

React.js Design Patterns

Learn how to build scalable React apps with ease

Anthony Onyekachukwu Okonta



React.js Design Patterns

Learn how to build scalable React apps with ease

Anthony Onyekachukwu Okonta



React.js Design Patterns

*Learn how to build scalable
React apps with ease*

Anthony Onyekachukwu Okonta



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 978-93-5551-367-0

www.bpbonline.com

Dedicated to

My beloved Parents:

Augustine Chukwuji Okonta

Bridget Ejime Okonta

&

My lovely siblings

Esther Ogechi Esiejobor

Gloria Nkechi Okonta

Gladys Siwekwu Okonta

Mary Uche Emioma

Judith Chinwe Ajudua

About the Author

Anthony Onyekachukwu Okonta is a software developer with 10+ years of experience in Software Development with different coding languages like vb.net, C#, Java, and JavaScript. He has worked in different industries, which include financial, education, and fintech. He graduated in 2010 from Novena university, Ogume, Delta state, Nigeria, where he obtained a bachelor's in information technology. Currently, he works as a senior software engineer for Technaura Systems GMBH. He enjoys playing football, swimming and hanging out with friends.

About the Reviewers

Daniel Chukwurah is a Web and Mobile Developer at Afrinvest West Africa. At Afrinvest, he develops investment solutions that are changing how people interact with money to achieve financial freedom. He enjoys building scalable web and mobile applications with a good user experience, that meets business goals and transforms lives. Prior to Afrinvest, he worked at Sidmach Technologies Limited as a Software Engineer. Daniel is a self-taught developer and currently studying for his MBA at Nexford University. He also enjoys speaking at tech events.

Ravi Chaudhary is a Technical Lead with working knowledge as a Fullstack Developer lead with rich experience in handling clients communication, along with that involved in various Architectural design decisions and building real-time complex web projects, API development using Node JS (MEAN Stack, Meteor), Databases like Mongo DB, Mysql along with Redis, Socket.io, Kafka, LAMBDA functions, Server Management(AWS, Digital Ocean along with techniques like Auto Scaling, Load balancers), and front-end frameworks like ReactJS.

He has hands-on experience in Agile methodologies, and scrum implementation and has also worked closely with Product, and Business teams to scale up the platform and improve efficiency.

LinkedIn Profile -

<https://www.linkedin.com/in/ravi-chaudhary-js>

Acknowledgement

There are a few people I want to thank for the continued and ongoing support they have given me during the writing of this book. First and foremost, I would like to thank my parents for continuously encouraging me to write the book — I could have never completed this book without their support.

I am also grateful to Daniel Chukwurah, Augustina Nwahiri, Chidimma Udensi, Augusta Madubuko, Sandra Odiete, and others too numerous to mention who gave me tremendous support throughout the writing of this book.

My gratitude also goes to the team at BPB Publications for being supportive enough to provide me with quite a long time to finish writing the chapters of this book.

Preface

This book covers many different aspects of building applications with React.js, the importance of building scalable applications with React.js, and its best practices. This book has a deep dive into some of the bad practices we commonly do when writing our applications and shows us better ways how to architect our applications.

This book takes a practical approach for React.js learners. It covers a few Real-time industry examples as well. It will cover information such as JavaScript, which Typescript is basically used for building React applications. It also covers state management, lifecycle methods, styling, and practical ways to structure your files to make them more readable and accessible.

This book is divided into **9 chapters**. They will cover basics and advanced React.js.

Chapter 1 This chapter will cover different patterns in writing React applications. Yes, React is opinionated, although it gives you the liberty to build and structure your applications according to your taste or that which fits your organization. However, there are some very bad practices that are not good and can have an effect on your application as your application begins to grow.

This chapter will identify common bad practices out there and suggest better practices to help build scalable React applications. We will be using JavaScript and TypeScript for this chapter. — This seems to be redundant since TypeScript was not used in any example illustrated, and React is built on JavaScript.

Chapter 2 This chapter will cover how to write components. React is a component-based library whereby different reusable components are used to build an application. Basically, this chapter will outline the different ways of writing components which include functional and class-based components.

Although we will focus more on functional-based components because it is the most recent in the React Docs, there are still applications built with

class components. We will be using JavaScript and TypeScript for our examples in this chapter.

Chapter 3 In this chapter, we will learn the to write clean and maintainable code. We would also look at the patterns used to achieve clean and maintainable code. We would conclude by learning how to use TypeScript to build react applications.

Chapter 4 This chapter will cover what React hooks are, and we will look at the most common hooks used in React. We would also look at how to write custom hooks and their benefits when building applications. We would see how hooks and custom hooks can be combined in a single application.

Chapter 5 will cover how to style React components. This will include pre-processors like Scss and Less. We will also cover styled-components and inline styles. Also, we would cover modular CSS.

Chapter 6 will look at what server-side rendering is, when to use it, and its pros and cons. We would also look at a popular server-side framework (Next.js) and how it differs from the client-side react library.

Chapter 7, we will look at how to fetch data in our React application. We would also look at the different types of libraries/APIs we can use to fetch data remotely into our application. We would also look at the difference between REST API and GraphQL and how to cache the fetched data and use it in our application. We would also look at promises, their benefits, and what it solves in our react application.

Chapter 8 will cover ways how to build scalable and high-performance applications in React. We will look at some best practices, concepts, and benefits of building such applications in React.

Chapter 9 will cover the testing of react components. We will be using Jest and Cypress for this chapter. Here, we will cover functional testing, integration testing, and end-to-end testing.

Code Bundle and Coloured Images

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

<https://rebrand.ly/iqt10w8>

The code bundle for the book is also hosted on GitHub at <https://github.com/bpbpublications/React.js-Design-Patterns>. 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. React Anti-Patterns and Best Practices

Introduction

Structure

Objectives

Bad practices in building a React application

Inline CSS

Large component

Props drilling

Do not pass all props

Not using the key in the map loop

Using the index as a key in the map loop

Nested ternary statements

Ambiguous naming

Do not mutate state directly

Do not style HTML tags directly

What are the best ways of architecting a react application?

Styling

Preprocessor (SASS/LESS)

File/folder structure

Naming conventions

Rendering with useMemo and useCallback

Use Prettier

Use Linter and its rules

Use try...catch

Log errors

React error boundary

Test your code

Component name must be in Pascal case

Extract reusable logic

Benefits of using JavaScript in React application

Benefits of using TypeScript in React application

Conclusion

2. Writing Components

Introduction

Structure

Objectives

Naming conventions

Camel case

Pascal case

Functional component

useState

useRef

useEffect

Class component

ComponentDidMount

ComponentDidUpdate

Best way to structure a react component

Data sharing between components

Props

Context API

Provider

Consumer

Redux

Store

Reducer

Actions

useReducer

Controlled versus uncontrolled components

Conclusion

3. Clean and Maintainable Code

Introduction

Structure

Objectives

Writing a clean code

Business continuity

Easy to read and understand

Achieving clean and maintainable code

Writing maintainable code

Design patterns

Writing testable codes

Checking for errors

Log errors for easy bug tracking and fixes

Using TypeScript to build applications

Conclusion

4. React Hooks

Introduction

Structure

Objectives

React hooks

useState

useReducer

useEffect

useLayoutEffect

useMemo

useCallback

useRef

useContext

Custom hooks

Conclusion

5. Styling

Introduction

Structure

Objectives

Styling a component

Types of styling

Inline CSS

External CSS

Sass

Bootstrap

Styled components

Tailwind CSS

Conventions of writing styles

BEM

Conclusion

6. Server-Side Renderings

Introduction

Structure

Objectives

What is Server-Side rendering?

When to use server-side rendering?

