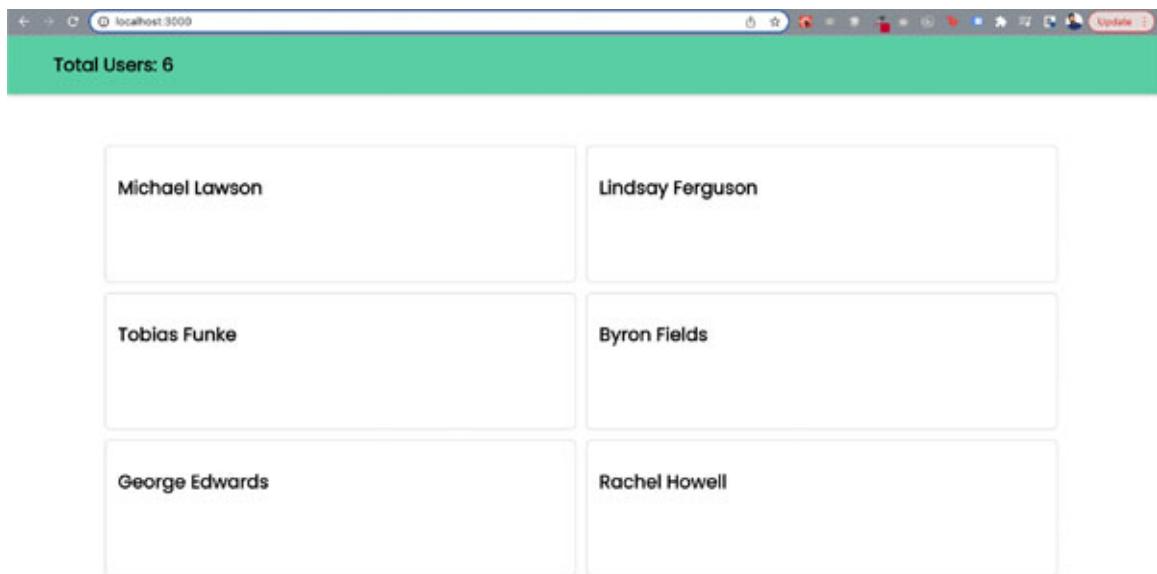
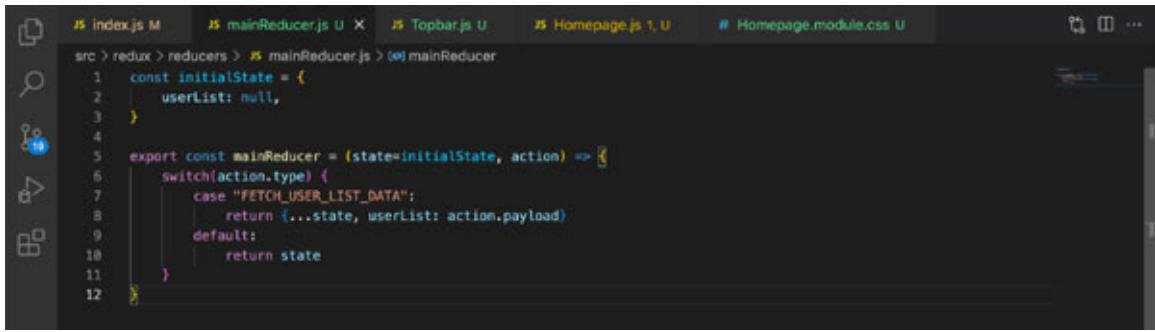


Figure 9.32: User listing page

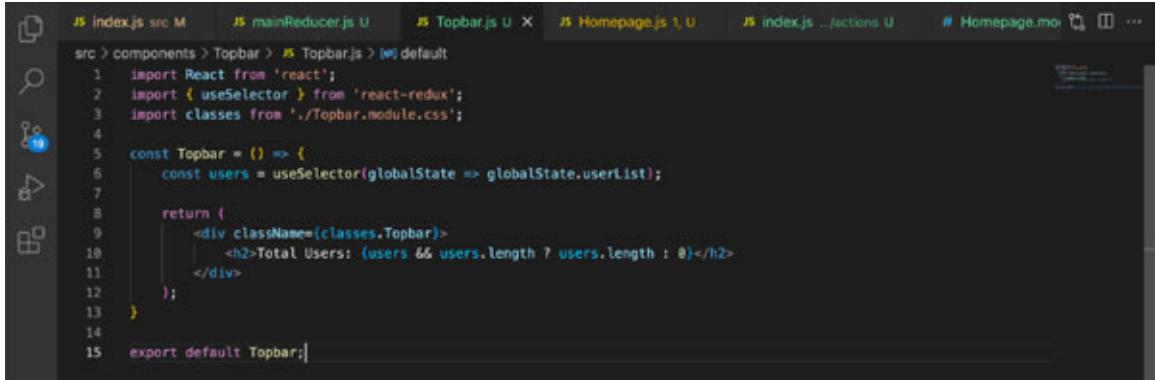
We need to setup the reducer to manage user-related data, as shown in the following screenshot:



```
src > redux > reducers > mainReducer.js > mainReducer
1 const initialState = {
2   userList: null,
3 }
4
5 export const mainReducer = (state=initialState, action) => {
6   switch(action.type) {
7     case "FETCH_USER_LIST_DATA":
8       return {...state, userList: action.payload}
9     default:
10      return state
11   }
12 }
```

Figure 9.33: Reducer to handle fetch action

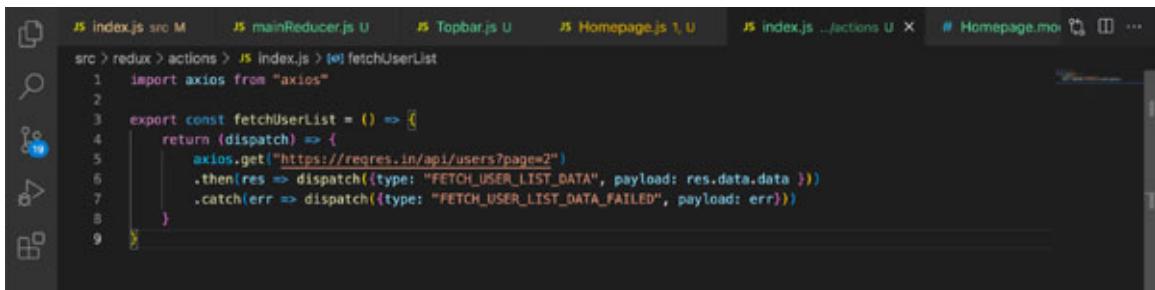
We need to Topbar to select the `userList` global state value as shown in the following screenshot (refer to [figure 9.34](#)):



```
src > components > Topbar > Topbar.js > default
1 import React from 'react';
2 import { useSelector } from 'react-redux';
3 import classes from './Topbar.module.css';
4
5 const Topbar = () => {
6   const users = useSelector(globalState => globalState.userList);
7
8   return (
9     <div className={classes.Topbar}>
10     <h2>Total Users: {users && users.length ? users.length : 0}</h2>
11   );
12 }
13
14 export default Topbar;
```

Figure 9.34: Access user's global state variable

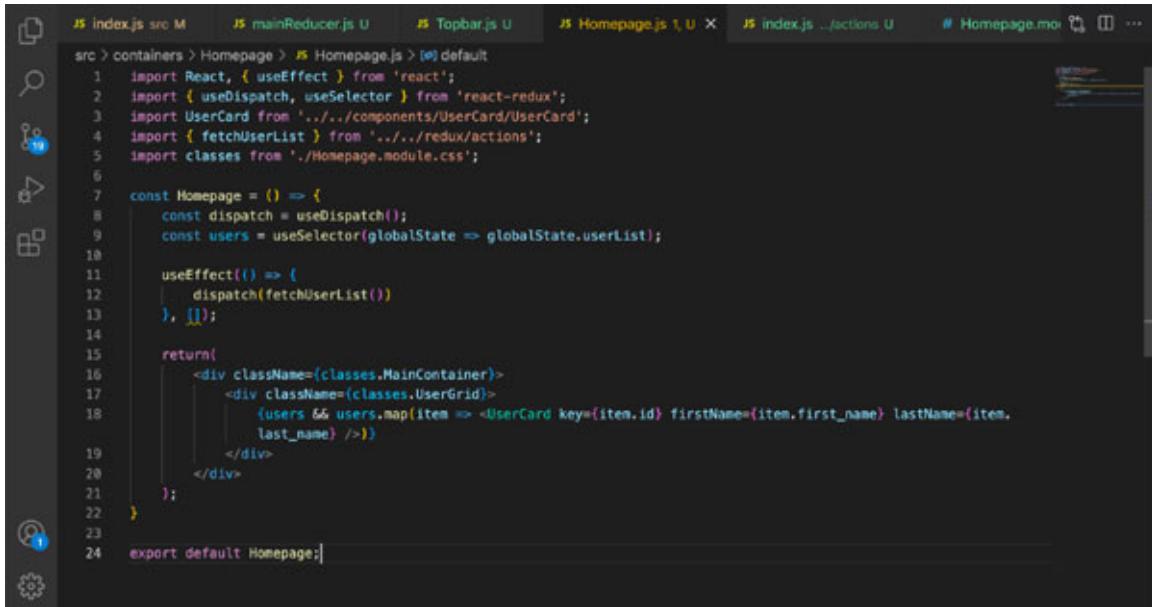
We also need to set up an action creator to send an API request and dispatch actions when we receive the response, as shown in the screenshot as follows:



```
src > redux > actions > index.js > fetchUserList
1 import axios from "axios"
2
3 export const fetchUserList = () => {
4   return (dispatch) => {
5     axios.get("https://regres.in/api/users?page=2")
6       .then(res => dispatch({type: "FETCH_USER_LIST_DATA", payload: res.data.data}))
7       .catch(err => dispatch({type: "FETCH_USER_LIST_DATA_FAILED", payload: err}))
8   }
9 }
```

Figure 9.35: Setup action creator

We can also dispatch actions to update the loading status and errors in the preceding action creator and reducer, but we would not do this in this example. The next step is to call the action creator on the homepage load and select the required global state variables, as shown in the following screenshot:



```
src > containers > Homepage > Homepage.js > default
  1 import React, { useEffect } from 'react';
  2 import { useDispatch, useSelector } from 'react-redux';
  3 import UserCard from './components/UserCard/UserCard';
  4 import { fetchUserList } from '../redux/actions';
  5 import classes from './Homepage.module.css';
  6
  7 const Homepage = () => {
  8   const dispatch = useDispatch();
  9   const users = useSelector(globalState => globalState.userList);
 10
 11   useEffect(() => {
 12     dispatch(fetchUserList())
 13   }, []);
 14
 15   return(
 16     <div className={classes.MainContainer}>
 17       <div className={classes.Usergrid}>
 18         {users && users.map(item => <UserCard key={item.id} firstName={item.first_name} lastName={item.last_name} />)
 19       </div>
 20     </div>
 21   );
 22 }
 23
 24 export default Homepage;
```

Figure 9.36: Dispatch action to fetch users

That was all the required changes to also handle asynchronous actions using middleware.

Conclusion

React provides us with an inbuilt mechanism for state management. It gets really complex and difficult to manage the state using the inbuilt mechanism when we build large applications where the state variable gets updated from multiple components and are required in multiple components. To solve this problem, we can use Redux. It allows us to manage the state in a global store. Now, you might think, why not manage the entire application state using redux? Well, that is not recommended. You manage the shared state using redux, and the state that is only required in the component should be managed in the component itself.

In the upcoming chapter, we will learn about the creation of production builds, and we will also learn about hosting React applications.

Questions

1. Why do we need Redux?
2. What are the four main concepts in Redux? Explain with a diagram.
3. How do we create asynchronous actions?

4. Why do we need multiple reducers? How do we configure it?
5. Why do we need action creators? How do we configure them?

Multiple choice questions with answers

- 1. Which is not a Redux concept?**
 - a. Store.
 - b. Reducer.
 - c. Flipper.
 - d. Action.
- 2. Which package do we use to connect redux with react application?**
 - a. react-router
 - b. react-redux-connector
 - c. redux-connect
 - d. react-redux
- 3. What is the data flow in Redux?**
 - a. action -> reducer -> store -> component
 - b. store -> reducer -> action -> component
 - c. component -> reducer -> action -> store
 - d. action -> store -> reducer -> component
- 4. Which function is used to select the value from the global store using the non-Hook function?**
 - a. useSelector
 - b. provider
 - c. connect
 - d. applyMiddleware
- 5. Which function is used to select the value from the global store using the Hook function?**
 - a. useSelector
 - b. provider

- c. connect
- d. applyMiddleware

Answers

Questio n	Correct Answer
1	c. Flipper
2	d. react-redux
3	a. action -> reducer -> store -> component
4	c. connect
5	a. useSelector

CHAPTER 10

Production Build and Hosting React Apps

In this chapter, we will talk about different tools that work behind the scenes to run our React app. We will see how NPM is used to manage packages. We will learn about Webpack and how it bundles all our JavaScript and CSS files into chunks to help reduce load speed. We will learn how to create a production build of our app and how to host it.

Structure

In this chapter, we will discuss the following topics:

- Babel
- NPM
- Production build
- Host React applications
- Webpack

Objectives

After studying this chapter, you will know how to create a production build of a react application. You will also know how to host a React application.

Production build

There are three build environments provided by default by create-react-app —development, test, and production.

The development build is used for development reasons. This version of our application runs in the local system of developers. Every time we run the command “`npm start`”, it starts the development version of the React

application. If you have installed the React developer tools extension for Chrome, you will be able to see which build of the application you are running, as shown in the following screenshot:

Figure 10.1: React development build message

It would also tell if you were running the production build, as shown in the screenshot that follows:

Figure 10.2: React production build message

This React developer tools extension also tells us if the current website is running on React or not, as shown in the screenshot given in [figure 10.3](#):

Figure 10.3: No react build message

The testing build is used to run the unit test files in the project. You can run the “`npm test`” command to run the test files.

The production build is what we host for our users. This is the version of our code that will run on the user’s browser. It is a minified, compressed, and optimized version of our application. To create a production build, we need to run the following command “`npm run build`”, the production build will be created because we created the project using create-react-app, and all the configurations are already available.

Let us see what happens when we run the “`npm run build`” command.

As you can see in the screenshot provided in [figure 10.4](#), the first thing to notice is the size of the total files is reduced drastically; we are just left with a few KBs. This compression is done because of the gzip compression.

Figure 10.4: Production build logs

A new folder called “`build`” is created. This will contain all the files required to host and run our production application. As you can see as follows, the total size of the project folder is almost 175 MB, and that of the build folder is almost 853 KB (refer to [figure 10.5](#)).

Figure 10.5: Project folder size versus build folder size

If you refer to [figure 10.4](#), you will notice that there are some files with names like “`.chunk.js`”. When we are building the react application, we create a lot of JavaScript files, load numerous packages, and create CSS files. When we create a production build, the Webpack creates these chunks of files for us. It would merge important component files together, and it would merge packages in a separate chunk, and so on. Let us understand the standard format for these files.

main.[hash].chunk.js:

- This is our application code. It includes files like `index.js`, `app.js`, and so on.

[number].[hash].chunk.js:

- These contain code for third-party libraries and packages, which is also known as vendor code.
- It might also contain code-splitting chunks. In the next chapter, we will learn in detail about the concept of code-splitting. In short, it allows us to split the entire `src` folder into smaller parts so that we load only the parts, which are required to render the application for a specific URL.
- Because the vendor chunks are separate, we can have a longer caching policy for them. This also helps to improve performance.

runtime-main.[hash].js:

- This is a small chunk of Webpack runtime logic that is used to load and run our application.

More chunks are created based on our code-splitting setup.

If you check the build folder, you will see that we have a static folder, and inside that, we have the css and js folders. As the structure is pretty self-explanatory, the JavaScript files will be stored inside the js folder, and the CSS files will be stored in the css folder, as shown in the screenshot given in [figure 10.6](#):

Figure 10.6: JS and CSS chunks after production build

If you notice in the file browser, we also have some “`.map`” files. When we create a production build, our code gets minified, uglified, compressed, and so on. These map files hold information regarding the original code, and that is how they help debug the production build code. The following code snippet is not how the code is transformed; this is just to give you some sense of what happens:

Original Code (Development Build):

```
const getFullName = ( fName, lName ) => {  
    return fName + " " + lName  
}
```

Transformed Code (Production Build):

```
const gfn=(fn,ln)=>{return fn+ln}
```

Now, imagine debugging the transformed version of the code.

Host React applications

There are numerous ways to host a React application. The following are some of the most common ones:

- AWS EC2 instance
- Firebase
- Netlify
- Glitch
- Heroku

We will use *netlify* to host our application. Please follow the following steps to host your application:

1. Open a terminal or command prompt based on your operating system.
2. Navigate to the project folder.
3. Create a production build. Run the following command “`npm run build`”.
4. Navigate to the build folder.
5. Run the following command to install netlify-cli “`npm i netlify-cli -g`”. Add sudo if you face permission issues.

6. Run the command “`netlify deploy`” to start the deployment process.
7. It might ask you to login to netlify and authorize the cli. If it gets stuck at “Waiting for authorization...” then open the link mentioned in the terminal yourself to authorize, as shown in the following screenshot:

Figure 10.7: Authorization link

8. It will say that “This folder is not linked to a site yet” and will give you options. Please select the one, which says, “Create and configure a new site”.
9. It will ask to select a team. Please select whichever one you want; most probably, it will just show you one.
10. It will ask you to enter a unique site name for your website. You can skip this step if you want *netlify* to give a random name.
11. It will host your Web app and will show you the admin URL, site URL, and site ID. You can open the site URL to preview your Web app.
12. Just one thing to remember, *netlify* hosting comes with a secured URL (HTTPs). If, in your application, you are using a non-secure URL(HTTP), then your application will not load. Please ensure that all the API requests or image files that you load are coming from a secure URL.

That are all the steps required to host your Web app.

Introduction to Webpack

Webpack is an open-source JavaScript module bundler. It is a module bundler primarily for JavaScript, but it can transform front-end assets such as HTML, CSS, and images if the corresponding plugins are included. It takes modules with dependencies and generates static assets representing those modules.

Create-react-app uses Webpack behind the scenes to create your application bundle.

As you can see in the following figure, it takes all the JavaScript, CSS, and image files and creates an optimized version of the files:

Figure 10.8: Bundling by Webpack

It accepts files from across your application and bundles them in a specified output file, as you can see in the screenshot that follows:

Figure 10.9: JS bundling by Webpack

As shown in the following diagram, it uses different loaders and tools to support the latest functionality on older browsers. For example, it uses a babel-loader to transform ES6 code to ES5 code so that older browsers can understand the code too, and it uses a css-loader to support CSS modules.

Figure 10.10: Loaders in Webpack

It also performs transformations to optimize, minify and compress code, as shown in the diagram given in [figure 10.11](#):

Figure 10.11: Plugins in Webpack

Conclusion

That was all about different environments and builds available in React. By now, you must have understood how to create a production build and host our react application.

In the upcoming chapter, we will learn about the performance optimization of React applications. We will learn how to benchmark the application's performance, how to identify performance issues, and how to fix them.

Questions

1. What are the three environments provided by CRA?
2. How do you create a production build?
3. What are chunks?
4. How do you host a React application?

5. What is Webpack? Why do we need it?

Multiple choice questions with answers

1. **What type of files cannot be optimized by Webpack?**

- a. JavaScript
- b. CSS
- c. Images
- d. Videos

2. **What is the command to create a production build?**

- a. npm build
- b. npm run build
- c. npm prod build
- d. npm production build

3. **Which folder contains all the production files?**

- a. build
- b. output
- c. production
- d. exports

4. **How do you run unit test files?**

- a. npm unit
- b. npm test
- c. npm unit-test
- d. npm testing

5. **The production code is split into multiple chunks?**

- a. True
- b. False

Answers

Question	Correct Answer
1	d. Videos
2	b. npm run build
3	a. build
4	b. npm test
5	a. True

CHAPTER 11

Performance Optimization

Introduction

React applications are extremely fast, but while building these applications, we introduce issues that result in a loss of performance. These issues could be due to bad code, usage of heavy images, unoptimized animations, slow APIs, not splitting the build files, and many more reasons. In this chapter, we will learn about application performance and how to optimize it. There are multiple ways it can be achieved, but first, we need to learn how to measure and benchmark it.

Structure

In this chapter, we will discuss the following topics:

- Measure the performance of React apps using React Profiler
- Using `shouldComponentUpdate` to improve performance
- `PureComponents`
- Memoization: `useMemo()` and `useCallback()`
- Code splitting
- Other tips and tricks to improve performance

Objectives

After studying this chapter, you should be able to use performance benchmarking tools to find performance issues and fix them. You will learn about the `shouldComponentUpdate()` lifecycle method, `useMemo()` hook, and `useCallback()` hook to avoid unnecessary re-rendering in class-based and functional components.

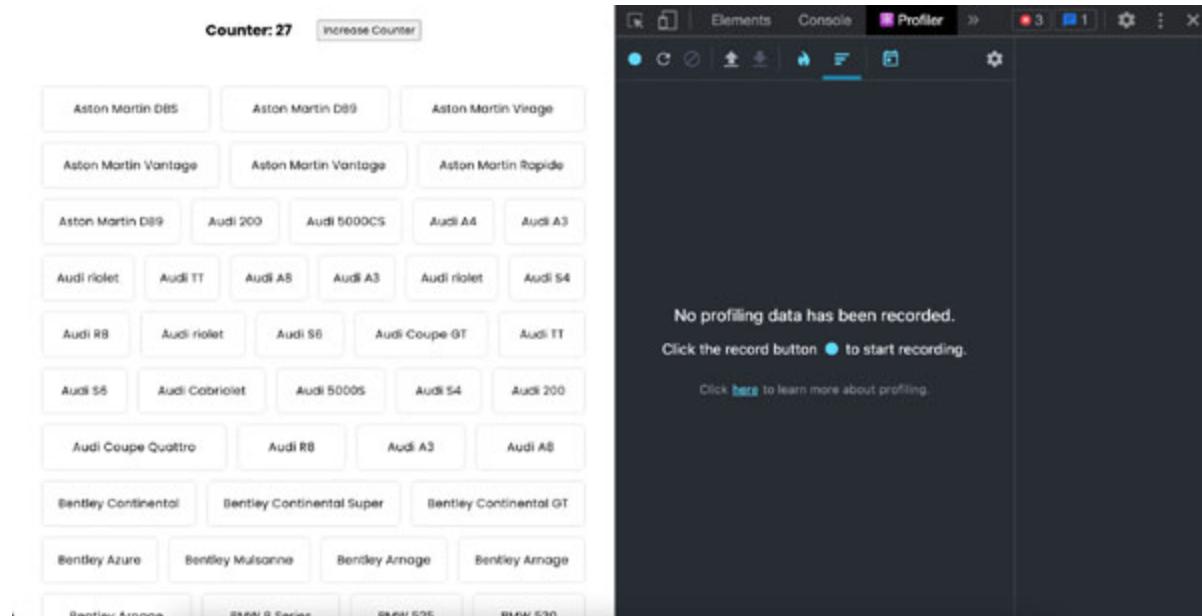
Why measure performance?

Before we can fix something, we need to know what is wrong with it, which is why measurement is so important. There are various tools that are used to measure the performance of a React application. In this chapter, we will learn about React Profiler.

React Profiler

It is a plugin available in React Devtools. It uses React's Profiler API to measure how frequently a React application is rendering or re-rendering. It also captures the frequency and time it takes for each component to render when the application is rendered to identify performance bottlenecks.

You can open the profiler from the browser Devtools, as shown in the screenshot in [figure 11.1](#):



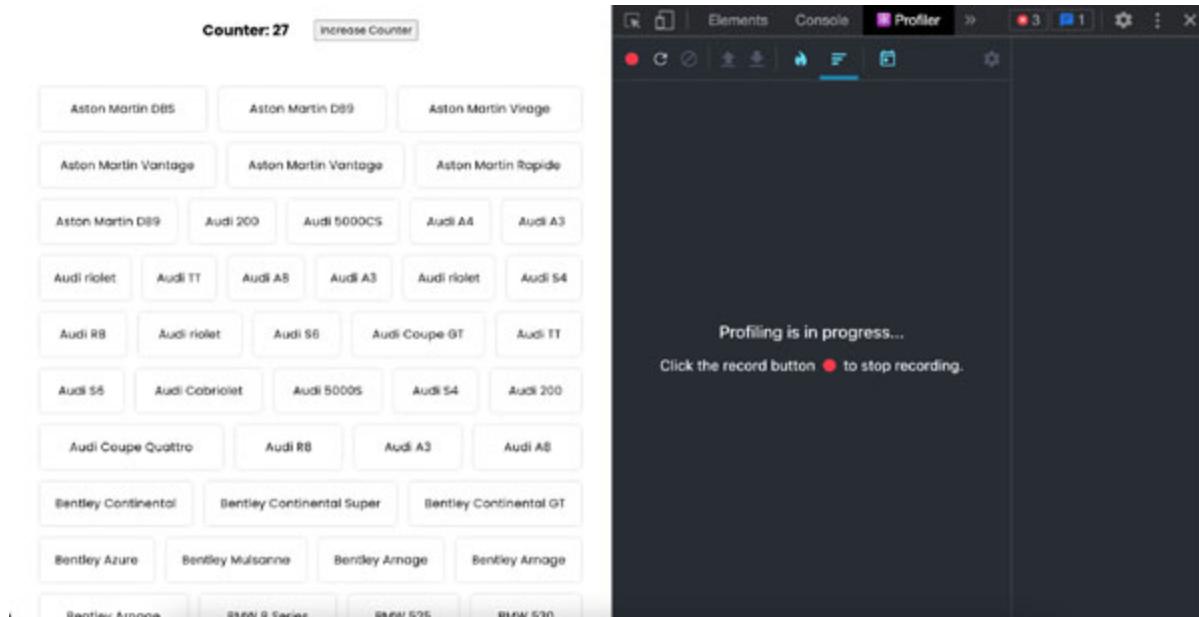


Figure 11.2: Profiling in-progress

Once you are done, click on the red stop button (refer to [figure 11.2](#)), and it will show you the performance data as shown in the screenshot (refer to [figure 11.3](#)):

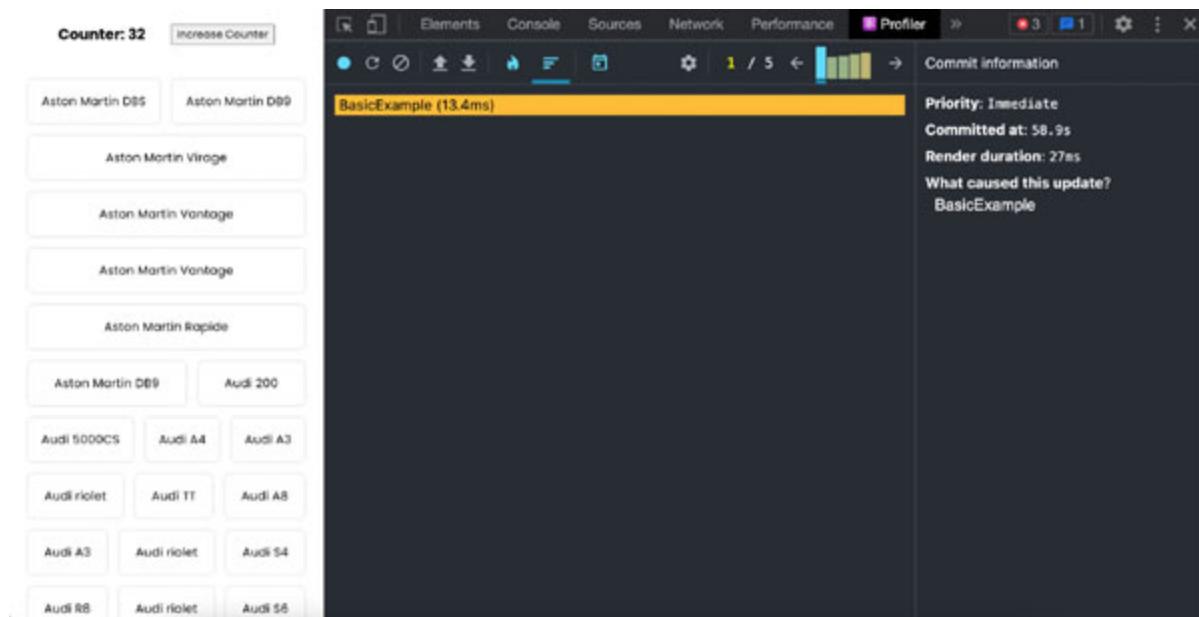


Figure 11.3: Profiling complete

Let us see what all information is provided by the profiler, as shown in the screenshot in [figure 11.4](#):

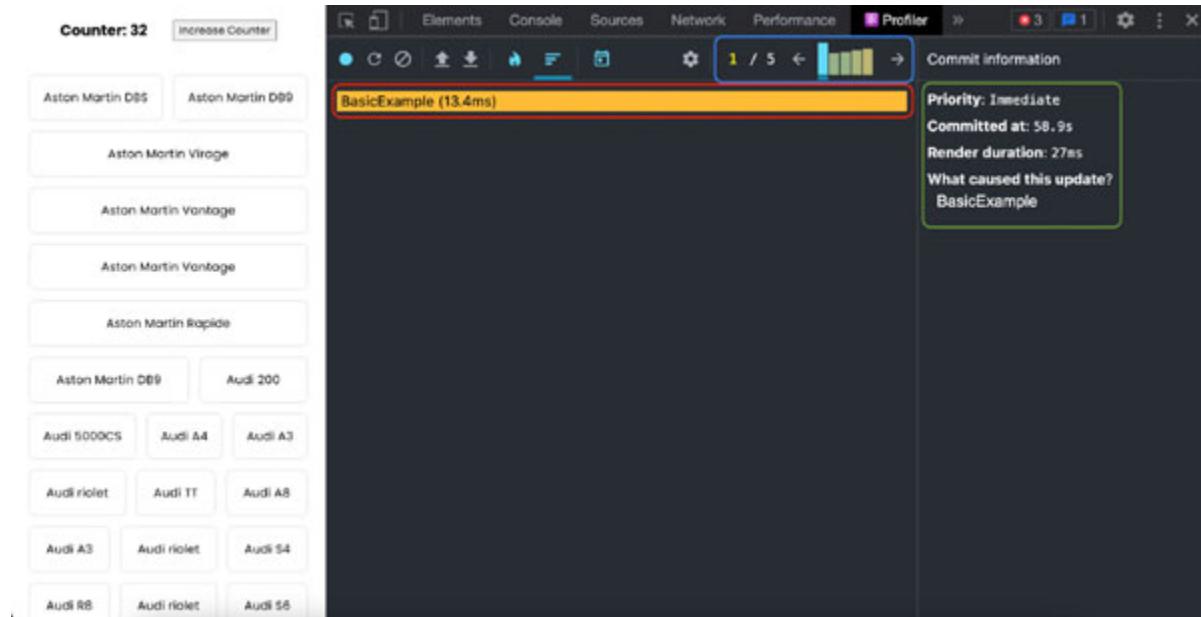


Figure 11.4: Profiling results

React works in two steps—render and commit. During the render phase, it compares virtual DOMs to find the nodes which need to be updated in the browser DOM. This is where the `render()` lifecycle method is called. During the commit phase, the browser DOM is updated. This is where `componentDidMount()` and `componentDidUpdate()` lifecycle methods are called.

The section in the blue box shows the commits recorded during the profiling. “1/5”, tells that there was a total of five commits, and we are seeing the details of the first commit. We also have five bars next to the count. Less height and green bars show that, the commit took less time to complete. More height and yellow bars show that it took more time to complete. So, when you start optimizing the performance, then, start with the tallest and yellowish bars.