Pros of using Next.js

Cons of using Next.js

Pre-rendering in Next.js

Static generation

Static page without data

Static page with data

GetServerSideProps

Next.js versus React.js

Maintainability

TypeScript support

Configuration

Server-side rendering

Image optimization

Server-side rendering with Node.js and Express.js.

Conclusion

7. Data Fetching

Introduction

Structure

Objectives

Fetching data in React

Fetch

Axios

GraphQL

Rest API versus GraphQL

Caching data

Memoization

useMemo

useCallback

State management
Promises
Conclusion

8. Building Scalable Applications

Introduction
Structure
Objectives
Conditional rendering
Ternary operation
Logical operation
If else operation
Memoization
useMemo
useCallback
TypeScript
File organization
Separation of concerns
Higher order component
Solid principles
Conclusion

9. Testing

Introduction
Structure
Objectives
How to test components?
Functional test
Rendering test
Testing with Jest
React testing library
Regression test
End-to-end test with Cypress
Integration test with Jest
API mocking with Jest
Conclusion

[Index](#)

CHAPTER 1

React Anti-Patterns and Best Practices

Introduction

This chapter will cover different patterns in writing React applications. Yes, react is opinionated, although it gives you the liberty to build and structure your applications according to your taste or which fits your organization. However, there are some very bad practices that are not good and can have an effect on your application as your application begins to grow.

This chapter will identify common bad practices out there and suggest better practices to help build scalable React applications. We will be using JavaScript and TypeScript for this chapter. This seems to be redundant since TypeScript was not used in any example illustrated and React is built on JavaScript.

Structure

The following topics are to be covered in this chapter:

- What bad practices/ways in architecting a react application?
- Best practices in architecting react applications?
- Benefits of using TypeScript in React applications

Objectives

The objective of this chapter is to look at the different ways of building a react application and determine the good from the bad. It will also give us an insight into the benefits of JavaScript and TypeScript in react applications.

Bad practices in building a React application

React is an opinionated library created by Facebook to help build applications with components. Yes, being opinionated gives one the liberty to design it how he or she wishes. The application can be structured however you want and will still function. We will look at some of the bad practices that we sometimes engage in when using React but are not aware of.

Inline CSS

Inline styles or inline CSS is simply styling your component or page from within the component code. Inline CSS means defining the CSS properties using a style attribute with an HTML tag or adding the styles directly to the element. That string will contain CSS property value pairs; each property will be separated from the other by a semicolon.

Let us take a look at an example that follows:

```
1. import React from 'react'
2.
3. const App=()=>{
4.
5. return (
6. <p style={{color: 'red'}}>This is just a test application
   </p>
7. );
8. };
9.
10. export default App
```

In the preceding example, we changed the default color of the paragraph tag from black to red. This is a simple example with relatively less hassle because it involves simply changing the paragraph color. It is easy to implement—a line of code and the color of the paragraph changes from black to red.

But what if we want this red color to be applied to all paragraphs in the component? Let us see what happens when we have multiple paragraphs.

Refer to the code that follows:

```
1. import React from 'react'
2.
3. const App=()=>{
4.
5.
6. return (
7. <p style={{color: 'red'}}>This is just a test application
  </p>
8. <p style={{color:'red'}}>This is just a test application 2
  </p>
9. );
10. };
11.
12. export default App
```

Although the previous words, it is not recommended for the following reasons that it should be used because of the following reasons.

- It cannot be reused.
- It makes the code unreadable when overused.
- It hampers the performance; on each re-rendering, the style object will be recomputed.

[Large component](#)

React is a library that is used to build large applications using reusable components. These components are the building blocks of the application. Often in our applications, we tend to write components that do everything from logic to display rendering. This can make the component too long and unreadable. When a component is unreadable, debugging and fixes become almost impossible.

Let us take a look at another example:

```
1. import React from "react";
```

```
2. const App = () => {
3.   const userList = [
4.     {
5.       id: 1,
6.       name: "Anthony Okonta",
7.       country: "Nigeria",
8.     },
9.     {
10.      id: 2,
11.      name: "Sandra Odiete",
12.      country: "USA",
13.    },
14.  ];
15.
16.  const userStateList = [
17.    {
18.      id: 1,
19.      state: "Lagos",
20.    },
21.    {
22.      id: 2,
23.      state: "Alabama",
24.    }
25.  ];
26.  return (
27.    <>
28.      <p style={{ color: "red" }}>This is just a test
application </p>
29.      {userList.map((item) => {
30.        return (
31.          <ul key={item.id}>
32.            <li>{item.name}</li>
33.          </ul>
34.        );
35.      })}
36.      {userStateList.map((item) => {
37.        return (
```

```

38.         <ul key={item.id}>
39.             <li>{item.state}</li>
40.         </ul>
41.     );
42.   } })
43. </>
44. );
45. };
46.
47. export default App;

```

From the preceding example, you can see that this will be an issue. This is because the component has multiple loops of different arrays, and all the results will be rendered in the component. Over time, the component will become unreadable and could cause re-rendering of the component.

Props drilling

Props is one of the building blocks in react application. We often need this because we must share data between components. We might even need to pass data from the parent component down to a child component far below in the application tree. This is called **props drilling**. Do not use props for this because you will need to pass the parameter(s) on every component till it gets to the child component that needs it.

To solve such problems, you will need to use **ContextApi** or **Redux**, depending on the size of your application.

Do not pass all props

When passing props to the child component, we are tempted to pass all props to the child component. This can cause serious performance issues because not all the props are required. Let us have a look at an example to understand this:

```

1. import React from 'react';
2.
3. const App={()=>{
4.
5.

```



```
6.   return (  
7.     <Component {...props} />  
8.   )  
9.  
10. export default App;
```

[Not using the key in the map loop](#)

When we want to render a list to our users, we often use the map method that loops over the list or array and displays it to the user. However, each item in the list needs to be unique because react uses the key to track all the records in the DOM. If this is not there, react cannot know what was added, removed, or modified. Let us look at the following example:

```
1. import React from 'react';  
2.  
3. const App=()=>{  
4.   const users=[  
5.     {  
6.       name:'Anthony Okonta',  
7.       country:'Nigeria'  
8.     },  
9.     {  
10.      name:'Chizoba Iwumune',  
11.      country:'Nigeria'  
12.    },  
13.  
14. ]  
15.  
16. return (  
17.   {  
18.     users.map(item=>{  
19.       return <ul>
```

```

20.             <li>{item.name} </li>
21.             <li>{item.country} </li>
22.         </ul>
23.     })
24. }
25. )
26.
27. export default App;

```

Using the index as a key in the map loop

In most of our applications, we need to fetch a list from an API to render to the user or display a static list to the user. In this case, we mostly use the **map()** method. The **map()** method takes in the current value, index (optional) and array (optional) and returns an array. This array will need a key, so that react can properly distinguish between elements using its Reconciliation algorithm.

This key is unique, and we tend to use the index of items in the array that starts from zero (0) as the key.

Kindly see the following example:

```

1. import React from "react";
2.
3. const App = () => {
4.   const userList = [
5.     {
6.       id: 1,
7.       name: "Anthony Okonta",
8.       country: "Nigeria",
9.     },
10.    {
11.      id: 2,
12.      name: "Chidimma Jane Udensi",
13.      country: "USA",
14.    },
15.    {

```

```

16.      id: 3,
17.      name: "Krishna Ravi",
18.      country: "India",
19.    },
20.  ];
21.
22.
23.
24.  return (
25.    <>
26.      <p style={{ color: "red" }}>This is just a test
application </p>
27.      {userList.map((item,index) => {
28.        return (
29.          <ul key={index}>
30.            <li>{item.name}</li>
31.            <li>{item.state}</li>
32.          </ul>
33.        );
34.      })}
35.
36.    </>
37.  );
38. };
39.
40. export default App;

```

Do not do this

The reason why this is not an effective method is that React does not understand which item was added/removed/reordered, as the index is given on each render based on the order of the items in the array. Although it is usually rendered fine, there are still situations when it fails.

So basically, when there is a re-render, the component is destroyed and recreated. This will cause some inconsistency when rendering the list.

[Nested ternary statements](#)

Ternary operators are very good when you want to do a conditional rendering of a component. But having nested operations can be very difficult to read because it is very messy. Please avoid it. Let us look at the following example:

```
1. import React,{useState} from 'react';
2.
3. const App=(props)=>{
4.   const [data,setData]=useState(0);
5.   const ComponentOne=()=>{
6.     return <p>Component One </p>
7.   }
8.   const ComponentTwo=()=>{
9.     return <p>Component Two </p>
10.  }
11.   const ComponentThree=()=>{
12.     return <p>Component Three </p>
13.   }
14.   return (
15.     <div>{data===1?<ComponentOne/>:data===2?<
ComponentTwo/>:<ComponentThree/>}</div>
16.   )
17. }
18.
19. export default App;
```

Please do not do this

Although this works the code becomes unreadable, and one can easily miss their way because of the nested ternary operators. The best way to solve this is to create different ternary statements or use an **if...else** statement. Let us look at the correction as shown in the following:

```
1. import React, { useState } from "react";
2.
3. const App = (props) => {
4.   const [data, setData] = useState(0);
5.   const ComponentOne = () => {
```

```

6.     return <p>Component One </p>;
7.   };
8.   const ComponentTwo = () => {
9.     return <p>Component Two </p>;
10.  };
11.  const ComponentThree = () => {
12.    return <p>Component Three </p>;
13.  };
14.  return (
15.    <div>
16.      {data === 1 && <ComponentOne />}
17.      {data === 2 && <ComponentTwo />}
18.      {data === 3 && <ComponentThree />}
19.    </div>
20.  );
21. };
22.
23. export default App;

```

This is a cleaner and less ambiguous way of doing it instead of using nested ternary statements.

[Ambiguous naming](#)

Most times, we are not conscious of how we name our variables, folders, files, and so on in our applications. We make up names as it comes to mind. This is either because we want to meet the deadline, or we want to make the application logic work asap! Basically, we have two types of naming conventions, which are as follows:

- Pascal case
- Camel case

Regardless of the case your organization or project sets as the standard, have in mind that whatever you name a folder, variable, function, or method

should make some sense, not just to you but to the other people who will pick up your code and review or work with them subsequently.

Do not mutate state directly

One of the key things in React is its state of immutability. This means that you should not change it directly but rather change it by creating a new object using the **setState**. Let us analyze the following code to further buttress this point:

Do not do this

```
1. import React,{useState} from "react";
2.
3.
4.
5. const App = () => {
6.   const [data,setData]=useState('');
7.   const onChange=(event)=>{
8.     data=event.target.value;
9.   }
10.  return <input type='text' onChange={onChange}/>
11. };
12.
13. export default App;
```

The preceding example shows that we have a state value called **data** that accepts a string. It is initialized to have an empty value. So, we created an arrow function that would be called when the input value is changed and then update our data value which was initially an empty string.

This violates the rules of react state immutability. For us to correct this, we must call the **setData** and pass the corresponding value from the input as its argument. Kindly see the following correction to the preceding example:

```
1. import React,{useState} from "react";
2.
3.
4.
5. const App = () => {
6.   const [data,setData]=useState('');
```

```

7.   const onChange=(event)=>{
8.       setData(event.target.value)
9.   }
10.   return <input type='text' onChange={onChange}/>
11. };
12.
13. export default App;

```

OR

```

1. import React,{useState} from "react";
2.
3.
4.
5. const App = () => {
6.   const [data,setData]=useState('');
7.   return <input type='text' onChange={(event)=>
      setData(event.target.value)}/>
8. };
9. export default App;

```

The preceding examples are the same, just that one is an anonymous function, and the other is not. They will both give you the same result. We are calling the **setData** method and passing the input string to it as an argument, thereby following the rule of React's immutability.

Now, we have an idea of what react state immutability is. Let us look at how we would handle an array.

Do not do this

```

1. import React,{useState} from "react";
2.
3.
4.
5. const App = () => {
6.   const [data,setData]=useState([]);
7.   const onChange=(event)=>{

```

```
8.     data.push(event.target.value);
9.   }
10.   return <input type='text' onChange={onChange}/>
11. };
12.
13. export default App;
```

Again, this would work, but it has so many issues because it is mutating the array directly. It violates the state immutability of React. So, to solve this, we would make a copy of the array using the spread operator and push the new data to it. Kindly find mentioned as follows:

```
1. import React,{useState} from "react";
2.
3. const App = () => {
4.   const [data,setData]=useState([1,2,3]);
5.   const onChange=(event)=>{
6.     setData([...data,Number(event.target.value)])
7.     console.log(data)
8.   };
9.   return <input type='text' onChange={onChange}/>
10. };
11.
12. export default App;
```

Our line 7 will output to console the previous values and whatever we enter in the input textbox. Although we converted it to a number before appending.

Do not style HTML tags directly

Most times, we are tempted to style our applications by giving them different tags same font size, color, and so on. We do this because it makes our lives easy, especially since we know what the application would look like. This is fun and easy to do because we just have to do it once and forget about it. Look at the following example:

```
1. p{
2.   color:red
3. }
```



```
4. div{
5.   background-color:green
6. }
```

This is good, but it poses great danger. It can cause style overrides in your application. The best way to avoid this is to use classes or Ids for styling.

Avoid using divs unnecessarily

We are tempted to use divs most of the time in our application rendering. When we write our component, the first HTML tag that comes to our mind is to insert a **div** in the render for the class component or return it for our functional component. Although this works, it does not give the browser enough information on what is happening. There is a concept called **semantic HTML**, which gives the browser enough information about the section in our react application. The essence of this semantic HTML is that it gives meaning to our HTML tags. Examples of these tags are header, section, nav, and so on. The HTML semantic tags also aid in SEO. Following are some examples:

```
1. import React from "react";
2.
3. const App = (props) => {
4.
5.   return (
6.     <div>
7.       This is a test
8.     </div>
9.   );
10. };
11. export default App;
```

Try to avoid the previous example. Instead, emulate the following example:

```
1. import React from "react";
2.
```

```
3. const App = (props) => {  
4.  
5.   return (  
6.     <section>  
7.       This is a test  
8.     </section>  
9.   );  
10. };  
11.  
12. export default App;
```

Do not access props directly

We need props in react application to pass value from one component to another. Most times, we access these values in a certain way which may work but is not a clean way of accessing these values. Let us look at the following example:

```
1. import React from 'react';  
2.  
3. const App=(props)=>{  
4.  
5. return (  
6. <div>{props.id}</div>  
7. )  
8. }  
9.  
10. export default App;
```

In the preceding example, we are accessing the **id** property from the **props** object. This property has been passed from a parent component to the **App** (child) child component. This will work, but it can become messy when used in multiple places. However, we can use object destructuring to handle this. Let us have a look at the following example:

```
1. import React from 'react';
2.
3. const App=(props)=>{
4.
5.   const {id}=props;
6.   return (
7.     <div>{id}</div>
8.   )
9. }
10.
11. export default App;
```

We have successfully destructured the props object in line 5 by getting the property id out of it. Here, the id property can now be used across the component as against using props.id.

What are the best ways of architecting a react application?