The section in red shows the components that were rendered in the commit. The section in green shows the commit information, like how much time it took for the commit to complete, why the element was rendered, and so on. Let us check the profiling results for the mounting of the application, as shown in the figure as follows:

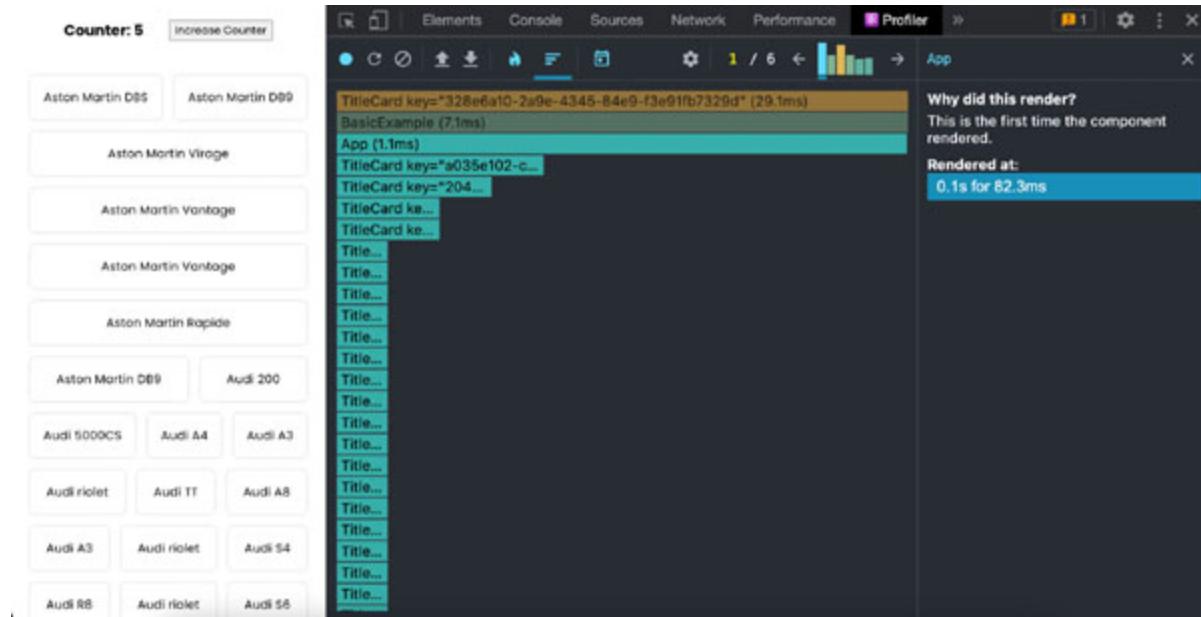


Figure 11.5: Profiling results—app load

Notice, now the total commit count is 6; it is because I reloaded the application and clicked on the “**Increase Counter**” button five times. This resulted in six renderings and commits of the application. The first bar is blue, which means we are viewing the first commit tracked in this profiling session. In the component listing section, the list from the “App” component is highlighted, which means we see the App component rendering details on the right side. It says that this app component was rendered for the first time.

Similarly, let us view the rendering details for one of the “**TitleCard**” components, as shown in the screenshot in [figure 11.6](#):

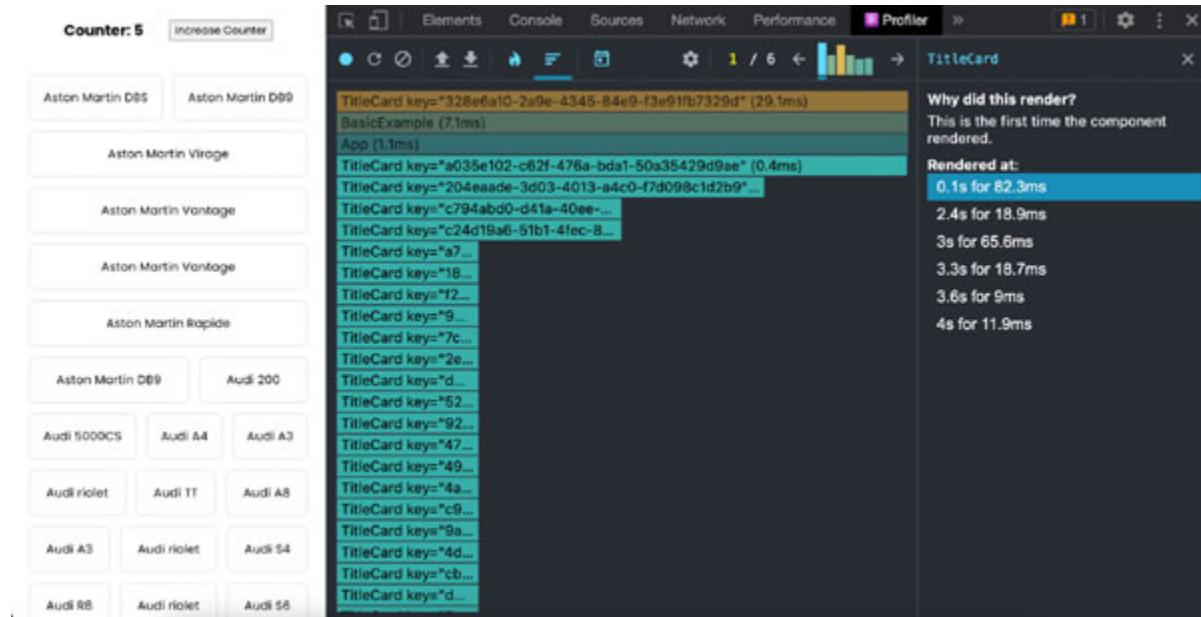


Figure 11.6: Profiling results—`TitleCard` component load

In the component listing section, the list from the first “`TitleCard`” component is highlighted, which means we see the `TitleCard` component rendering details on the right side. It says that this component was rendered a total of six times in this profiling session. We can also get details about each render. As you can see, the first item is highlighted in the preceding screenshot, and it says that the component was rendered for the first time.

Let us have a look at the second time this same `TitleCard` component is rendered. Notice now the second bar is blue, which means we are looking at the second commit. It says that the component was rendered because the parent component was rendered, as shown in the following screenshot:

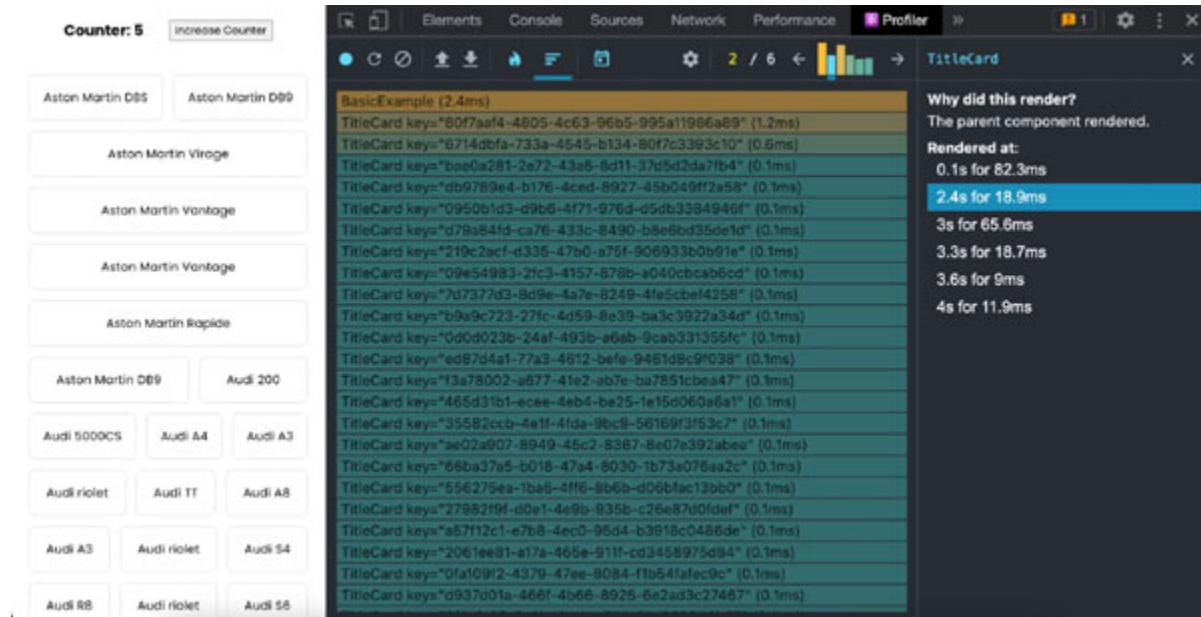


Figure 11.7: Profiling results—TitleCard component load

Let us hold on for a moment and look at why this component was rendered six times. It makes sense for it to render on application load, but it should not be getting rendered when we click on the “**Increase Counter**” button. Every time we click on the button, it is updating the counter, which is required, but it is also re-rendering the entire TitleCard grid. This is a grid with simple cards but imagine a scenario where you had more complex, more element-dense components; it would be such a bloat on the performance.

Hope this gave you some sense of the kind of performance issues we end up introducing by writing bad code.

How do we fix this?

One option is to extract the entire counter-related code to a separate component. It would work in this scenario because there is no dependency of the state value on the other elements. That might not be true in other scenarios, so we have another solution too.

We use something called **memoization**. It is a technique where a function returns a cached version of the output for the same inputs. In other words, if you pass the same inputs to a memoized function, then it only calculates the output for the first time and then simply returns the cached value for subsequent execution. React provides us with an inbuilt hook called **useMemo()** for this purpose. Let us talk about that.

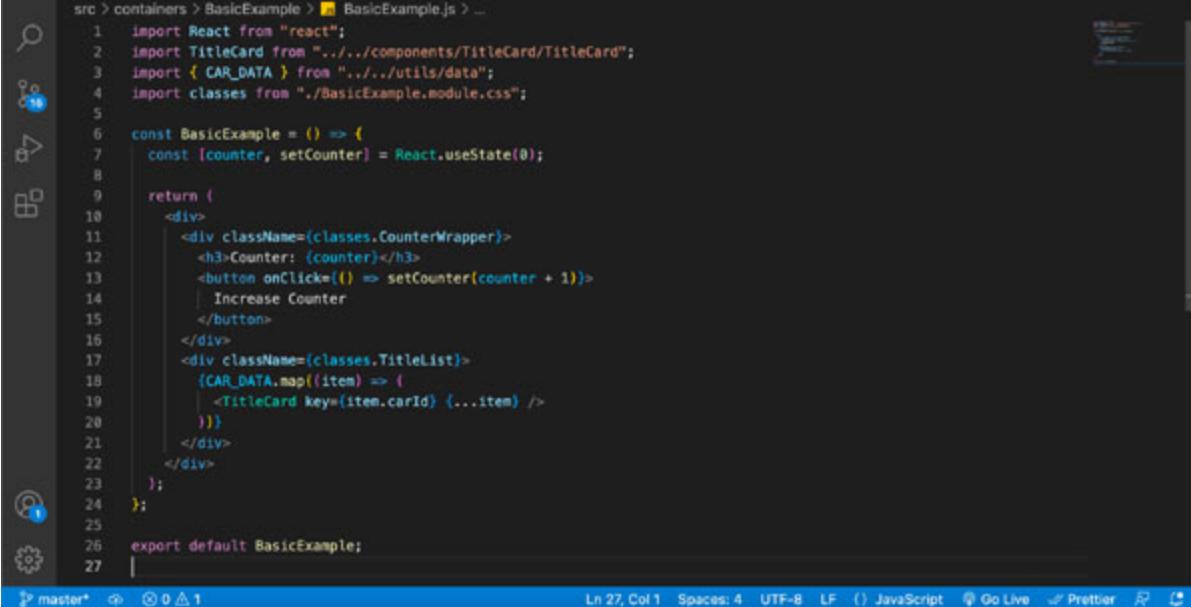
useMemo()

The **useMemo()** hook is used to return a memoized value. The following is the syntax:

```
const memoizedValue = React.useMemo(() =>
  computeExpensiveValue(a, b), [a, b]);
```

The **useMemo()** method accepts a callback function and a dependency array. The callback function will only recompute the memoized value when any of the dependencies have changed.

In the screenshot provided in [figure 11.8](#) is the implementation of the **BasicExample** without memoization:



```
src > containers > BasicExample > BasicExample.js > ...
1 import React from "react";
2 import TitleCard from "../../components/TitleCard/TitleCard";
3 import { CAR_DATA } from "../../utils/data";
4 import classes from "./BasicExample.module.css";
5
6 const BasicExample = () => {
7   const [counter, setCounter] = React.useState(0);
8
9   return (
10     <div>
11       <div className={classes.CounterWrapper}>
12         <h3>Counter: {counter}</h3>
13         <button onClick={() => setCounter(counter + 1)}>
14           Increase Counter
15         </button>
16       </div>
17       <div className={classes.TitleList}>
18         {CAR_DATA.map((item) => (
19           <TitleCard key={item.carId} {...item} />
20         )));
21       </div>
22     </div>
23   );
24 }
25
26 export default BasicExample;
```

Figure 11.8: BasicExample component implementation without memoization

You can find the implementation of the **BasicExample** component with the usage of the **useMemo()** hook in the screenshot as follows:

```
src > containers > BasicExample > BasicExample.js > ...
1 import React from "react";
2 import TitleCard from "../../components/TitleCard/TitleCard";
3 import { CAR_DATA } from "../../utils/data";
4 import classes from "./BasicExample.module.css";
5
6 const BasicExample = () => {
7   const [counter, setCounter] = React.useState(0);
8
9   const titleCardList = React.useMemo(
10     () => CAR_DATA.map(item) => <TitleCard key={item.carId} {...item} />, []
11   );
12
13   return (
14     <div>
15       <div className={classes.CounterWrapper}>
16         <h3>Counter: {counter}</h3>
17         <button onClick={() => setCounter(counter + 1)}>
18           Increase Counter
19         </button>
20       </div>
21       <div className={classes.TitleList}>{titleCardList}</div>
22     </div>
23   );
24 };
25
26 export default BasicExample;
27
```

Ln 27, Col 1 Spaces: 4 UTF-8 LF ⓘ JavaScript ⚡ Go Live ✅ Prettier ⌂ ⌂

Figure 11.9: BasicExample component implementation with memoization

Look at Lines 9–11. We have moved the code that was computing the entire **TitleCard** list inside the memoized function. We have kept the dependency array empty because we do not need to compute this list again because, in this scenario, the **CAR_DATA** list is never getting updated.