We have looked at some of the bad practices in architecting a react application, and now we will examine some of the best ways to structure a react application. In the beginning, we all agreed that react is an opinionated library, which gives one the liberty to write or structure his/her application, so the following practices are not entirely the worlds accepted practices, but they help in making your applications scalable, testable, and less prone to errors.

Styling

A crucial component of building applications is styling. A component's style plays a crucial role in creating a pleasing interface for the user. There are useful styling techniques for applications and component design.

- **Modular CSS:** CSS modules let you write styles in CSS files but consume them as JavaScript objects for additional processing and safety. CSS Modules are very popular because they automatically make class and animation names unique, so you do not have to worry

about selector name collisions. CSS Modules are named with [name].**modules.css**, where the name is the name of the file.

- **Styled components:** Styled components is a library for React that allows you to use component-level styles in your application that are written with a mixture of JavaScript and CSS.

Styled components allow React developers to write plain CSS in React components without having to worry about the clashing of class names. The following code is an example of how to use styled-components:

```
1. import React from "react";
2. import styled from "styled-components";
3.
4. const Container = styled.div`
5.   background-color:red;
6.   color:#fff;
7. `;
8. const App = () => {
9.   return <Container>This is just a test</ Container>;
10. };
11.
12. export default App;
```

[Preprocessor \(SASS/LESS\)](#)

SASS and LESS are very good CSS Preprocessors. A CSS preprocessor is a program that lets you generate CSS from the preprocessor's own unique syntax. SASS and LESS are very good with react applications because of their very rich features, such as Mixins and so on, for SASS.

[File/folder structure](#)

Most of the time, we are unsure of how to set up our folder and file structure to best serve the application we are developing. Yes, react is opinionated, but one thing we should keep in mind is that file/folder

structure is important because it gives a sense of direction on where to put/find some things for your application.

This is up to you but always have in mind the other person who will take over the code when doing this file structure.

Naming conventions

This is very important because it gives meaning to the application. It makes the application to be easily understood even before looking at the logic. When the naming of variables, functions, methods, and file/folder are done properly, tracing problems in the application is simple since one is aware of what each variable or function is doing. I usually suggest keeping this in mind when naming your application.

Rendering with useMemo and useCallback

The **useMemo** and **useCallback** are very effective performance hooks methods when using react hooks. The **useMemo** is used to cache a computed calculation and returns its value. This calculation is not done every time on render. It is a hook that takes in two parameters, namely, the fat arrow function and the dependency array. The dependency array is optional, but if a parameter is passed, it is only when the parameter changes that the function will run again, and the resulting value is returned. This is the same with **useCallback**, but the main difference is that **useMemo** returns the memorized value while **useCallback** returns the memorized function. Let us look at the following example to help us understand the work of these two hooks and why they are so important in building scalable and fast applications.

We would start with the **useMemo**. Let us look at the following code:

```
1. import React, {useMemo} from "react";  
2.  
3. const App = (props) => {  
4.  
5.   const value=useMemo(()=>{  
6.     return 123;
```

```

7. },[])
8.
9.   return (
10.     <section>
11.       {value}
12.     </section>
13.   );
14. };
15.
16. export default App;

```

Line number 5 will always return the value **123** no matter how many times the component re-renders. This is because the dependency array has not changed and so react does not need to recompute the value again.

Let us look at what **useCallback** will do:

```

1. import React,{useCallback} from 'react';
2.
3. const App=()=>{
4.   const [count, setCount] = useState(0)
5.
6.   const increaseCount=useCallback(()=>{
7.     setCount(count+1)
8.   },[])
9.
10. }
11.
12. return (
13.   <button onClick={increaseCount}>Set Increase</button>
14. )
15.
16. export default App;

```

[Use Prettier](#)

Prettier is an open source that enforces code consistency. It makes your code look neat and readable. It enforces the rules that have been set by you.

Use Linter and its rules

Linter helps you organize your code, especially your import hierarchy. Linter reminds you of some JavaScript errors. Having Linter and its rules in your application can make your life easy.

Use try...catch

No matter how perfect our application is, there are bound to be errors. Errors can be from an API or even from a user input, which we did not see coming or even thought of during testing. This is why it is important to always wrap your logic or API calls with try and catch so that it can catch unexpected errors.

Log errors

In as much as we catch errors, we also need to log them. Logging these errors can tell us what exactly happened during the lifecycle of an operation within the application. We can log this error to a file or create a service where these errors are pushed to an API or even to a database. This is essential and is typically one of the first points of call when things go wrong in an application on production. It can save the day.

React error boundary

There are times in our application when we have some error in the UI, and the page will just show a white screen; this can be solved by using React error boundary. React error boundary helps to catch application errors and show a friendly message or screen to the user. The friendly error message or screen is usually a fallback component. Let us refer to the following example:

1. `class ErrorBoundary extends React.Component {`
2. `constructor(props) {`
3. `super(props);`

```
4.    this.state = { hasError: false };
5.  }
6.
7.  static getDerivedStateFromError(error) {
8.    return { hasError: true };
9.  }
10.
11. componentDidCatch(error, errorInfo) {
12.    errorService.log({ error, errorInfo });
13.  }
14.
15. render() {
16.    if (this.state.hasError) {
17.      return <p>Oops, something went wrong.</p>;
18.    }
19.    return this.props.children;
20.  }
```

This will return to Line 17 if Line 16 condition is true.

This will not catch asynchronous errors that occur from API calls because they are not inside the error boundary. To catch such asynchronous errors, we should add the try and catch block.

[Test your code](#)

Most developers do not like writing tests for their applications (me included), but as time went by, I started indulging in writing unit tests and integration tests. This is very important because as the application grows, it will become very important to implement so as to avoid issues. Tests solidify the code and give you the assurance that your function and logic are working as expected.

This is also important because you can implement the edge cases that come to your mind and test it without many hitches as against you running the

application and passing parameters from your screen to test. This saves a lot of time and gives some level of assurance that your code is production ready and also reduces the likely bugs from the testers.

Component name must be in Pascal case

React component naming convention is usually in Pascal case. This starts with a capital letter, and failing to do so will result in the component not rendering. As you can see from all our examples so far, we named our component **App**. Notice that the naming convention started with a capital letter.

Do not do this

```
1. import React from 'react';
2.
3. const app={()=>{
4.
5.
6. return (
7.   <p>This is just a test</p>
8. )
9.
10. export default app;
```

Rather do this

```
1. import React from 'react';
2.
3. const App={()=>{
4.
5.
6. return (
7.   <p>This is just a test</p>
```

8.)
- 9.
10. export default App;

Extract reusable logic

This is very important in react applications because it solves unnecessary repetition. The logic that will be used across the application can be extracted to a component and used at any given point in time. This is another form of **do not repeat yourself (DRY)** technique. This will save you from many lines of code and also make you isolate bugs because this is only one source of truth, which is your reusable component/logic.

Benefits of using JavaScript in React application

React is a library that uses JavaScript. We are going to look at the advantages/benefits of using JavaScript in our react application.

- **Easy to learn/use:** JavaScript is simple to learn and implement. There are loads of tutorials out there that teach JavaScript from elementary to advanced. This knowledge is the same one would use in react applications. Even if your JavaScript knowledge is from vanilla JavaScript, it would still be useful. However, you would need to start learning some ECMA SCRIPT 6 (**es6**) upwards to make use of react very well. Most tutorials on react out there are mainly on JavaScript as against TypeScript, so this is a big advantage for those who like to code in JavaScript.
- **Speed:** Another benefit of developing react applications in plain JavaScript is because of its speed in development and execution. Since it is a client-side language (for most react applications), its development is straightforward and easy and has no compilation time.
- **Performance:** React JS was designed with the provision of high performance in mind. The core of the framework offers a virtual DOM program and server-side rendering, which makes complex apps run extremely fast that has JavaScript as its backbone.