Let us check back the profiler results. As you can see, the **BasicExample** is getting rendered six times—one render for application load and then five renders for the counter increment, as shown in the following screenshot:

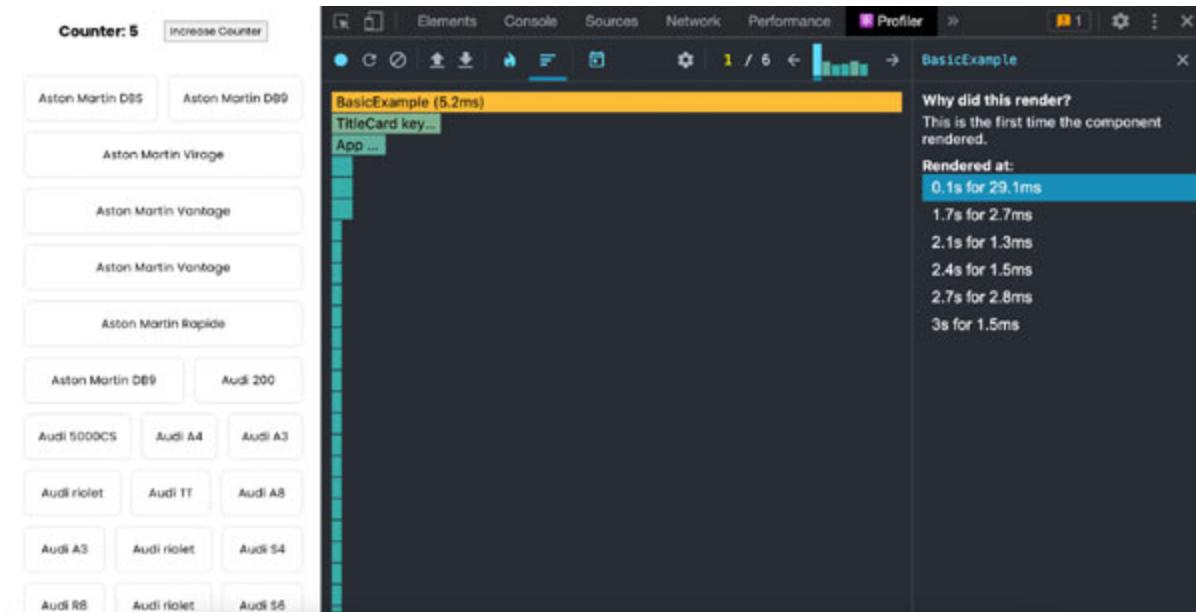


Figure 11.10: BasicExample component profiling with memoization

But now, if we check the profiling results for the `TitleCard` component, notice it is only rendered once when the application was loaded, as shown in the screenshot in [figure 11.11](#):

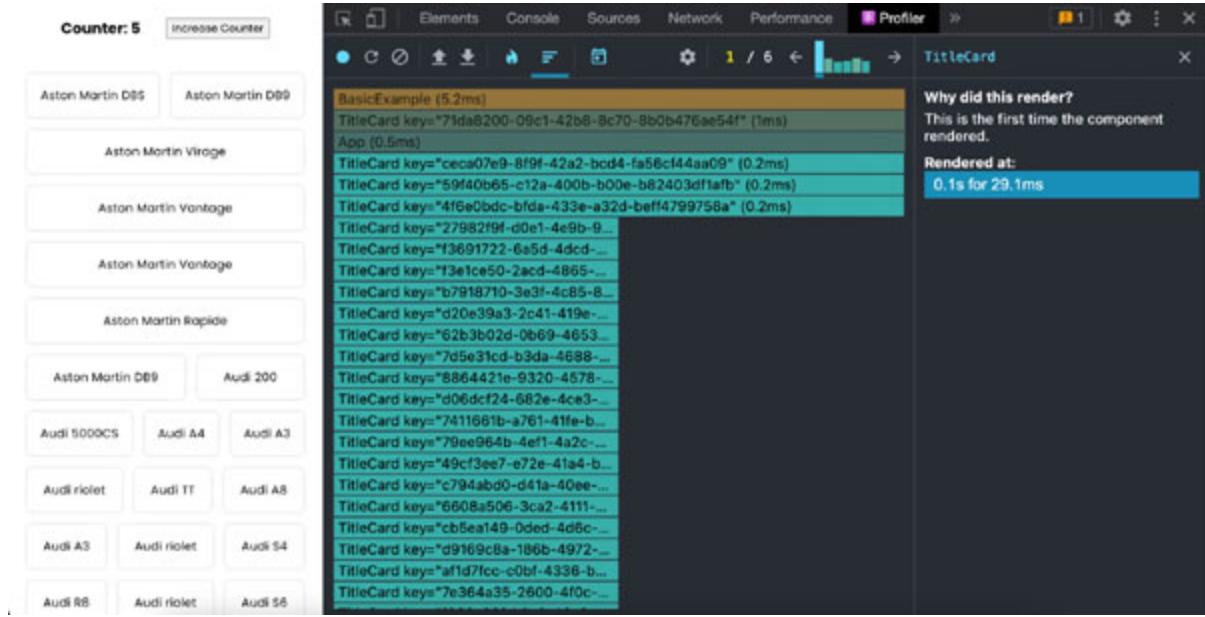


Figure 11.11: BasicExample component profiling with memoization

useCallback()

Let us look at another very important issue in our code. We have created an inline callback function in Line 17, which is updating the counter as shown in the following screenshot:

The screenshot shows a code editor with the following code:

```
src > containers > BasicExample > BasicExample.js > ...
1 import React from "react";
2 import TitleCard from "../../components/TitleCard/TitleCard";
3 import { CAR_DATA } from "../../utils/data";
4 import classes from "./BasicExample.module.css";
5
6 const BasicExample = () => {
7   const [counter, setCounter] = React.useState(0);
8
9   const titleCardList = React.useMemo(
10     () => CAR_DATA.map(item) => <TitleCard key={item.carId} {...item} />, []
11   );
12
13   return (
14     <div>
15       <div className={classes.CounterWrapper}>
16         <h3>Counter: {counter}</h3>
17         <button onClick={() => setCounter(counter + 1)}>
18           Increase Counter
19         </button>
20       </div>
21       <div className={classes.TitleList}>{titleCardList}</div>
22     </div>
23   );
24 };
25
26 export default BasicExample;
```

The code implements a basic React component named `BasicExample`. It uses `useState` to manage a counter state and `useMemo` to memoize the list of title cards. The component contains a counter display and an increase counter button.

Figure 11.12: BasicExample component implementation

The issue here is that every single time this component is rendered, it creates a new copy of the inline callback function. This is bad for the performance, and we are unnecessarily creating a new copy of a function. How do we fix this? Even if we moved it outside the return statement, it would still create new copies on rendering, as shown in the screenshot in [figure 11.13](#):

The screenshot shows a code editor with the following code:

```
src > containers > BasicExample > BasicExample.js > ...
1 import React from "react";
2 import TitleCard from "../../components/TitleCard/TitleCard";
3 import { CAR_DATA } from "../../utils/data";
4 import classes from "./BasicExample.module.css";
5
6 const BasicExample = () => {
7   const [counter, setCounter] = React.useState(0);
8
9   const titleCardList = React.useMemo(
10     () => CAR_DATA.map(item) => <TitleCard key={item.carId} {...item} />, []
11   );
12
13   const updateCounter = () => setCounter(counter + 1);
14
15   return (
16     <div>
17       <div className={classes.CounterWrapper}>
18         <h3>Counter: {counter}</h3>
19         <button onClick={updateCounter}>Increase Counter</button>
20       </div>
21       <div className={classes.TitleList}>{titleCardList}</div>
22     </div>
23   );
24 };
25
26 export default BasicExample;
```

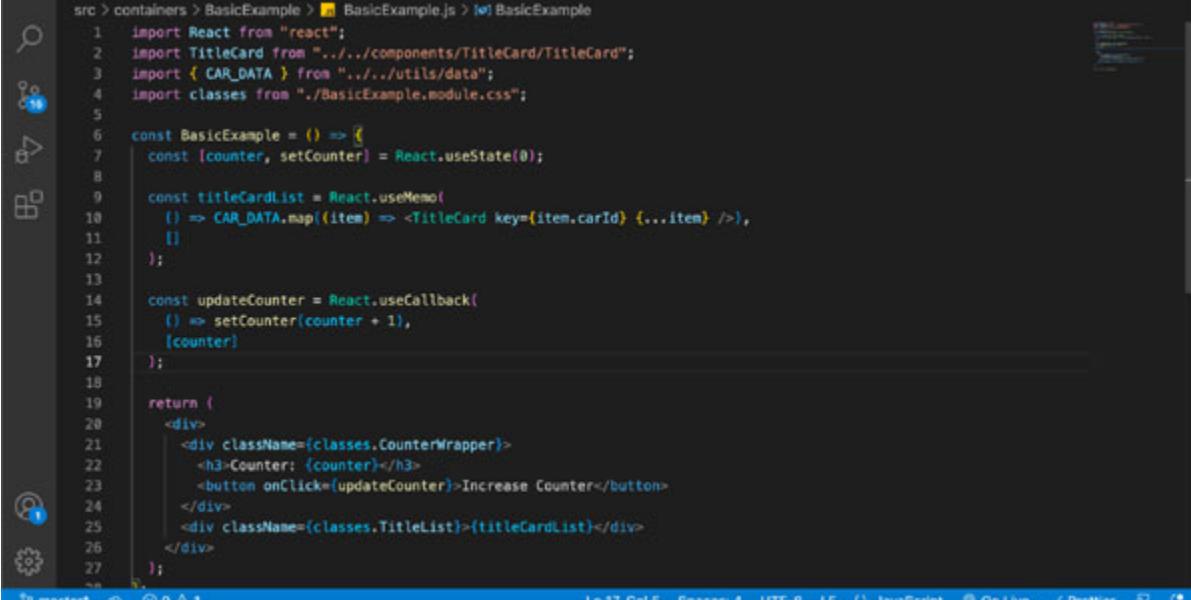
The code is identical to figure 11.12, except for the addition of a `const updateCounter = () => setCounter(counter + 1);` declaration. This variable is used as the `onClick` handler for the increase counter button, instead of an inline anonymous function.

Figure 11.13: BasicExample component implementation before `useCallback()`

To solve this problem, we use another hook provided by React, which is called `useCallback()`. This hook creates a memoized callback. The following is the syntax:

```
const memoizedCallback =  
  React.useCallback(originalCallbackFunction, [a, b]);
```

It accepts the original callback function definition and a dependency array which, when updated, will result in updating the memoized callback as shown in the screenshot that follows:



```
src > containers > BasicExample > BasicExample.js > BasicExample  
1 import React from "react";  
2 import TitleCard from "../../components/TitleCard/TitleCard";  
3 import { CAR_DATA } from "../../utils/data";  
4 import classes from "./BasicExample.module.css";  
5  
6 const BasicExample = () => {  
7   const [counter, setCounter] = React.useState(0);  
8  
9   const titleCardList = React.useMemo(  
10     () => CAR_DATA.map((item) => <TitleCard key={item.carId} {...item} />),  
11     []  
12   );  
13  
14   const updateCounter = React.useCallback(  
15     () => setCounter(counter + 1),  
16     [counter]  
17   );  
18  
19   return (  
20     <div>  
21       <div className={classes.CounterWrapper}>  
22         <h3>Counter: {counter}</h3>  
23         <button onClick={updateCounter}>Increase Counter</button>  
24       </div>  
25       <div className={classes.TitleList}>{titleCardList}</div>  
26     </div>  
27   );  
};
```

Figure 11.14: BasicExample component implementation after `useCallback()`

Look at the code from Lines 14–17. We have added a counter-state variable as a dependency because we want to be able to get an updated callback only when the counter value is updated.

If we pass an empty dependency array, then the counter value will be updated only for the first time and not for any subsequent attempts. Give it a try when you practice.

PureComponents

So far, we have talked about functional components. What about class-based components? How do we avoid unnecessary re-renders in class-based components? We can use something called **PureComponents**.

PureComponents improve performance by avoiding unnecessary re-rendering of components. If the previous value of state or props and the new value of state or props are the same, it does not re-render the component. It does a shallow comparison to check if the new and old values are the same for props and state variables.

Let us try an example where even when the value is not changing, it re-renders the component, as shown in the following screenshot:

```
qafikhani [new]
Drafts / sharp-currying-scs767 •
JS App.js
import React from "react";
class App extends React.Component {
  state = {
    name: "ReactJS"
  };
  updateGreeting = () => {
    console.log("Rendering App Component");
    this.setState({ name: "ReactJS" });
  };
  render() {
    return (
      <div>
        <h1>Hello {this.state.name}</h1>
        <button onClick={this.updateGreeting}>Update</button>
      </div>
    );
  }
}
cdc1416bb6
```

Figure 11.15: Unnecessary re-rendering without value change

Pure components are class-based components. We can create pure components by using the **PureComponent** class provided by React, as shown in the following screenshot:

```
js App.js
1 import React, { PureComponent } from "react";
2
3 class App extends PureComponent {
4   state = {
5     name: "ReactJS"
6   };
7
8   updateGreeting = () => {
9     this.setState({ name: "ReactJS" });
10 }
11
12 componentDidUpdate() {
13   console.log("Rendering App Component");
14 }
15
16 render() {
17   return (
18     <div>
19       <h1>Hello ${this.state.name}</h1>
20       <button onClick={this.updateGreeting}>Update
21     </div>
22   );
23 }

```

Figure 11.16: Avoiding unnecessary re-rendering using pure components

Try a working example at the following URL:

[https://codesandbox.io/s/before-pure-component-scs767?
file=/src/App.js:0-470](https://codesandbox.io/s/before-pure-component-scs767?file=/src/App.js:0-470)

Because Pure Component does only a shallow comparison, it cannot be used to avoid rendering issues when the state and props values have more complex values like nested objects and arrays. To overcome this shortcoming, React provided us with another method called **shouldComponentUpdate()** which can stop the updation lifecycle based on a false return value.

shouldComponentUpdate()

If you remember the lifecycle methods, there was one called **shouldComponentUpdate()**. We can use it to avoid unnecessary re-renders. We know that rendering is a heavy process, especially when the layout is very complex.

Every time the **setState()** function is called. It starts the entire cycle of component updates. A new virtual DOM is generated, compared with the old version of the virtual DOM, and then it updates the actual DOM. This can hurt the performance of a React App drastically if done when not

required. So, the rule is simple, avoid unnecessary re-renders as much as possible.

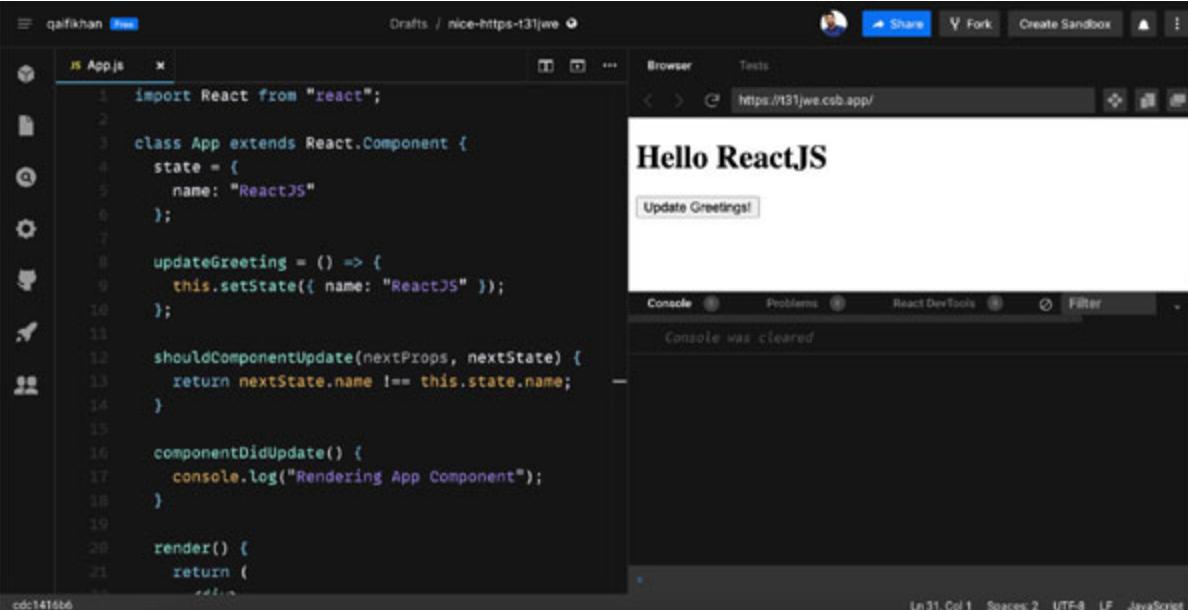
And how do we do that?

We can use the **shouldComponentUpdate()** lifecycle method. It gives us access to the new props and new state for which the component is getting updated. We can write custom logic to check if a re-render is required or not. If a re-render is required, then we can re-render, or else we can avoid the re-render. The **shouldComponentUpdate()** method needs to return a boolean value. If the returned value is true, then the update lifecycle continues, and the component is re-rendered; else, the re-rendering is stopped.

The following is the syntax:

```
shouldComponentUpdate(nextProps, nextState) {  
  return true; //or false based on some conditions  
}
```

So, let us try the same example we had in Pure Component, as shown in the [figure 11.17](#):



The screenshot shows a code editor interface with a file named `App.js`. The code is as follows:

```
import React from "react";  
  
class App extends React.Component {  
  state = {  
    name: "ReactJS"  
  };  
  
  updateGreeting = () => {  
    this.setState({ name: "ReactJS" });  
  };  
  
  shouldComponentUpdate(nextProps, nextState) {  
    return nextState.name !== this.state.name;  
  }  
  
  componentDidUpdate() {  
    console.log("Rendering App Component");  
  }  
  
  render() {  
    return (  
      <div>
```

To the right of the code editor is a browser window displaying the application. The title bar says "Hello ReactJS". Below the title, there is a button labeled "Update Greetings!". At the bottom of the browser window, the developer tools are visible, showing the "Console" tab with the message "Console was cleared".

Figure 11.17: Avoiding unnecessary re-rendering using shouldComponentUpdate()

Look at Lines 12–14; we have used **shouldComponentUpdate()** and added a condition that it should return true only when the value of the name is different in the current and next state.