Benefits of using TypeScript in React application

React is a library that uses JavaScript. But react also uses TypeScript, which transpiles to JavaScript using babel so that our browser can understand it. There is a slightly different syntax of TypeScript and JavaScript, but we would look at the benefits of using Typescript in our react application.

- **Types safety:** TypeScript offers type safety checks when variables are declared, or functions are defined. This helps avoid wrong variable assignments to already defined variable types. This will help reduce errors and bugs to their barest minimum.
- **Interfaces:** Typescript allows you to define complex type definitions for your application using Interfaces. This helps to check the types that are passed and so ensures that the number of bugs are reduced.

Conclusion

What we have learnt so far in this chapter are the bad practices most of us use in React. We also learnt the best practices on how to create your React application and the benefits of JavaScript and TypeScript in React. We all know React is opinionated, but this chapter has helped us to know some of the best ways to go about your application.

In the upcoming chapter, we will look at how to write components and the benefits of controlled vs uncontrolled components. We will look at class components and functional components and some of their lifecycle methods.

CHAPTER 2

Writing Components

Introduction

This chapter will cover how to write components. React is a component-based library whereby different reusable components are used to build an application. Basically, this chapter will outline the different ways of writing components, which include functional and class-based components.

Although we will focus more on functional-based components because it is the most recent in React Docs, there are still applications built with class components. We will be using JavaScript and TypeScript for our examples in this chapter.

Structure

The following topics are to be covered:

- Naming conventions
- Functional components
- Class components
- Best ways to structure components
- Data sharing between components
- Controlled versus uncontrolled components

Objectives

In this chapter, we will learn the different naming conventions in programming. We will also look at the naming conventions when naming variables, methods, and functions. We will also look at how we name Components in React.

We will look at functional components and class components and ways to share data between components. We would conclude by looking at

controlled versus uncontrolled components.

Naming conventions

In programming, there are two mostly used naming conventions, namely:

- Camel case
- Pascal case

Camel case

Camel case naming convention is starting the name of a variable, function, or method with a small letter or the first part with small letters, and the first letter of the other word starts with a capital letter. Kindly look at the following example:

```
1. const userName='Anthony'
```

From the preceding example, you can see the variable username starting with small letters for the first word and the first of the second-word Name starting with a capital letter. This is how variables are named in react if they have two words in the variable. If the variable name has just one word, it will all be small letters. Kindly see the following example:

```
1. const user='Anthony'
```

Pascal case

Pascal case naming convention is giving the name of a variable, function, or method with a capital letter. Kindly look at the following example:

```
1. const UserName='Anthony'
```

This naming convention is what react uses to name components. All React components must follow the pascal naming convention. This is the only way the component will render. Let us look at an example of the following component:

```
1. import React from 'react';  
2.
```

```
3. const App={()=>{
4.
5.   return (
6.     <p> Hello world</p>
7.   )
8.
9. }
10.
11. export default App;
```

The preceding example shows us an **App** component. You can see in line 3 that we named the arrow function component with a Pascal naming convention. This is the main rule of react component naming. This is the first rule to making it render.

Functional component

Functional components are plain JavaScript functions that take in props and return a react component. This component can be either a fat arrow function or a normal function. They can be written like the following:

```
1. import React from 'react'
2.
3. const App={()=>{
4.
5.   return (
6.     <p>This is just a functional component </p>
7.
8.   );
9. };
10. export default App
```

OR

```
1. import React from 'react'
```

```
2.  
3. function App(){  
4.  
5.   return (  
6.     <p>This is just a functional component </p>  
7.  
8.   );  
9. };  
10.  
11. export default App
```

The functional component can accept **props** (optional) and render the **props** value in the react component body. Let us take a look at an example of this:

```
1. import React from 'react'  
2.  
3. function App(props){  
4.   const {id}=props;  
5.   return (  
6.     <p>This is just a functional component with {id} </p>  
7.  
8.   );  
9. };  
10.  
11. export default App
```

The preceding example shows that the App functional component receiving **props** object that has an **id** property, which we destructured on line 4 and displayed in the render on line 6.

The functional components have some very common **hook** methods. React hooks have enabled most developers to build scalable react applications. Let us look at some of the following lifecycle methods:

useState

- useRef
- useEffect

useState

The **useState** method is a very common react hook that is used in most functional components. The **useState** method is used to store state values (objects, strings, Boolean, and so on) in a functional component. These values are mutated during the lifecycle of the component. The **useState** takes an initial value which can be empty in the case of a string. The value can be updated during the lifecycle of that component. Let us look at an example of the **useState**.

```

1. import React,{useState} from "react";
2.
3.
4. const App=()=>{
5.   const [user,setUser]=useState('');
6.
7.   return (
8.
9.     <input    type='text'    onChange={(e)=>setUser
      (e.target.value)} />
10.  )
11. }

```

The preceding example shows on line one that we imported the **useState** hook from react library. We then defined the user state on line 5 and added the **setUser**, which will update the value that is set to the user. The user has an initial value of the empty string.

In line 9, we call the **onChange** handler event and pass the anonymous function to it to update the state value.

Let us look at the same example with TypeScript.

```

1. import React,{useState} from "react";

```



```

2.
3.
4. const App=()=>{
5.   const [user, setUser]=useState<string>>('');
6.
7.   return (
8.
9.     <input    type='text'    onChange={e=>setUser
      (e.target.value)} />
10.  )
11.
12. }

```

The difference between this and the previous example is that TypeScript requires a type which is a string, and we provided it in line 5.

[useRef](#)

The **useRef** method is also a very common react hook that is used in most functional components. The **useRef** method is used to point to an element in the **Document Object Model (DOM)**. This can be used to create uncontrolled elements. Let us see an example of the **useRef** in action where we want programmatically to browse a file:

```

1. import React,{useRef} from "react";
2.
3.
4. const App=()=>{
5.   const refInput=useRef(null);
6.
7.   const browseFile=()=>{
8.     if(refInput.current)
9.     {
10.

```

```

11.     refInput.current.click()
12.   }
13. }
14. return (
15.   <>
16.     <button onClick={browseFile}>Click to browse
        file</button>
17.     <input type='file' ref={refInput} style={{ display:
        'none' }} />
18.   </>
19. )
20.
21. }
22.
23. export default App;

```

In the preceding example, you can see we added the **useRef** hook from React on line one. We then used **useRef** to browse for a file with the click of the button in line 16. There is another use case of **useRef**, which we would look at. This example is to get the value from the input file:

```

1. import React,{useRef} from "react";
2.
3.
4. const App=()=>{
5.   const refInput=useRef(null);
6.
7.
8.   return (
9.     <>
10.      <input type='text' ref={refInput} style={{ display:
        'none' }} />
11.    </>
12.  )
13.
14. }

```

```
15.  
16. export default App;
```

We can get the value of the input file by using the **ref** also. Here, we can use the **refInput.current.value** to get what the user typed in on line 10.

useEffect

The **useEffect** method is also a very common react hook that is used in most functional components. The **useEffect** method is an asynchronous hook that lets us do asynchronous tasks on the component. These asynchronous tasks include calling of API and setting the data to a **useState**. The **useEffect** takes in two parameters, namely:

- Fat arrow function
- An optional dependency arrays

The fat arrow function takes in an optional **return** statement, which is used to clean up the component. This is necessary to avoid component leakage. Let us see an example of the **useEffect** hook. We would like to make an API call to <https://jsonplaceholder.typicode.com/todos> and render the list in the react component.

```
1. import React,{useEffect,useState} from 'react';  
2. const App=()=>{  
3.   const [data,setData]=useState([]);  
4.   useEffect(()=>{  
5.     fetch('https://jsonplaceholder.typicode.com/todos')  
6.       .then(response=>response.json())  
7.       .then(data=>setData(data))  
8.       .catch(err=>console.log(err.message))  
9.   }  
10.  , [])  
11.  
12.  return(  
13.    data.length>0 &&data.map(item=>{  
14.      return <ul key={item.id}>  
15.        <li>{item.title}</li>  
16.      </ul>  
17.    })
```

```
18.   )
19.
20. }
21. export default App;
```

The preceding component just illustrates how we use the **useEffect** hook for asynchronous operations. We used the `fetch` method to call the API in line 5 and used promise chaining to get the data and store it in our data **useState** hook. We have an empty array as the second parameter of the **useEffect**. This indicates that it should run just once. Should we need conditional fetch from the API, we need to populate the dependency array with some defined parameters.