Try a working example at the following URL:

<https://codesandbox.io/s/shouldcomponentupdate-t31jwe>

If we return false, then even if the values are different, the component will not be updated. Give it a try when you practice.

Code splitting

If you remember the base concept of Single Page Applications, there is an HTML file, bundled CSS file, and a bundled JS file. Imagine you write a large application, maybe something like Amazon; it would have a huge file size for the JS bundle. Even if the bundle size is 10 MB, it is huge for your users, especially on mobile devices. Try to download a 10 MB file over a 4G network, and you will know that it takes more than 30 seconds to download. If it takes 30 seconds for your Web app to load, then you will lose the majority of your users.

What is the solution for this huge JS bundle? Simple, we split the bundle into smaller chunks. If you are using create-react-app to build your React application, then it automatically configures the bundling using Web pack.

We can take this even a step further. Let us take Amazon, for example, when the user lands on the homepage, they would not need the code for the Product Details Page, User Account Page, Order History page, and so on. Right? We can do code splitting at the route level. This is called lazy loading. You do not load everything on the application load, and you load the code as required. We use `React.lazy()` to load components dynamically.

This is how we would configure our routes, as shown in the following code snippet:

```
import Home from './containers/Home';
import ProductDetails from './containers/ProductDetails';
import OrderHistory from './containers/OrderHistory';
const App = () => (
  <Router>
    <Routes>
      <Route path="/" element={<Home />} />
      <Route path="/about" element={<ProductDetails />} />
      <Route path="/about" element={<OrderHistory />} />
```

```
        </Routes>
    </Router>
);
```

To introduce lazy loading, we would write the code as shown in the following code snippet:

```
const Home = lazy(() => import('./containers/Home'));
const ProductDetails = lazy(() =>
import('./containers/ProductDetails'));
const OrderHistory = lazy(() =>
import('./containers/OrderHistory'));
const App = () => (
    <Router>
        <Routes>
            <Route path="/" element={<Home />} />
            <Route path="/about" element={<ProductDetails />} />
            <Route path="/about" element={<OrderHistory />} />
        </Routes>
    </Router>
);
```

What happens if the **ProductDetails** component is huge and takes a few seconds to load? The users will see a blank white screen. We can replace this with a custom fallback UI, maybe a progress loader. For this, we use a component called **Suspense**, as shown in the following code:

```
const Home = lazy(() => import('./containers/Home'));
const ProductDetails = lazy(() =>
import('./containers/ProductDetails'));
const OrderHistory = lazy(() =>
import('./containers/OrderHistory'));
const App = () => (
    <Router>
        <Suspense fallback={<div>Loading...</div>}>
            <Routes>
                <Route path="/" element={<Home />} />
                <Route path="/about" element={<ProductDetails />} />
                <Route path="/about" element={<OrderHistory />} />
            </Routes>
        </Suspense>
    </Router>
);
```

```
</Router>
);
```

Conclusion

Normally React applications are very fast. We introduce performance issues by writing bad code, unoptimized animations, slow APIs, not splitting the build files, and many more reasons. So, normally you would not have to worry about performance issues, but if you notice that your app's performance is not good, then you can use auditing tools to benchmark the performance and make the necessary changes to get the best performance.

In the upcoming chapter, we will start with the fundamentals of UI/UX. We will learn to use Figma to create wireframes and design simple Web apps.

Questions

1. What is React Profiler? How to use it to analyze performance issues?
2. What is useMemo()? How do you use it?
3. What is useCallback()? How do you use it?
4. What is shouldComponentUpdate()? How do you use it?
5. What is lazy loading? How do you use it?

Multiple choice questions with answers

1. We can use the useMemo() hook in both functional and class-based components?
 1. True
 2. False
2. **We can use the useCallback() hook in both functional and class-based components?**
 - a. True
 - b. False
3. **What happens if you keep the dependency array empty in useCallback() hook?**

- a. It throws an error
 - b. It does not call the function
 - c. It does not affect the code execution
 - d. It will execute the callback only once
4. What happens when you return false from `shouldComponentUpdate()`?
- a. It throws an error
 - b. It allows updating the component
 - c. It does not allow updating the component
 - d. It updates the component only once.
5. A component can only be updated once?
- a. True
 - b. False

Answers

Question	Correct Answer
1	b. false
2	b. false
3	d. It will execute the callback only once
4	c. It does not allow updating the component
5	b. False

CHAPTER 12

Starting with Tools and Concepts of UI/UX

In this chapter, we will introduce UI/UX. We will talk about different areas in UI/UX design. We will build a strong foundation here by understanding the principles of UI/UX design. We will learn how to use Figma to create wireframes, designs, and prototypes.

Structure

In this chapter, we will discuss the following topics:

- Introduction to UI/UX
- Difference between UI and UX
- UI/UX Tools
- Fundamental concepts
- Different stages
- Typography
- Color schemes
- Font hierarchy

Objectives

We will explore different tools used in the industry for UI/UX design. We will see what each of these tools is used for. We will discuss different options for icons and free versus paid services to get icon packs. After studying this chapter, you will be able to use Figma to design wireframes, designs, and prototypes.

Introduction to UI/UX

UI stands for “User Interface”, and UX stands for “User Experience”. UI focuses on how the design looks; what is the color theme? What is the layout? Whether to show a list of cards or a table? On the other hand, UX defines the flow of the application. What information the user will see on a page? What will be the next page? How the user will be navigated to the next page? And so on.

There are specific roles for UI designer, UX designer, and even UI/UX designer. Depending on the role, the person handles different responsibilities. Both UI/UX go hand-in-hand. It is not good to have a very beautiful design but terrible navigation and flow. It is also not good to have a very smooth flow but a terrible design. Both are undesirable. If the application looks bad, then UI needs to be fixed, but if the application is confusing and difficult to navigate, then the UX needs to be fixed.

In this book, we will focus on the UX while wireframing and UI while designing. Do not worry; we will learn about these concepts soon.

UI/UX tools

For UI/UX, we have several tool options to choose from, such as Figma, Sketch, and AdobeXD. All the top tools are pretty much similar to each other in terms of features and workflows, so really, it is just a matter of preference and pricing options.

Here is a quick comparison of all three tools in [*table 12.1*](#):

Criteria	Figma	Sketch	AdobeXD
Platform	Desktop and browser application	Desktop and browser application	Desktop and mobile application
Operating system	Windows, macOS, and Linux	macOS	Windows, macOS, iOS, and Android
Price	Free starter version or \$12 per editor/month	30-day free trial, then \$9/month per editor (USA)	7-day free trial, then \$9.99/month (USA)

Table 12.1: UI/UX tools comparison

The most commonly used tool is Figma, and we are going to use Figma in this book to learn wireframing, designing, and prototyping.

Different stages involved in UI/UX projects

The following are the main stages involved in a UI/UX project or a product design project:

- User research
- Information architecture
- Wireframe
- Design
- Prototype

User research

In this stage, the product designer or the UI/UX person will research the users who will be using the product. A “User Persona” is designed based on the ideal user of the product. This is a very important stage because the more you understand about the users who will be using the product, the more useful product can be designed by you.

Here are three questions to ask when designing user personas:

1. **Who is the user?** This is where you gather user info such as age, gender, demography, occupation, and so on.
2. **What is the main goal or problem which has to be solved?** This is where you define why the users will use a certain product.
3. **What is the main barrier to achieving this goal?** This is where you list down all the challenges faced by the users.

For example, say you are building the designs for an online learning platform where the main target audience is college students. In this scenario, a user persona might look like as shown in the following:

Figure 12.1: Sample user persona for an online learning platform

After creating this User Persona, you would know what information should be highlighted more, which journeys are the pain-point for the users, what will be the key features of the product, and so on.

Information architecture

This is the stage where you collect all the information that needs to be shown throughout different journeys. The objective is to help users find information seamlessly. To do this, you need to understand how the pieces fit together to create the larger picture and how items relate to each other within the system. This sets the context for the wireframe stage.

For example, the homepage of the learning platform will have the following information shown to the users:

Figure 12.2: Sample information architecture

Wireframe

It is the UI/UX stage where we just sketch up the screens of the application we are designing. It can be done on a pen-paper, as shown in the following image:

Figure 12.3: Pen-paper wireframe

It can also be done using tools like Figma, as shown in the following screenshot:

Figure 12.4: Figma wireframe

During the wireframing stage, the focus is mainly on space allocation, deciding what content needs to be shown, available functionalities, and feature journeys/flows.

Design

This is the stage where the wireframes are converted to actual designs. This is the stage where you see what the final product will look like. It does not have to be an exact replica of the wireframes with just colors added. The layout can differ; you might see cards in wireframes but might end up having a table in the final design. Remember, the focus of wireframes is mainly on space allocation, deciding what content needs to be shown, available functionalities, feature journeys/flows, and so on, and not how it should appear.

For example, here is the design for the wireframe we created in the previous example:

Figure 12.5: Sample design

Prototype

This is the stage where you make your designs interactive to show how the user journey will flow or how a certain feature will function. For example, let us say you have created a video upload journey containing five different screens. It would always be better for your clients or end users to be able to see the first screen and interact with the clickable items to explore the remaining screens as a flow rather than seeing all five screens as separate designs and trying to figure out how it all works together.

A lot of time, start-ups build a small product as a prototype to show potential investors or users to see how they receive the product or a new feature. This helps in validating ideas without actually spending time and effort of the engineering team.

So, let us see how we create prototypes in Figma. We are going to try a simple click effect, where when you click on the course card, it will take you to the course details page. Please follow the following steps:

1. Click on the **Prototype** option in the right-side panel, as shown in the following screenshot:

Figure 12.6: Switch to the prototype

2. Select a frame to start prototyping, as shown in the following screenshot:

Figure 12.7: Select a frame to start prototyping

3. It is recommended to set the **Flow Starting Point**, which can be done only for a frame and not an element. When a user loads the prototype, this frame will be loaded as the starting point. Please click on the “+” button as shown in the following screenshot:

Figure 12.8: Setup flow starting point

4. Select the course card and then click on the “+” button next to **Interactions** to set up the click listener, as shown in the following screenshot:

Figure 12.9: Add an interaction

5. To set up an interaction, we need to provide a few values:

- a. An action that will trigger the interaction.
- b. The result of the interaction. For example, navigate to a certain frame, scroll to a certain element, and so on.

In our case, we will set the action as click, as shown in the following screenshot:

Figure 12.10: Setup interaction on the first-course card

6. Select what needs to be done when the course card is clicked, as shown in the following screenshot:

Figure 12.11: Setup result on click

7. Select where the user will be navigated, as shown in the following screenshot:

Figure 12.12: Select the frame for navigation

8. For the navigation option, we get an additional configuration to control the transition between two frames. The transition values are very similar to what we have in CSS transitions. You can simply select **Smart Animate**, and Figma will create a smooth transition for you, as shown in the following screenshot:

Figure 12.13: Select transition style

9. That is it, and our first prototype is ready. Now, we can view the design as a prototype by clicking on the prototype play icon on the topbar, as shown in the following screenshot:

Figure 12.14: Click the play button to preview the prototype

This was one example of a prototype where we went from one frame to another. You can also add interactions where on click, you change variations of components. Let us try a simple example, say, when you click on stars in the details page, it should update the rating. When you click on the fifth star, then the rating should change to 5 stars. When you click on the fourth star, then the rating should change to 4 stars, and so on. The functionality might not make sense, but this is just for your reference. Before you can prototype this functionality. Please create a frame for the rating stars. Right click on the frame and click on "Create Component". Once the component is created, right click again to create variations. Add variations for all 5 ratings. Please follow the following steps:

1. Switch to the *prototype* mode. Select the fifth star and then click on the “+” button next to **Interactions**, as shown in the following screenshot:

Figure 12.15: Select a star and add an interaction

2. Select on click and change to option in the configuration dropdowns, as shown in the following screenshot:

Figure 12.16: Select on click and change to options

3. Select a component variation to change to as shown in the following screenshot:

Figure 12.17: Select a variation from the dropdown

4. Click on the play prototype button to preview your updated prototype.

That is how you can show/hide components or switch between different variations of a component.

You can refer to this example in the following Figma file:

<https://www.figma.com/file/KZWWKCKUvy5M9W0knLuaKY/RFJS--Course-Grid-Prototype?node-id=1%3A2&t=PavTxYDgXk3cJOcV-1>

We hope this gave you some sense of what are the different stages involved while working on a design project.

Fundamental concepts

In this section, we will talk about the core design concepts in UI/UX:

- White space
- Contrast
- Scale
- Alignment
- Colors
- Typography
- Visual hierarchy

White space

“White space basically refers to the empty areas in a user interface”. It does not have to be white, it could be of any color, and it just refers to the empty space. It is also known as negative space.

Let us try an example with white background. Following is a screenshot of a landing section:

Figure 12.18: Landing page with white background

The highlighted red part is the white space, as shown in the following screenshot:

Figure 12.19: A landing page with white background—highlighted white space

Let us try an example with a dark background. Following is a screenshot of another landing section:

Figure 12.20: A landing page with dark background

The highlighted red part is the white space, as shown in the following screenshot:

Figure 12.21: A landing page with a dark background—highlighted white space

White space should be applied based on the following four factors:

- Space availability
- Number of components
- Alignment
- Consistency

Figure 12.22: Improper whitespace

Notice the following in the previous screenshot:

1. **Space availability and number of components:** Look at the topbar, we have a total of five elements, but all the elements are spaced on the left and right. We can fix this by adding white space between the menu items too.

2. **Alignment:** Look at the orange and blue white spaces. The heading “Top 5 muscle...” and body “The standard Lorem...” is not aligned properly. We can align them by reducing the orange-white space from the body element. Similarly, we can align the heading and reading duration in the center vertically by reducing the blue-white space above the duration text.

3. **Consistency:** Look at the green-white spaces; there is inconsistency. The space is less on the left and more on the right-hand side. We can fix this by giving equal space on both sides.

After making the preceding changes mentioned, our design will look as shown in the following:

Figure 12.23: Whitespace issues fixed

Contrast

Contrast helps separate elements from each other by adjusting the color, brightness, and/or opacity.

Compare the following two screenshots:

Figure 12.24: Whitespace issues fixed

As you can see in the preceding screenshot.

1. The menu items are not visible due to not enough contrast between the text and background color.
2. The heading text is not visible because the text color does not have enough opacity. It has very high transparency, which makes it difficult to read the heading.
3. The description text is not visible because the text color is not bright enough. It is a very light shade of gray.

Let us fix all these issues and see the fixed design in the following screenshot:

Figure 12.25: Whitespace issues fixed

Scale

When talking about scale, we need to think of two things—everything should be readable, and there should be enough white space. What this means is that if we are talking about text elements, then it needs to be readable, and if we are talking about other components, then they should be scaled up or down to have the right amount of white space.

Let us check out an example to understand text scaling issues. Have a look at the following screenshot:

Figure 12.26: Too small text size

As you can see in the preceding screenshot, the text is not readable. Similarly, even if the text is too large even, that is a problem, as shown in the following screenshot:

Figure 12.27: Too large text size

That is why we need to scale text such that it is legible but does not affect the visual hierarchy of the page or design, as shown in the following screenshot:

Figure 12.28: Text scaling issue fix

Similarly, we can face scaling issues with non-text components, as shown in the following screenshot:

Figure 12.29: Non-text scaling issue fix

As you can see in the preceding screenshot, there seems to be no white space. All the components are scaled to take up all the available space, and this is also a problem. So, we need to scale our components such that they leave enough white space for a cleaner design, as shown in [*figure 12.28*](#).

Alignment

It is an invisible component in our designs. It provides a structure and flow to our layout.

Checkout the screenshot that follows and lists down the alignment issues:

Figure 12.30: Alignment issues

We have added dotted lines to highlight the alignment issues, as shown in the following screenshot:

Figure 12.31: Alignment issues—highlighted

Here are the alignment issues as highlighted in the previous screenshot:

1. The card title, topic, and part are not aligned with each other.
2. The start button is not aligned with the info card.
3. The **Join Now for FREE** click-to-action is not aligned with the cards.

We have fixed the alignment issues in the following screenshot:

Figure 12.32: Alignment issues—fixed

Colors

Colors are the most important part of our user interface. Every website has a few sets of colors that are used throughout to kind of make a theme. This theme is called the color scheme of the website.