Class component

Class components are components that extend **React.component** subclass. This class component accepts **props** and renders them. It also starts with a constructor, which is then called by the superclass. Let us see an example of a class component:

```
1. import React from "react";
2. import "./styles.css";
3.
4. class App extends React.Component{
5.   constructor(props){
6.     super(props);
7.   }
8.   render(){
9.
10.    return <p>This is a test</p>
11.
12.   }
13.
14. }
15. export default App
```

Class components have some common lifecycle methods. Some of the common lifecycle methods includes:

ComponentDidMount

ComponentDidUpdate

ComponentDidMount

The **ComponentDidMount** method is a very important lifecycle method in class component. This lifecycle method is called during the mounting phase of a react component lifecycle. This method helps us modify the content of the react component before displaying data to the user. Let us see this lifecycle method in action with an example.

```
1. import React from "react";
2. import "../styles.css";
3.
4. class App extends React.Component{
5.   constructor(props){
6.     super(props);
7.     this.state = {
8.       message: "This is a test message"
9.     };
10.
11.   }
12.   componentDidMount() {
13.     this.setState({
14.       message: "This is a new message"
15.     });
16.   };
17. }
18. render(){
19.
20.   return <p>{this.state.message}</p>
21.
22. }
23.
24. }
25. export default App
```

ComponentDidUpdate

The **ComponentDidUpdate** method is a very important lifecycle method in class component. This lifecycle method is called after the component has been updated. Let us see this lifecycle method in action with an example:

```
1. import React from "react";
2. import "./styles.css";
3.
4. class App extends React.Component{
5.   constructor(props){
6.     super(props);
7.     this.state = {
8.       message: "This is a test message"
9.     };
10.
11.   }
12.   componentDidUpdate(prevProps, prevState) {
13.     if (prevState.message !== this.state.message) {
14.       console.log('Message state has changed.')
15.     }
16.
17.   }
18.   render(){
19.
20.     return <p>{this.state.message}</p>
21.
22.   }
23.
24. }
25. export default App
```

Best way to structure a react component

React component is the way to build a small to robust application. These React components need to be structured in a good way so that they can be testable, scalable, and easy to identify bugs. Though react is opinionated, there are better ways to achieve a good react component structure.

Let us look at some of the best ways to keep a good react component structure.

- **Avoid large components:** The large components are a pain to read, understand, and debug most of the time. Yes, you can have a large component, and the application would work just fine, but when issues

arise, how would you be able to debug? Therefore, you should break down your large components into smaller components so it can be readable, testable, and identify bugs easily.

- **Give good naming of components and variables:** It is very important to write variable names, method names or components names that are reasonable. Avoid ambiguous naming.
- **Keep your folder structure precise and understandable:** File and folder structure is also very important in achieving a good component structure. Give folder structure (however you want) to your project so that it is easy to understand what files are to be put in a particular folder.
- **Move reusable logic into a separate class or function:** Generally, in programming, always keep in mind the DRY concept. Whatever reusable logic you feel the app or component will use, move it to a function or method and call it wherever you want to use it in the application.
- **Try to write tests:** Tests are very important when writing an application. Lots of developers hate writing tests, but they have their own advantages and trust me, they are huge. This gives you some assurance that your components are working as expected and would reduce the number of bugs in your application when written well.

Data sharing between components

In react, we must share data between components. There are different ways to do this depending on how complex your state is changing or how big your application is. Let us look at some ways of doing this:

- Props
- ContextApi
- Redux
- useReducer

Props

Props are essential in React. Props is one of the ways to pass data from one component to another in React. Props are objects that are passed from the parent component down to the children's components. Props is a short form of properties.

We will look at a simple way of passing **props** from the parent component to the child component:

```
1. import react from 'React';
2.
3.
4. const data={
5.   id:1,
6.   name:'Chidimma Jane Udensi',
7.   state:'Imo State',
8.   Village:'Akokwa',
9.
10. }
11. const App={()=>{
12.
13.   const secondComponent=(props)=>{
14.     const {id,name,state,village}=props;
15.
16.     return (
17.       <>
18.         <p>{id}</p>
19.         <p>{name}</p>
20.         <p>{state}</p>
21.         <p>{village}</p>
22.       </>
23.     )
24.   }
```



```
25.  
26.   return (  
27.     <secondComponent props={data}/>  
28.   )  
29.  
30. }  
31.  
32. export default App;
```

The preceding example shows how we passed props from the **App** component to the second component. The second component needs a prop, and we passed the object data to it and destructured it on line 14.

Context API

Context API is also a way of passing data from one component to another. The main difference between context API and Props is that context API is not passed down from parent to child on every component. The context API has two main methods:

- Provider
- Consumer

Provider

This is one of the essential parts of **contextApi**. The Provider accepts a value that is to be passed down to the child components.

Consumer

This is one of the essential parts of **contextApi**. The Consumer allows the calling component to subscribe to the context changes.

Redux

Redux is an open-source JavaScript library that keeps a global state that makes applications behave consistently. The redux library consists of the following three parts:

- Store
- Reducer
- Actions

Store

The store is one of the parts of the Redux open-source JavaScript library. This is the part where the global state is stored. This part of redux is immutable, which simply means, not possible to change.

Note: Immer in the redux toolkit enables you to write mutable code.

Reducer

The reducer is one of the parts of the Redux open-source JavaScript library. The reducer is a pure function that takes in two arguments (initial state and action) and returns a new state.

Actions

The actions are one of the parts of Redux open-source JavaScript library. The action is a JavaScript object that tells the reducer what the user wants to do in the store. The actions are instructions from the user to either mutate (using immer) a state in the store or create a copy of the state in the store.

useReducer

The **useReducer** method is also a way to share data between components. The **useReducer** method is used when you have complex state changing. The **useReducer** method has an initial state and has an action and returns the current state and a **dispatch** method. Let us look at an example for the same:

```
1. import { useReducer } from "react";
2.
3. const initialTasks = [
4.   {
5.     id: 1,
```

```
6.     title: "Drink Coffee",
7.     completed: false,
8.   },
9.   {
10.    id: 2,
11.    title: "Write Code",
12.    completed: false,
13.  },
14.];
15.
16.const reducer = (state, action) => {
17.  switch (action.type) {
18.    case "COMPLETED":
19.      return state.map((todo) => {
20.        if (todo.id === action.id) {
21.          return { ...todo, completed: !todo.completed };
22.        } else {
23.          return todo;
24.        }
25.      });
26.    default:
27.      return state;
28.  }
29.};
30.
31.function TodoTask() {
32.  const [todos, dispatch] = useReducer(reducer,
    initialTodos);
33.
34.  const handleComplete = (todo) => {
```

```

35.     dispatch({ type: "COMPLETED", id: todo.id });
36.   };
37.
38.   return (
39.     <>
40.       {todos.map((todo) => (
41.         <div key={todo.id}>
42.           <label>
43.             <input
44.               type="checkbox"
45.               checked={todo.completed}
46.               onChange={() => handleComplete(todo)}
47.             />
48.             {todo.title}
49.           </label>
50.         </div>
51.       ) )}
52.     </>
53.   );
54. }
55.
56. export default TodoTask

```

Controlled versus uncontrolled components

Controlled components are basically data that is handled by react components, whereas uncontrolled components are data handled by the DOM. The controlled component is basically handled by events or input from the user. Let us look at an example of both:

```

1. import React, {useState} from 'react';
2.

```

```

3.
4. const App=()=>{
5.   const [user, setUser]=useState('');
6.
7.   return (
8.     <input type='text' onChange={(e)=>
       setUser(e.target.value)}/>
9.   )
10. }
11.
12. export default App;

```

The preceding code snippet is an example of controlled components. You can see on line 8 that as the user inputs values in the textbox, we are controlling the input by setting the values in the user state.

```

1. import React, {useRef} from 'react';
2.
3.
4. const App=()=>{
5.   const userInputRef=useRef();
6.
7.   return (
8.     <input type='text' ref={userInputRef} />
9.   )
10. }
11.
12. export default App;

```

The preceding code snippet is an example of uncontrolled components. You can see that we made a reference to the Dom element on line 8. We can get the value of the user by using the **ref** attribute.

Conclusion

In this chapter, we have been able to look at how we should name components and variables. We also looked at functional components, class components and some of their commonly used lifecycle methods.

We further looked at the best ways of structuring components, and ways data is being shared between components. We ended with controlled versus uncontrolled components with some examples.

In the upcoming chapter, we will be looking at clean and maintainable codes and their benefits.

CHAPTER 3

Clean and Maintainable Code

Introduction

This chapter will cover how to write clean and maintainable code. This will cover the principles and benefits of writing clean code. We will be using JavaScript and TypeScript for this chapter.

Structure

The following topics are to be covered:

- Writing a clean code
- Achieving clean and maintainable codes
- Writing maintainable code
- Using TypeScript to build applications

Objectives

In this chapter, we will learn to write clean and maintainable code. We will also look at the patterns used to achieve clean and maintainable code. We will conclude by learning how to use TypeScript to build react applications.

Writing a clean code

Clean code is very necessary for programming. Clean code is needed because of the following:

- Business continuity
- Easy to read and understand

Business continuity

Business continuity is critical in every organization. It is critical because there must be continuity when a developer leaves a company or business, and the software must continue to run for the organization's benefit. The developer must write code that is simple to maintain so that businesses can continue to operate with or without the developer.

Easy to read and understand

The code should be simple to read and understand. Every developer who writes code should consider who will read, debug, add features, and support the application. No one should contact you as a developer after you have left the company to inquire about what you wrote or to comprehend the logic of what you were doing.

Achieving clean and maintainable code

Now that we have examined the reasons why we need to write clean codes, we should consider how we can do so. Let us take a look at some of the ways this clean code can be implemented and discuss them briefly:

- **Keep it simple stupid:** Keeping it simple is one way to achieve clean coding. KISS is just telling you, as a developer, to keep the code simple. It is just a reminder that you should not over-engineer things. Do not complicate things all in the name of standards, thereby making it too complicated.
- **Do not repeat yourself:** This is one way of achieving clean code. Repeating oneself is not a good practice in general. Imagine repeating the same code logic in more than one place in your application; there will be multiple redundancies of such codes. Instead of repeating yourself, please move reusable logic/code to a function or method and call it wherever or whenever you want.
- **Try to explain yourself in code:** Whatever code you write—be it a method, function, or even a variable declaration, you should write it in a way that you are explaining to the next person that is going to read, debug, and support it. This is very important because your code should be easy to read and understand without much debugging.

- **Avoid too many arguments in a function:** Most functions/methods will require an argument(s). These arguments should be kept at most three parameters. If they are more than 3, please make the method receive an object and then destructure the object for your use in the method. This makes it clean and simple to read because the object is extendable (more arguments can go inside of it).

Let us look at an example for the same:

```
1. function                                concatUser(firstname,middlename,
   lastname,country,mobile){
2.
3.                                     return
   firstname+'.'+middlename+'.'+lastname+'.'+country+'.'+m
   obile;
4.
5. }
```

Do not do this!!!

The preceding code is a simple function that concatenates a user's details. You can see that the arguments it receives in line 3 are about five arguments and returns the concatenated string in line 5. This is not a good practice because the arguments are more than 3. Let us look at the same code in the following typescript:

```
1.
2.
3. function                                concatUser(firstname:string,
   middlename:string,lastname:string,country:string,mobile
   :string):string{
4.
5.                                     return
   firstname+'.'+middlename+'.'+lastname+'.'+country+'.'+m
   obile;
6.
7. }
```

This is the same example as the first one. The only difference is that the arguments are strongly typed, and the **concatUser** function must return a string because we defined that it should return a string.

To make the **concatUser** function look clean, we should make the parameters an object and then destructure the object and get our parameters out. Let us look at the following example to achieve this:

```
1. const obj={
2.   firstname:'Anthony',
3.   middlename:'Onyekachukwu',
4.   lastname:'Okonta',
5.   country:'Nigeria',
6.   mobile:'+2348039436123'
7. }
8.
9. const user=concatUser(obj);
10.
11. console.log(user)
12.
13. function concatUser(userObj){
14.                                     const
      {firstname,middlename,lastname,country,mobile}=userObj;
15.                                     return
      firstname+', '+middlename+', '+lastname+', '+country+', '+m
      obile;
16.
17. }
18.
19.
```

The preceding example shows how we can avoid multiple arguments in the **concatUser** function. The preceding code in line 20 will print **Anthony, Onyekachukwu, Okonta, Nigeria, +2348039436123**.

This looks cleaner and simpler. Let us look at how it would be using the same code in typescript:

```
1. interface IUser{
2.   firstname:string;
3.   middlename:string;
4.   lastname:string;
5.   country:string;
6.   mobile:string
7. }
8.
9.
10. const obj:IUser={
11.   firstname:'Anthony',
12.   middlename:'Onyekachukwu',
13.   lastname:'Okonta',
14.   country:'Nigeria',
15.   mobile:'+2348039436123'
16. }
17.
18. const user=concatUser(obj);
19.
20. console.log(user)
21.
22. function concatUser(userObj:IUser){
23.                                     const
      {firstname,middlename,lastname,country,mobile}=userObj;
24.                                     return
      firstname+', '+middlename+', '+lastname+', '+country+', '+m
      obile;
25.
26. }
```

The preceding code in line 20 will print **Anthony, Onyekachukwu, Okonta, Nigeria, +2348039436123**. We just made sure we made an interface for the `userObject` and have its properties to be strongly typed, thereby prompting the user to make sure that the parameters entered conform with what has been defined.

- **Code should be loosely coupled:** Having a loosely coupled code is very important because it will make all parts of the application to be independent but work together. The advantage of this is that anyone can come in (even a new joiner) and add a new code or functionality to the existing application without breaking the current working code.
- **Remove commented or unused code:** When we develop applications, we tend to comment on code that we wrote by error or that we will later use or codes that are causing bugs in our application. Doing this is not a good thing because it makes the application code unnecessarily long. Always ensure you remove commented or unused codes (functions, methods, or variables) from your code before committing to production.
- **Use self-descriptive names:** When naming a variable or method, please try to give it a meaningful name. Give self-descriptive names that are simple to understand at first glance. Consider the following examples of ambiguous naming and self-descriptive naming:

1. `const a='Anthony';`
2. `const b='Okonta';`

The preceding code is simply assigning parameters to variables after declaration. Anthony is assigned to variable **a**, whereas **Okonta** is assigned to variable **b**. This at first glance, does not make any sense because what is **a** and what is **b**? Maybe the user is trying to assign these names as **firstname** and **lastname** or something else. For us to correct this, let us use something more descriptive now:

1. `const firstName='Anthony';`
2. `const lastName='Okonta';`

The preceding example makes a whole lot of sense because we can see what the developer is trying to do here. He is simply declaring

descriptive values **firstName** and **lastName** and assigning the values **Anthony** and **Okonta** to them, respectively.