In the following screenshot, we have a color picker from Figma:

Figure 12.33: Figma color picker

The color box shows different shades of the blue we have selected. We can make it light or dark by moving the selector inside the box. In addition, we can add/reduce the transparency from the color.

Let us talk about some commonly used terms for colors:

1. **Hue:** Full-color saturation
2. **Tint:** Add light to the hue
3. **Shade:** Add darkness to the hue
4. **Tone:** Add gray to the hue

We have visualized Hue, Tint, Shade, and Tone in the following screenshot:

Figure 12.34: Color components—Hue, Tint, Shade, and Tone

We have the following three types of color palettes:

1. Monochromatic

Figure 12.35: Monochromatic palette

2. Analogous

Figure 12.36: Analogous palette

3. Complementary

Figure 12.37: Complementary palette

Typography

There are four main concepts that we will learn in typography. If you know CSS, then you will already know about these concepts:

- **Tracking:** Space between letters. Also known as letter spacing.
 1. Most of the time, we would stick with the normal values.
 2. Wider tracking is mostly used for headings and labels.
- **Leading:** Space between lines of text. Also known as line height.
- **Font style:** Serif, sans-serif, and script.

- **Font weight:** It defines how thick or thin is the font.

Visual hierarchy

Visual hierarchy refers to giving priority to different elements based on their importance. This helps to focus the user's attention on more important things automatically. We will use all the previous concepts, such as white space, contrast, scale, alignment, color, and typography, to design a good visual hierarchy between elements.

Look at the following screenshot, it has a few issues:

1. There are multiple texts which look the same, but we do not know which one is the heading and which one is the subtext.
2. All the texts and buttons lack enough contrast with the background, which makes it difficult to read.
3. There is a lot of white space.

Figure 12.38: Card with no visual hierarchy

After adding the fixes, the card looks better as shown in the following screenshot:

Figure 12.39: Card with proper visual hierarchy

Conclusion

We hope this gave you some understanding of the different stages of UI/UX design. Now you should be comfortable with the fundamental concepts of UI/UX design. You should also know how to sketch wireframes, create Web designs, and functional prototypes.

In the upcoming chapter, we will go over some trending design patterns used in UI/UX design like Neomorphism, Glassmorphism, Pastel Colors, Soft Gradients etc.

Questions

- Q1. What are the different stages of UI/UX design?
- Q2. What is the difference between wireframes and design?
- Q3. What is white space?
- Q4. What are color schemes? Explain a few types of color schemes.
- Q5. What is Visual Hierarchy?

CHAPTER 13

Trending UI Patterns

In this chapter, we will explore some of the trending design patterns, such as glassmorphism and neomorphism, to add more depth to our page designs and pastel backgrounds for a very clean and minimalistic look. We will also learn about the dark and light theming of a Webpage.

Structure

In this chapter, we will discuss the following topics:

- Glassmorphism
- Neomorphism
- Trying out soft gradients
- Working with geometric elements
- Pastel backgrounds
- Designing dark mode

Objectives

After studying this chapter, you will be able to use Figma to design websites with glassmorphism, neomorphism, pastel backgrounds, geometric elements, and dark mode.

Glassmorphism

This is another popular trend that emerged in 2021. As the name suggests, this gives us a frosty-glass kind of effect on our Web designs, as shown in the following screenshot:

Figure 13.1: Sample glassmorphism Webpage

As you can see in the preceding screenshot, both the credit card and the form card look like glass. They are transparent, the background is kind of blurry, and the edges seem elevated, which gives it a 3D look. These exactly are the three things we use to design glassmorphic cards:

1. **Transparent background:** This will make our elements see-through.
2. **Background blur:** This will blur the background elements to give it a frosty glass-like effect.
3. **Shadows:** We add inner and box shadows to add depth around the edges which will give it a 3D effect.

Let us try to implement the previous glassmorphism design.

As you can see in the following screenshot, we have all the required components, just that the cards are black. Let us do our magic and add glass-effect:

Figure 13.2: Glassmorphism starter design

The first step will be to make it transparent. We are going to change the background color to white and try adding the transparency value of maybe 5%–10% and see what looks good. Remember, this is the designing stuff; a specific value does not always work; you have to try different values to see what looks best in your design.

Figure 13.3: Step-1 add transparency

Now, we got to add blur to the background. Try values between 5 and 10 and check what looks the best.

Please follow these steps to add the background blur:

1. Click on “+” next to the effects on the bottom right of the design pane.

Figure 13.4: Add effect

2. Select “**Background blur**” from the dropdown.

Figure 13.5: Select background blur

3. Set the value for blur. Higher values will make it more blurry, whereas smaller values will make it less blurry.

Figure 13.6: Set the blur value

This will add the frosty-glass effect, as shown in the following screenshot:

Figure 13.7: Step-2 add background blur

As you can see in the preceding screenshot, the cards are blending into the background image, and they do not look elevated or 3D. To achieve this, we will add two shadows to each card.

Please follow the following steps to add shadows:

1. Click on “+” next to the effects on the bottom right of the design pane.
2. Select “**Inner Shadow**” or “**Box Shadow**” from the dropdown, depending on what you are trying to do.

Figure 13.8: Select the inner shadow

3. Set the values to style the shadow.

Figure 13.9: Select values for the inner shadow

The shadow values are the same as those we have in CSS shadows. X and Y control the horizontal and vertical positioning of the shadow. The blur value controls the intensity of the blur; the higher the value more blurred the shadows are going to be, and vice versa.

As mentioned earlier, we will add two shadows to each card. First will be an inner shadow that will add an effect on the top and left edge, which will make it appear as if there is a light source on the top left side, as shown in the following screenshot:

Figure 13.10: Inner shadow

The second will be box shadow which will add a shadow effect on the bottom and right edge, as shown in the following screenshot:

Figure 13.11: Box shadow

Hope this gives you clarity on how to add glassmorphism to your designs.

Here is the starter Figma file for you to try as well:

<https://www.figma.com/file/zEbW95QIC6jEzhyr9lfWxA/RFJS-Glassmorphism-Starter-File?node-id=0%3A1>

Here is the final design after adding glassmorphism to the cards:

<https://www.figma.com/file/BZF9eKaTBD0oNMvEpAVVQ8/RFJS-Glassmorphism-Final?node-id=0%3A1>

Neomorphism

Neomorphism was one of the hottest design trends of 2020. It combines three elements—monochromatic color schemes, low contrast, and subtle shadows, which will create a very subtle but beautiful 3D effect. It can be used on both web and mobile design, as shown in the following screenshots:

Figure 13.12: Sample neomorphism Web-design

We can have both light and dark themes on neomorphism, as shown in the following screenshot:

Figure 13.13: Neomorphism (a) light theme and (b) dark theme

Let us design the order overview page. Here is the starter design URL:

<https://www.figma.com/file/5RuAjhXBXy1XYAuQhTlGc3/Neomorphism-Starter?node-id=0%3A1>

Here is what the starter design looks like:

Figure 13.14: Orders overview starter design

To add the neomorphism 3D effect to the cards, we need to add box shadows to them. Let us try adding to one of the cards, and then you can add the same shadows to the remaining ones. Please follow the following steps:

1. We need to add two box shadows.
2. The first box shadow will make it appear as if the light is coming from the top left side of the card.
3. Click on the effects and then select box shadow.
4. Add the following properties:

Figure 13.15: Add top-left shadow

5. Click again on the effects and select box shadow to add the second shadow. This will make it appear that the card has a height and the shadow is coming because of the light.
6. Add the following properties:

Figure 13.16: Add bottom-right shadow

7. After adding the two shadows, your first card should be as shown in the following screenshot:

Figure 13.17: Card with neomorphism

8. Similarly, add shadows to the remaining cards to achieve the desired effect.

I hope this gave you some clarity on how to add neomorphism to your designs.

Trying out soft gradients

The gradient is a smooth transition between two or more colors. There are four types of gradients available in Figma—linear, radial, angular, and diamond. Linear and Radial are the most commonly used, and we will learn these two in this chapter.

A linear gradient is when the transition between colors is linear. It is very common in button background colors. It could be in any direction, horizontal, vertical, or even at an angle, as shown in the following screenshots:

Figure 13.18: Linear gradient (a) diagonal, (b) vertical, and (c) horizontal

Please follow the following steps to add a linear gradient to your design:

1. Click on “+” next to fill on the bottom right of the design pane. A gradient is usually added as a background color, as shown in the screenshot given in [figure 13.19](#):

Figure 13.19: Add a fill

2. Click on the color box as shown in the screenshot given in [figure 13.20](#).

Figure 13.20: Click on the color box next to “FFFFFF” to show the color config popup

3. Click on the dropdown to select the type of gradient, as shown in the screenshot given in [figure 13.21](#).

Figure 13.21: Select the type of gradient from the dropdown

4. Set two or more colors and angles for the gradient, as shown in the screenshot given in [figure 13.22](#).

Figure 13.22: Setup gradient values

A radial gradient is where the transition between colors is elliptical or circular, as shown in the following screenshots:

Figure 13.23: Radial gradient (a) circular and (b) elliptical

There can be more than two colors in both linear and circular gradients, as shown in the following screenshots:

Figure 13.24: Multiple colors (a) linear and (b) radial gradient

Work with geometric elements

We can also construct designs using geometrical shapes, such as squares, rectangles, circles, cones, and cubes. Here is an example shown in the following screenshot:

Figure 13.25: A landing page with geometrical elements

There is no new concept in the preceding design. You can implement it using the concepts you have learned so far. Here is the starter code for the preceding landing section:

<https://www.figma.com/file/etyiQq9j1q3F5FDdSoMaUt/RJFS-Geometrical-Design-Starter?node-id=0%3A1>

Here is the completed version for your reference:

<https://www.figma.com/file/3npxP3SA0aJPbRdIF3l5tq/RJFS-Geometrical-Design-Final>

Pastel backgrounds

You might have seen some websites with minimalistic designs and pastel colors for the theme. These are also very popular for that clean and minimalistic look. Here are some samples of pastel color palettes, as shown in the following screenshot:

Figure 13.26: Sample pastel color palettes

Here is a screenshot, shown in [figure 13.27](#), of a landing page with a pastel background:

Figure 13.27: Sample pastel landing page

Here is the link to the final Figma design file:

<https://www.figma.com/file/1EbzCzlEpwCErfZrTmwsYP/pastel-landing-page?node-id=0%3A1>

Designing dark mode

Another common pattern you must have noticed, especially on blog websites, has the option to toggle between light and dark mode. As the name suggests, in the light mode, you will have a lighter shade of background, whereas, in the dark mode, the background is going to be some shade of a dark color.

Here is a blog with the light theme, as shown in the following screenshot:

Figure 13.28: Blog page—light theme

Here is a blog with a dark theme, as shown in the following screenshot:

Figure 13.29: Blog page—dark theme

Please follow the following steps to convert the light theme blog page to the dark theme:

1. Change the page background color to “#262525”.
2. Change the description card background color to “#3A3939”.
3. Change the text color to “#FFFFFF” for the blog body and topbar menu items.
4. Change the stroke color for the blog card to “#000000”.

Notice all we did was change the light background colors to dark colors and updated the text colors to white.

Here is the Figma file URL for your reference:

<https://www.figma.com/file/EgrT5rmvjnJ4m6x2rq4j88/Blog-Dark-Light?node-id=1%3A55>

Conclusion

You can use different design trends, such as glassmorphism, neomorphism, pastel palettes, and geometric designs, to add a fresh and clean look. It is good to provide an option for users to switch between the light and dark modes, especially on text-heavy websites.

Questions

- Q1. What is glassmorphism? Show an example.
- Q2. What is neomorphism? Show an example.
- Q3. What are gradients? Explain different types of gradients with visual examples.
- Q4. What is dark mode?
- Q5. What are geometric elements?

CHAPTER 14

Prepping for React Interviews

In this chapter, we will talk about the interview processes for the frontend developer role using ReactJS. We will also go through some commonly asked interview questions with more emphasis on the topics which are frequently asked in the interviews.

Structure

In this chapter, we will discuss the following topics:

- React Interview Process
- Resume Template
- Interview Preparation Material

Objectives

After studying this chapter, you should understand the interview process for the frontend developer role. You should also be able to create a resume suited for a frontend developer role.

React Interview Process

When we apply for a frontend role, the overall process usually remains the same, but the emphasis differs based on the role you are applying for. For example, if you are applying for a Junior Frontend Developer, there might be a take home assignment, live project coding round, data structures and algorithm round and non-tech rounds. If you are applying for a Senior Frontend Developer role then you might have 2 live project coding rounds, a DSA round and non-tech rounds. In the upcoming sections, we will go through some frequently asked questions on ReactJS.

Must have skills for a frontend developer - HTML, CSS, JavaScript, React & Redux

Additional skills for a frontend developer - TypeScript, Tailwind, and Redux-Query

Resume Template

Full Name

Contact Number

Role you are applying for

Contact Email

ABOUT ME [Summary]

I started building frontend applications when I was in my first year of college and I have enjoyed building new products ever since. Fast forward 4 years, I have perfected my craft by working on various frontend projects across different domains like e-commerce, social media, blogging applications, and chat applications. I have learned the required skills to be an effective frontend developer and I look forward to learning much more by being a part of the industry.

INTERNSHIP EXPERIENCE [Please note: Internships are very important, they prove that you have what it takes to be in the tech industry]

Company Name [For Internship 1]

What projects you worked on and what have you learned at the internship. If possible then provide links to the live web applications that you have worked on and explain your contribution.

Duration: 6 months

Company Name [For Internship 2]

What projects you worked on and what have you learned at the internship. If possible then provide links to the live web applications that you have worked on and explain your contribution.

Duration: 3 months

SELF PROJECTS

Project Title - E-commerce App [For Project-1]

Live Application: Link [Ensure that the link is short]

Technical Stack: HTML, CSS, JavaScript, React & Redux

Explain the project in 3-4 lines.

Project Title - E-commerce App [For Project-2]

Live Application: Link [Ensure that the link is short]

Technical Stack: HTML, CSS, JavaScript, jQuery, Bootstrap

Explain the project in 3-4 lines.

TECHNICAL SKILLS

Frontend: HTML5, CSS3, jQuery, Bootstrap, Tailwind, ReactJS, Redux, Redux Query

Programming Languages: JavaScript with ES6

TOOLS

Performance & Optimization: Google Lighthouse

Code Editors: VS Code, Codepen, Codesandbox, JSFiddle

Database: Fundamentals of MySQL and MongoDB [Watch crash-course videos, that should be enough to learn fundamentals]

Version Control: Github, Bitbucket, Gitlab [Learn Github and watch crash-course videos of Bitbucket and Gitlab, it's pretty much the same]

SOFT SKILLS

Problem Solving, Leadership, Verbal and Written Communication, Critical Thinking, Team Player. [Be ready with examples to support your claim, the interviewer might ask you to explain]

Preparation Material

In this section, we have divided the questions into Easy, Intermediate, and Hard. First, you will find the list of questions for all the difficulty levels. It is recommendable that you try to answer these before viewing the answer. The answers have been provided in the next section. Writing the answers is one thing that personally helps to prepare for interviews or any QnA session. When you write, it helps in 2 ways – it makes you *think*, helps you *write* more crisp answers which shows you know the topic well. You must understand that it's such a red flag when candidates over-explain because they do not know how much to explain for a certain question. Your answers have to be crisp, up-to-the-point and then you can ask follow-up questions to check if the interviewer is expecting more details.

Easy Questions

1. What is React?
2. What makes React so fast?
3. What is JSX? Can we use plain JavaScript to create React components?
4. How do you handle data inside a component?
5. How do you pass data from parent to child component?
6. What are the differences between state and props?
7. How do you update the state of the component?
8. Why you should not rely on the values of “`this.props`” and “`this.state`” for calculating the next state?
9. What is the difference between **stateful** and **stateless** components?
10. “In React, everything is a component”. Explain.
11. What are hooks in React?
12. Let’s say you are creating a TODO app and you want to store data on the browser. How would you do it?
13. What is the key prop? Why is it used?
14. How can you update state of a parent component from a child component?
15. What are the limitations of React?

Intermediate Questions

1. Explain the main phases of creation lifecycle of React components.
2. Explain the main phases of updation lifecycle of React components.
3. What is the use of `render()`?
4. Why should component names start with capital letter?
5. Why it is advised to update state after mount or updation and not in `render()` or `shouldComponentUpdate()`?
6. What is context API? Why is it used?
7. Why were hooks introduced?

8. Describe the `useState()` hook.
9. Describe the `useEffect()` hook.
10. What are controlled components?
11. What is ref? How do we use it? When should we use it?
12. What is routing in React?
13. Explain conditional redirection.
14. Can we implement a default page or not found page in React?
15. What is the use of `useNavigate()` hook?

Hard Questions

1. Is it possible to create lifecycle methods like `componentDidMount()`, `componentDidUpdate()`, and `componentWillUnmount()` using hooks? If so, how?
2. Explain code splitting.
3. Explain different components of React router.
4. What are error boundaries in React?
5. What is React Fiber? What are its main goals?
6. How can we do static type checking?
7. How will you programmatically trigger a click event in React?
8. How Virtual DOM works?
9. Let's say you have created a tic-tac-toe game. There are total 9 boxes on the game grid. When the user clicks on one grid item, how will you avoid re-render of remaining 8 boxes?
10. Let's say you have to create an infinite scrolling Instagram feed. What kind of performance challenges you will face? How would you fix them?
11. Let's say need to improve your application's load time. What steps you will take to fix it?
12. Your application is crashing with following message, "Too many re-renders. React limits the number of renders to prevent an infinite loop". How will you fix this?

13. Explain how Redux works.
14. Browsers do not support JSX then how do React applications work on browsers?
15. How to re-render the view when the browser is resized without using media queries?

Easy Questions [Solutions]

1. What is React?

Ans:

React is a frontend library written in JavaScript. It was developed by Meta(formerly Facebook) and later open-sourced. It is still maintained by Meta and dev communities. It is used to create high performance frontend applications. It can be used to create both single and multi page applications but mostly it is used for single page applications. Some big brands have built their applications on React. For example, Netflix, Airbnb, Facebook, and so on.

2. What makes React so fast?

Ans:

React uses something called a Virtual DOM which gives it a performance advantage. Rendering a webapp in a browser is an expensive task. Normally, when you update a webapp it re-renders the entire page but when you talk about React, it uses Virtual DOM to update the browser DOM with only the specific node/elements which are changed. This makes the updates faster and hence better performance.

3. What is JSX? Can we use plain JavaScript to create React components?

Ans:

It is a JavaScript extension which looks very much like HTML but it's not actually HTML. It's just JavaScript code made to look like HTML to make development easier. The learning curve is not that steep if you already are comfortable with HTML. It makes writing React component very easy and with significantly lesser code than writing plain JavaScript.

For example, we would write the following piece of code to programmatically create HTML elements using JavaScript

```
const para = document.createElement("p")
const paraText = document.createTextNode("Hello World")
para.appendChild(paraText)
```

The same thing can be written using JSX as shown in the following code snippet:

```
<p>Hello World</p>
```

Also, notice how similar it looks to plain HTML.

4. How do you handle data inside a component?

Ans:

State is used to manage data inside a component. Say for instance, you are creating a counter component where on every button click the count is increased by 1. This count value will be a part of the state variable and on every button click you will update the state which will trigger the component update lifecycle and the component will be re-rendered with updated value.

Earlier state was possible only in class-based components but now using hooks you can have state even inside the functional component.

5. How do you pass data from parent to child component?

Ans:

We use props to pass data from parent to child components. Props syntax looks very similar to HTML attributes. We simply add props while using the component and inside the component all the props key-values are available as an object.

Here is an example,

```
<ProfileCard fName="John" lName="Doe" />
const ProfileCard = (props) => {
  return (
    <p>{`${props.fName} ${props.lName}`}</p>
  )
}
```

6. What are the differences between state and props?

Ans:

Props	State
It is the data passed from parent component to child component.	It is the data handled inside the component
It is read-only or immutable.	It is mutable or it can be updated
It can be accessed by the child components.	It cannot be accessed by the child components.

7. How do you update the state of the component?

Ans:

We can use the `setState()` method to update the state of a component. Even though we have access to the state variable, we should not update it directly because that will not trigger the updation lifecycle and our component will not be re-rendered with new values.

```
state = {
  counter: 1
}
```

We should update it as shown in the following code snippet:

```
setState({counter: 2})
```

We should not update it directly as shown in the following code snippet:

```
this.state.counter = 2 //This will not trigger the update lifecycle
```

8. Why you should not rely on the values of “this.props” and “this.state” for calculating the next state?

Ans:

The reason is that `setState()` is an asynchronous function. If you use `this.state` to calculate the next state then you might end up using an older state value. For this specific usecase, `setState()` accepts an additional argument which is a callback function. This callback function gives us access to the latest value of state and props.

So rather than doing this:

```
setState({counter: this.state.counter})
```

Do this:

```
setState((state, props) => {
```

```
        return {counter: state.counter};  
    })
```

9. What is the difference between stateful and stateless component?

Ans:

Stateful Component	Stateless Component
These are components with internal state.	These are just dummy components with no state.
These are also known as smart components.	These are also known as dumb components.

10. “In React, everything is a component.” Explain.

Ans:

React applications follow a component-based architecture which means that the entire application is divided into smaller components. If you visualize a React application, then it'll look like a tree where each node will represent a component. The root node will be the `<App>` component with the other components as child nodes.

11. What are hooks in React?

Ans:

A hook is a special function that enables you to “hook into” React features in function-based components. Hooks were introduced in React v16.8. For example, `useState()` is an inbuilt hook that allows you to have state related functionalities in a functional components.

There are a lot of hooks provided by React like `useState()`, `useEffect()`, `useMemo()` and so on. A lot of hooks are also provided by different packages/libraries like `useLocation()`, `useReducer()` and so on.

12. Let’s say you are creating a TODO app and you want to store data on the browser. How would you do it?

Ans:

We can use local storage to store data in the browser.

13. What is the key prop? Why is it used?

Ans:

The key prop helps React identify which elements have updated, or are added, or are removed. All the elements that you generate programmatically should be given a key value to give them a stable identity. If you create elements by iterating on an array and you do not provide the key prop then React will throw a warning which can be seen in the browser's console. Just remember that every item in the list should be unique.

Let's try an example to understand this better.

Say you have an array of 3 items, [10, 20, 30]. You iterated on the array and rendered 3 cards. The first card has key=10, second has key=20 and third has key=30. Now we add one more item in the array, the updated array looks like [10, 15, 20, 30]. Again, you iterate on the array and render 4 cards such that each card has a key same as that of the array item. React knows that it has three cards with same key and only one new card is added and not four.

14. **How can you update state of a parent component from a child component?**

Ans:

We can create a function inside the parent component which will update the state of the component. We need to pass this function reference as a prop to the child component. When we call the function reference in the child component it will call the function in the parent component which will update the state. For example,

Parent Component:

```
const onUpdateName = (name) => {
  this.setState({fullName: name});
}

.....
<ChildComponent updateName={this.onUpdateName} />
```

Child Component:

```
.....
props.onUpdateName("John Doe");
```

15. **What are the limitations of React?**

Ans:

One of the biggest challenges faced by new React developers is due to the fact that React is a library and not a framework. React has given tools to create UI but building production level applications is a lot more than that. You end up using third party libraries to introduce functionalities like routing, complex state management and so on. This adds to the learning curve.

Intermediate Questions [Solutions]

1. Explain the main phases of creation lifecycle of React components.

Ans:

Mounting is the stage when the component is created and inserted in the browser DOM. This is the stage when the `render()` function is called for the first time.

The first step during the mounting of a component is initialization. This is where the component starts its journey by setting up the state and the props. The initialization process is done inside the constructor. Say your component is expecting props to set the initial state, you can do that inside the constructor as shown in the following code snippet:

```
class ProductCard extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      isWishlisted: this.props.wishlisted  
    }  
  }  
}
```

Due to the class field declaration introduced in ES6, the previous code snippet can also be written alternatively without the constructor as shown in the following:

```
class ProductCard extends React.Component {  
  state = {  
    isWishlisted: this.props.wishlisted  
  }  
}
```

Once the component is initialized, the `render()` method is triggered, and the code written inside the `render()` method is executed. If more components are used inside the `render()` method then those child components are rendered too.

When the current component along with its child components is rendered. React calls the final callback which is `componentDidMount()`. It just tells us that the component is mounted in the DOM. This is the stage when we should make API requests to fetch data and update the component state if required.

2. Explain the main phases of updation lifecycle of React components.

Ans:

The updation cycle is triggered when you call `setState()` inside the component or the props are updated by the parent component. There are 3 major lifecycle methods involved in the updation phase - `shouldComponentUpdate()`, `render()` and `componentDidUpdate()`.

When the component is updated the first callback method is `shouldComponentUpdate()`. This method returns a boolean value that decides whether the component should be updated or not. By default, it returns true, but we can conditionally return true or false to avoid any unnecessary re-renders. We will talk more about avoiding unnecessary re-renders in the “Performance Optimization” chapter.

If the `shouldComponentUpdate()` method returns false then the updating cycle is stopped. If the `shouldComponentUpdate()` method returns true only then the `render()` method is called. As you already know this is when the browser DOM is actually updated. When the `render()` method is called, it also renders all the child components of that component.

When the current component along with its child components is updated. React calls the final callback which is `componentDidUpdate()`. It just tells us that the component is updated in the browser DOM. This is the stage when we can make API requests to fetch data from the backend.

3. What is the use of `render()`?

Ans:

The `render()` method contains all the code which is needed to render the UI for a class-based component. Whenever a component is mounted or updated, all the code inside `render()` method is executed. This is the stage when Virtual DOM is created, and the diffing algorithms gets into play.

4. Why should component names start with capital letter?

Ans:

The reason is that the lowercase values are assumed to be HTML tags. When your component starts with a lowercase letter then React assumes that it is an HTML tag which is incorrect.

5. Why it is advised to update state after mount or updation and not in render() or shouldComponentUpdate()?

Ans:

Say you make an API request in the `constructor()` or `render()` method and the API fails. You would want to show some error message in the component for the failed API request but the problem is that your component might not be rendered at this point and you won't be able to show the error to the user.

6. What is context API? Why is it used?

Ans:

When we have info like current theme or current authenticated user, it usually needs to be passed to multiple components. We know to pass info between components we use props but the problem in this specific scenario is that a prop might be passed from the app component all way down to 10 levels. This creates a situation called props drilling where you have to pass props from one component through multiple components because it's needed in a deep child component.

To fix this situation, we can use Context. It gives us a way to make these values available in different components without having to pass them as props through every level.

7. Why were hooks introduced?

Ans:

The simple answer is that, hook allow state and lifecycle related functionality in functional components. There are other reasons why hooks were introduced in React:

- It enables us to reuse stateful logic between components. We can create our custom hook which can contain this reusable stateful logic and simply use it in all the required components.
- It enables us to structure a component into smaller functions based on relevant piece of logic rather than lifecycle methods.
- We get to avoid classes and all the troubles of “this” keyword.

8. Describe the `useState()` hook.

Ans:

`useState()` is an inbuilt hook provided by React. This hook allows us to create state variables and methods to update those state variables in function-based components. Checkout this code snippet:

```
const [counter, setCounter] = useState(0);  
.....  
<h1>{`Count: ${counter}`}</h1>  
<button onClick={setCounter(counter + 1)}>Increase  
Count</button>
```

In the preceding example, we have created counter as a state variable, and the `setCounter()` method will be used to update the counter value. We use destructuring syntax to create the state variable and the method to update it. We also pass an initial state value inside `useState()` hook. The data type of this value can be anything based on what we need in our state variable.

9. Describe the `useEffect()` hook.

Ans:

As the name suggests, the `useEffect()` hook is used to carry out a side effect each time there is a state change. By default, effects run after every completed render which includes the first and any subsequent render, but you can choose to fire them only when certain values have changed.

Here is the syntax for creating effects using the `useEffect()` hook:

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

```
useEffect(() => {
  //Callback function body
});
```

10. What are controlled components?

Ans:

When the component state is managed by the browser it is known as uncontrolled components. When the component state is managed by the React application then it is called controlled components.

Most common example for this is forms. When we implement forms there are two ways. We can either implement form elements as controlled components or uncontrolled components. It is preferred to create forms using controlled components but if your form is really simple where you do not require custom errors or any custom behavior then you can also implement forms as uncontrolled components.

11. What is ref? How do we use it? When should we use it?

Ans:

If you remember in Vanilla JS we used to add IDs to our HTML elements and used to access them using `getElementById` method but we do not access HTML elements that way in React. Refs is one of the ways to access elements in React. Refs provide a way to access DOM nodes or React elements created in the render method. Refs is not a replacement of ID attribute. We use Ref only when we cannot perform an action declaratively.

To create Refs in a classbased component we use the `createRef()` method provided by React as shown in the following code snippet:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.inputRef = React.createRef(); //We create a new
    ref called inputRef. "inputRef" is just a variable
    name, you can call it anything you want.
  }
  render() {
    return <input ref={this.inputRef} />; //We assign
    inputRef to the ref attribute of JSX element.
```

```
    }  
}
```

When a ref is passed to an element in render, a reference to the node becomes accessible at the current attribute of the ref as shown in the following code snippet:

```
const node = this.inputRef.current; //current property  
gives us access to the reference node. In this example,  
it'll give us access to the DOM node of the input element.
```

The value of the ref differs depending on the type of the node:

- When the ref attribute is used on an HTML element, the ref created in the constructor with `React.createRef()` receives the underlying DOM element node as its current property.
- When the ref attribute is used on a custom class component, the ref object receives the mounted instance of the component as its current property. For example, `<CustomClassComponent ref={this.mRef} />`
- You cannot use the ref attribute on function components because they don't have instances. For example, `<CustomFunctionalComponent ref={this.mRef} />`, this is not allowed.

To create refs in a function based component we use the inbuilt `useRef()` hook provided by React. The `useRef()` hook returns a mutable ref object whose current property is initialized to the passed argument (`initialValue`) as shown in the following code snippet. The returned object will persist for the full lifetime of the component.

```
import { useRef } from 'react'; //Import useRef hook  
const inputRef = useRef(null); //Create ref using useRef()  
hook  
<input ref={inputRef} /> //Attach ref to React element  
console.log(inputRef.current.value) //Access the element  
node using current.
```

12. What is routing in React?

Ans:

Routing is a process in which a user is directed to different pages based on their action or request. For example, on the Youtube homepage, you see a grid of video cards and when you click on any of the cards it redirects you to the video watch page. This redirection on click is called Routing.

When you are building React applications, even if you build a simple application like a Blog App even then you are going to need to handle routing. You will have multiple screens in the application like homepage, blog page, create/edit blog page, author page, and so on and each page will have its own URL which you will need to map with the right components.

React doesn't provide an inbuilt API to handle routing. That is why we use a package called react-router to handle routing.

13. Explain conditional redirection.

Ans:

You can programmatically redirect users to different endpoints based on conditions. Let's say you are building an "Edit Profile" page. This page should only be accessible if the user is logged in. So, you can add a condition in the routers such that if user is logged-in then load the edit profile page else redirect to the login page. This is called conditional redirection.

14. Can we implement a default page or not found page in React?

Ans:

Yes, we can implement a 404 page using react router. To load not found(404), something went wrong(500) and unauthorized(401) pages we need to create different components for each and setup the route component in the App component.

If we want to load a 404 page if the URL is not matching then we can use the "*" wildcard to match as shown in the following code snippet:

```
<Routes>
  <Route path="/" element={<Home />} />
  <Route path="/profile" element={<UserDetails />} />
  <Route path="*" element={<NotFound />} />
</Routes>
```

15. What is the use of useNavigate() hook?

Ans:

The **useNavigate()** hook was introduced in the React Router v6 to replace the **useHistory()** hook.

Earlier, the **useHistory()** hook was used to access the history object. It gave us access to the **push()** or **replace()** methods.

In the new version, navigation API provides a **useNavigate()** hook which is used to navigate programmatically in our React application.

Here is an example:

```
import { useNavigate } from "react-router-dom";
const navigate = useNavigate();
React.useEffect(() =>{
  if(!props.userLoggedIn) {
    navigate("/login");
  }
})
```

Hard Questions [Solutions]

1. Is it possible to create lifecycle methods like **componentDidMount()**, **componentDidUpdate()**, and **componentWillUnmount()** using hooks? If so, how?

Ans:

Yes, it is possible. Let's give them a try.

componentDidMount() means trigger when the component is mounted. We know that effects are triggered on mounting and any subsequent updates. The **useEffect()** hook also accepts a second argument which is a dependency array. If we keep this dependency array empty then this effect will only be triggered on mounting as shown in the following code snippet:

```
const Component = () => {
  React.useEffect(() => {
    console.log("Simulating componentDidMount()");
  }, []);
}
```

componentDidUpdate() means trigger once the component is updated. If you want to also trigger effect after every render then simply remove the second argument from the previous code snippet. This will trigger the effect both on initial render as well as any subsequent state change. To trigger only on state change, we can store a flag which can tell us whether the component was initially rendered or it was updated. If the component was being initially rendered then we can simply exit the effect without executing any further lines of code.

componentWillUnmount() means trigger when the component is about to be unmounted from the browser DOM. The **useEffect()** has a rule, if a function is returned from **useEffect()**, that function will be triggered only when the component is removed from the DOM. We could implement **componentWillUnmount()** using hooks as shown in the following code snippet:

```
const Component = () => {
  React.useEffect(() => {
    return () => {
      console.log("Simulating componentWillUnmount()");
    }
  }, []);
};
```

2. Explain code splitting.

Ans:

If you remember the base concept of Single Page Applications, there is an HTML file, bundled CSS file and a bundled JS file. Imagine you write a really large application maybe something like Amazon, it would have a huge file size for the JS bundle. Even if the bundle size is 10MB it's huge for your users especially on mobile devices. Try to download a 10MB file over a 4G network and you would know that it takes more than 30 seconds to download. If it takes 30 seconds for your webapp to load then you will lose majority of your users.

What's the solution for this huge JS bundle? Simple, we split the bundle into smaller chunks. If you are using create-react-app to build your React application then it automatically configures the bundling using webpack.

We can take this even a step further. Let's take Amazon for example, when the user lands on the homepage, they would not need the code for Product Details Page, User Account Page, Order History page and so on, right? We can do code splitting at the route level. This is called lazy loading. You do not load everything on application load, you load the code as required. We use `React.lazy()` to load components dynamically.

3. Explain different components of React router.

Ans:

React doesn't provide an inbuilt API to handle routing. That is why we use a package called react-router to handle routing

Here are a few things to remember to set up a simple route in your React application:

- Instead of an anchor tag, we use a component named `<Link>` provided by react-router-dom.
- To map components with URLs we use a component named `<Route>` provided by react-router-dom.
- `<Routes>` acts as a container/parent for all the individual routes that will be created in our app. Whenever the location changes, `<Routes>` looks through all its children `<Route>` elements to find the best match and renders the appropriate components.
- Wrap your entire app inside the `<BrowserRouter>` component provided by react-router-dom. The `<BrowserRouter>` uses HTML5 history API to manage the routing. It listens to URL changes and then handles the UI accordingly. It can either be used in index.js or app.js file.

4. What are error boundaries in React?

Ans:

These are class-based components that catch JavaScript errors anywhere in their children components and display a fallback UI instead of throwing an error that application crashed.

To convert a component into an error boundary we need to define a new lifecycle method called `componentDidCatch(error, info)` and static `getDerivedStateFromError()` :

```
class ErrorBoundary extends React.Component {
  state = {
    handleError: false
  };
  componentDidCatch(err, message) {
    console.log(message);
  }
  static getDerivedStateFromError(error) {
    return { handleError: true };
  }
  render() {
    if (this.state.handleError) {
      return <SomethingWentWrong />;
    }
    return this.props.children;
  }
}
```

We can use this error boundary component and wrap our app inside it.

```
<ErrorBoundary>
  <App />
</ErrorBoundary>
```

5. What is React Fiber? What are its main goals?

Ans:

React Fiber was introduced in React 16. It is the new reconciliation algorithm in React.

Main goals of React fiber:

- Improved performance
- Ability to split interruptible work in chunks.
- Ability to prioritize, rebase and reuse work in progress.
- Ability to return multiple elements from `render()`.
- Better support for error boundaries.

6. How can we do static type checking?

Ans:

For small projects we can use PropTypes from the prop-types package. For production level applications, it is recommended to use TypeScript for static type checking. TypeScript performs type checking at compile time and shows errors/warning in your code.

7. How will you programmatically trigger a click event in React?

Ans:

We can use ref to access the HTML object for the element and then use the `click()` method to trigger the event as shown in the following code snippet:

```
<button ref={btnRef}>Click Me!</button> //To setup ref  
.....  
this.btnRef.click(); //To trigger
```

8. How Virtual DOM works?

Ans:

As the name suggests, the virtual DOM is a virtual representation of the actual DOM. Just like the actual DOM, virtual DOM is a node tree that lists elements and their attributes and content as objects and properties.

Virtual DOM is a representation of the user interface. So, whenever the state of a component changes a new virtual DOM is created. The new virtual DOM is compared with the previous one to find the updated nodes. If there are some nodes that need to update then only those are updated in the browser DOM. If there is no difference between the old and the new virtual DOM then the browser DOM is not updated. This makes the performance far better when compared to manipulating the browser DOM directly.

9. Let's say you have created a tic-tac-toe game. There are total 9 boxes on the game grid. When the user clicks on one grid item how will you avoid re-render of remaining 8 boxes?

Ans:

We can use memoization to memoize the rendered components. Let's assume that we have used functional component to render grid and each grid item is rendered using a component. We can use the `useMemo()` hook to render the grid items and add a condition that it should re-render the grid item only when the value of that grid item is

updated. This way when the user clicks on a grid item only the value for that grid item will be updated and the other 8 grid items will not be re-rendered.

10. Let's say you have to create an infinite scrolling Instagram feed. What kind of performance challenges you will face? How would you fix them?

Ans:

There are two main performance challenges that we will face:

- Initially, the list will be small so we don't need to worry about handling a huge list but the image size could be a challenge. Imagine the images are high resolution and are heavy in size. We need to ensure that we are getting minified images from backend and in the required dimensions.
- With each scroll the length of the feed will increase which means more components gettings rendered which will definitely add to the performance issues. We can use the concept of windowing to solve this problem. This ensures that only the items visible on the screen are in memory. We can use packages like react-virtualized or react-window for this.

11. Let's say need to improve your application's load time. What steps you will take to fix it?

Ans:

Our application could be slow either due to high download size or unnecessary re-renders. We'll need to follow the following steps:

1. Use “Google Lighthouse” to benchmark our application and analyze where we are losing the performance.
2. If the performance loss is due to high bundle size then we optimize by code splitting.
3. If the performance loss is due to large content download then we optimize the media files and reduce their size which will result in smaller download size.
4. Once we have fixed the major issues on Lighthouse then we analyze our application using React profiler provided by React-

dev-tools. This will help us identify the components which are taking more time to render or if there are some component which are being re-rendered unnecessarily. We must fix those.

12. Your application is crashing with following message, “Too many re-renders. React limits the number of renders to prevent an infinite loop”. How will you fix this?

Ans:

This message is usually received when there is a `setState()` loop. We need to identify what is resulting in this and fix it. General rule of thumb, we do not put `setState()` inside `render()` because that will definitely create a loop of updation cycle.

13. Explain how Redux works.

Ans:

Redux revolves around 4 main concepts:

- Global Store
- Actions
- Reducer
- Subscriptions

Let's summarize the entire Redux flow:

1. It all starts with an action. To trigger an action we use the `dispatch` method provided by the Redux store.
2. Once the action is dispatched, the action object goes to the reducer.
3. Reducer checks if it knows how to update the state based on the action type and updates the state accordingly.
4. Once the global store is updated, it broadcasts the updated data to all the subscribers.

14. Browsers do not support JSX then how do React applications work on browsers?

Ans:

Babel converts JSX to plain JavaScript. Behind the scenes JSX is converted to `React.createElement()` methods. This is why the React module is imported from the React library.

The `React.createElement()` method takes three arguments:

- The element name. For example, p, h1, div, and so on.
- An object which holds all the attributes in the element.
- An array of the element's child elements.

Here is an example:

JSX code:

```
<div>
  <h1>Hello</h1>
</div>
```

JavaScript code after Babel conversion:

```
React.createElement("div", null, React.createElement("h1",
  null, "Hello"));
```

15. How to re-render the view when the browser is resized without using media queries?

Ans:

We can listen to the resize event of window object as shown in the following code snippet:

```
setScreenSize = () => {
  this.setState({width: window.innerWidth, height:
    window.innerHeight });
}
componentDidMount() {
  window.addEventListener("resize",
    this.setScreenSize);
}
componentWillUnmount() {
  window.removeEventListener("resize",
    this.setScreenSize);
}
render() {
  return(
```

```
<div>`${this.state.width} ${this.state.height}`  
</div>  
)  
}
```

That's all the questions for your interview preparation from this book. We hope you understood the React concepts and we wish you all the best for your interview!

Index

Symbols

401, 404, and 500 pages, handling [144](#)

A

- accumulator [21](#)
- action creator [198](#)
- API Endpoints [97, 98](#)
- API requests
 - creating [99-101](#)
- app component [34](#)
- async actions
 - using middleware [199-203](#)
- async keyword
 - for creating asynchronous functions [167, 168](#)
 - for restructuring our existing code [168-170](#)
- await keyword
 - for creating asynchronous functions [167, 168](#)
 - for restructuring our existing code [168-170](#)
- Axios [99](#)
 - global setup [103-105](#)
 - request intercept [105](#)
 - response intercept [106](#)
 - using, for DELETE request [103](#)
 - using, for POST request [101, 102](#)
 - using, for PUT request [102](#)

B

- backend apps [2](#)
- backend connection [96, 97](#)
- backend frameworks and libraries [5](#)
- browser DOM [82](#)

C

- callback function [162-164](#)
- Callback Hell [164](#)
- class-based components [66-73](#)
- Codepen [7](#)
 - JavaScript settings [9](#)
 - new pen [8](#)

- pen settings [8](#)
- code splitting [230-232](#)
- component
 - exporting [56](#)
 - importing [56](#)
 - rendering [57](#)
- component lifecycles [84](#)
 - mounting [84-89](#)
 - unmounting [92](#)
 - updating [89-92](#)
- components [52](#)
- conditional redirect [140-142](#)
- controlled component [148](#)
 - using, for forms [157-159](#)
- create-react-app [35](#)
 - using [36](#)
- CSS modules [63](#)
 - using [64](#)
 - working [63, 64](#)
- custom hooks
 - creating [123-125](#)

D

- dark mode
 - designing [279, 280](#)
- database [2](#)
- default export
 - using [60, 61](#)
- directory structure, React project
 - node_modules [38](#)
 - package.json [39](#)
 - public [38](#)
 - src [38](#)
- DOM
 - versus, virtual DOM [82-84](#)
- Dribbble
 - URL [35](#)
- dynamic elements
 - rendering, objects and list used [46-49](#)
- dynamic URLs
 - handling [137-140](#)

E

- ECMAScript [6](#)
- ES5 code [7](#)
- ES6
 - features [6, 7](#)
- ES6 code [7](#)

external styles [44-46](#)

F

filter() method [23](#)
findIndex() method [23](#)
find() method [23](#)
frontend apps [2](#)
 versus, backend apps [2, 3](#)
frontend frameworks and libraries [5](#)
Functional components [52-55](#)

G

geometric elements
 working with [277](#)
glassmorphism [264](#)
 implementing [264-270](#)
gradients [274](#)

H

Hooks [110](#)
 class-based component, converting to functional [111-114](#)
 provided by React Router [140](#)
 useContext [114](#)
 useEffect [114](#)
 useMemo [115](#)
 useReducer [114](#)
 useRef [114](#)
 useState [114](#)
HTTP [97](#)
 methods [98](#)
 response codes [98](#)

I

inline styles [44, 45](#)

J

JavaScript, for ReactJS [6](#)
 array function: filter() [23](#)
 array function: map() [19, 20](#)
 array function: reduce() [20-23](#)
 array functions: find() and findIndex() [23-25](#)
 arrow functions [13, 14](#)
 classes [25, 26](#)
 destructuring [17, 18](#)

- inheritance [26](#), [27](#)
- methods [25](#)
- properties [25](#)
- rest property [14](#), [15](#)
- spread property [15](#), [16](#)
- templates strings [12](#), [13](#)
- variables, creating with let and const [10-12](#)

JSX [39-44](#)

JSX Fragment [55](#)

L

- linear gradient [274](#)
 - adding [274-276](#)
- logged-in state
 - restraining [160-162](#)

M

- map() function [19](#)
- memoization [223](#)
- mounting lifecycle [84-89](#)
- multi-page applications [35](#)

N

- named exports [61](#)
 - using [61](#), [62](#)
- neomorphism [270](#), [271](#)
 - adding [272](#), [273](#)
- network requests
 - scalable code architecture [106](#)
- NodeJS
 - installing [36](#)

P

- pastel backgrounds [278](#)
- performance
 - measuring [218](#)
- ProductCard component [52](#)
- production build [208](#)
 - creating [209-211](#)
- project structure
 - refactoring [170-172](#)
- promise [164](#)
 - creating [165](#), [166](#)
 - multiple promises, chaining [166](#)
- props [58-60](#)
 - passing, to components [74-78](#)

PureComponents [227-229](#)

Q

query params
handling, in URLs [143](#)

R

radial gradient [276, 277](#)
React application
 hosting [212, 213](#)
React elements [52](#)
React interview process [284](#)
 easy questions [286](#)
 easy questions, solutions [288-293](#)
 hard questions [287, 288](#)
 hard questions, solutions [300-307](#)
 intermediate questions [286, 287](#)
 intermediate questions, solutions [293-300](#)
 preparation material [285, 286](#)
 resume template [284](#)
ReactJS [32](#)
 component-based design [32-34](#)
 single-page, versus multi-page Web apps [34, 35](#)
React module
 default export [55](#)
 named export [56](#)
React Profiler [218-222](#)
React project
 creating [35-38](#)
 directory structure [38](#)
React-Router [130, 131](#)
 setup, in React app [131-136](#)
reduce() method [20-23](#)
Redux [176](#)
 action creators [198, 199](#)
 actions [176, 177](#)
 async actions, using middleware [199-203](#)
 configuring, in React app [184, 185](#)
 fetch state, from global store in components [187-190](#)
 global store [176](#)
 global store, creating [185, 186](#)
 global store, updating from components [190-192](#)
 implementing, with Hooks [196-198](#)
 installation [179-184](#)
 multiple reducers, handling [192-196](#)
 reducer [177, 178](#)
Refs [46, 154](#)
 creating [154, 155](#)

- using [156](#)
- request intercept [105](#)
- response intercept
 - using [106](#)
- responsive components [65](#)
- routing [130](#)

S

- search parameters [143](#)
- shouldComponentUpdate() [229, 230](#)
- single-page application [34](#)
- stateful component [65, 66](#)
- stateless component [65, 66](#)

U

- UI/UX [236](#)
 - alignment issues [256, 257](#)
 - colors [258, 259](#)
 - contrast [252, 253](#)
 - fundamental concepts [248](#)
 - scale [254, 255](#)
 - tools [236](#)
 - typography [260](#)
 - visual hierarchy [260, 261](#)
 - white space [249-252](#)
- UI/UX project [237](#)
 - design [240](#)
 - information architecture [238](#)
 - prototype, creating [241-248](#)
 - user research [237, 238](#)
 - Wireframe [239, 240](#)
- uncontrolled component [148](#)
 - using, for forms [148-157](#)
- unmounting lifecycle [92](#)
- updating lifecycle [89-92](#)
- useCallback() hook [225-227](#)
- useEffect() hook [117-123](#)
- useMemo() hook [223-225](#)
- user experience (UX) [34](#)
- useState() hook [115-117](#)

V

- virtual DOM [82, 83](#)

W

- Web development [1](#)

[CSS 3](#)

CSS layout, with code [4](#)

[HTML 3](#)

HTML layout, with code [3](#)

[JavaScript 3](#)

JavaScript layout, with code [4](#)

Webpack [213-215](#)