- **All functions must do one thing:** We all write functions in our applications to help us out. These functions can do a variety of things depending on our goals, but keep in mind that each function should only do one thing. Assume you want a function that converts all words to capital letters and then sends the result to an API endpoint. Most of the time, we return the capitalized result and pass the parameters to the end point within the same function. Let us take a look at the following example to see what we are talking about.

```
1. capitalizeWords('anthony')
2.
3. function capitalizeWords(word){
4.   const convertedWord=word.toUpperCase();
5.
6.   sendConvertedWord(convertedWord)
7. }
8.
9. async function sendConvertedWord(convertedWord){
10.  return await fetch('api.com',{
11.    method:'POST',
12.    headers:{
13.      accept:'application/json'
14.    },
15.    body:(JSON.stringify({
16.      word:convertedWord
17.    })))
18.  })
19. }
```

The preceding code shows two functions (**capitalizeWords** and **sendConvertedWord**), which does two different things. The preceding **capitalizeWords** accepts an argument called word and capitalizes the

word and then calls the **sendConvertedWord** API and passes the parameter to it on line 6. This will work but will have issues because the function is not doing just one thing, which is to convert the passed in word to capital letters. Let us look at the following correct one.

```
1. const capitalizedWord=capitalizeWords('anthony');
2. if(capitalizedWord)
3. {
4.   sendConvertedWord(capitalizedWord)
5. }
6.
7. function capitalizeWords(word){
8.   if(typeof word === 'string'){
9.     return word.toUpperCase();
10.  }
11.  else{
12.    return null;
13.  }
14.
15. }
16.
17. async function saveWord(convertedWord){
18.   return await fetch('api.com',{
19.     method: 'POST',
20.     headers:{
21.       accept: 'application/json'
22.     },
23.     body: (JSON.stringify({
24.       word: convertedWord
25.     })))
26.   })
27. }
```

You can see from the preceding code that we successfully made the **capitalizeWords** function do just one thing, which is capitalize words and return them to the caller on line 1. We then send the returned params to the **sendConvertedWord** function in line 7.

Please note that this is a dummy API used in this example.

- **Keep functions short and precise:** Keeping functions short and precise is one way to achieve clean code. When your functions become so large, it becomes difficult to read. Try to keep them very short, at most 50–60 lines.
- **Give descriptive branch names when branching from master:** The majority of our code will be pushed to an online repository (GitHub, Bitbucket, and so on) because it will undoubtedly be the source of truth, allowing us to have **Continuous Integration (CI)** and **Continuous Delivery (CD)**. We would create a branch from the master to develop a new feature or fix a bug. Assume we are asked to fix an infinite loop bug in production. We should execute the following command:

```
git checkout -b fix/infinite_loop_fixes.
```

The preceding command will create a branch called **fix/infinite_loop_fixes**. This will give the PR reviewer an idea of what is going on. Or you can name it like this:

```
git checkout -b fix/ticket_number_infinite_loop_fixes.
```

The **ticket_number** is the number that the ticket of the fixes was logged.

- **Give descriptive titles when committing your code to a repository:** When committing codes for **Pull Requests (PR)**, it is very important to give good titles/descriptions as the message. This is necessary because it gives the reviewer and others a sense of what you did inside the code. Let us look at two examples to buttress this point. We would look at one commit that has to do with refactoring the **capitalizeWords** function:

```
git commit -m 'refactoring'
```

```
git push
```

```
git commit -m 'refactoring the capitalizewords function'
```

```
git push
```

The first commit code just says refactoring. The refactoring is vague and does not provide a detailed account of what was done inside the code.

The second commit code refactoring the **capitalizewords** function shows us a detailed message pointing to the fact that it was just the **capitalizewords** function that was changed inside the code. This gives the reviewer a sense of what happened and what he is looking out for inside the code as he reviews it.

- **Write unit tests:** Most developers hate writing unit tests. It can be annoying sometimes but it is very important to write tests for your code. Testing your code will give you an idea of what kind of errors and scenarios to expect and how to avoid them. However, unit tests are mostly functional tests and do not go as far as testing the user interface. There are tools you can use to test your React application. Examples of such tools includes React testing library, Jest, and so on.

Writing maintainable code

Writing code is important but writing maintainable code is very important to any organization or company. As a developer, it is not just to write the code that should work but write code that can be easily fixed from bugs, if any, maintained and supported. We would look at a few practices which help us achieve a maintainable code base.

- Design patterns
- Writing testable codes
- Checking for errors
- Log errors for easy bug tracking and fixes

Let us now have a brief discussion about each of the preceding ways mentioned to achieve a maintainable code base.

Design patterns

Design patterns are solutions to software design problems. Design patterns give a defined way/pattern of building applications. Having design patterns is very important if you want to have a code base that can easily be

supported and maintained. We have so many design patterns, but we would just list some of them here:

- Repository pattern
- Singleton pattern
- Domain-driven design pattern

These design patterns have their unique ways of solving software design problems. They are unique in their own ways. This will help in making the code maintainable; there is a given pattern everyone must follow.

Writing testable codes

It is important to understand that writing code is beyond its working; it should also be easy to test. When writing a code, please make sure you keep in mind that you should write codes that are testable. This is important because when the codes are testable, it is easy to find and resolve issues.

Checking for errors

In every application you write, there are bound to be errors in it. Errors are inevitable, and that is why you, as a developer, need to check for them. It is important to note that wrapping your codes around with try and catch can help check for these errors and display friendly messages to the user and log these errors to a file or save them into a database.

Log errors for easy bug tracking and fixes

Logging errors is very important when building an application because it will give an idea of what happened during the runtime of the application. These logs are then taken and analyzed as the first point of contact in troubleshooting.

Using TypeScript to build applications

Typescript is a superset of JavaScript. TypeScript helps to build scalable applications because it is strongly typed. TypeScript compiles to JavaScript at runtime with the help of Babel. We have used some typescript examples

in this book so far, but we would look at the concepts and things to note when building applications with typescript.

Conclusion

We have looked at the different ways to achieve clean and maintainable code. In the upcoming chapter, we will look at React hooks and their benefits. We will also look at how we can use custom hooks to build our applications.

CHAPTER 4

React Hooks

Introduction

This chapter will cover what React hooks are, and we will look at the most common hooks used in React. We will also look at how to write custom hooks and their benefits when building applications. We will see how hooks and custom hooks can be combined in a single application.

Structure

The following topics are to be covered:

- React hooks
- Custom hooks

Objectives

In this chapter, we will learn how to write custom hooks and use inbuilt hooks in React. We would look at an example project to learn more about hooks.

React hooks

React hooks are lifecycle methods used in functional components. React hooks were introduced in React 16.8. The lifecycle methods, which were in-class components, were collapsed into React hooks. React hooks will NOT work inside the class component. Some of these inbuilt React hooks include the following:

- useState
- useReducer
- useEffect

- `useLayoutEffect`
- `useMemo`
- `useCallback`
- `useRef`
- `useContext`

There are some rules to follow when using React hooks. We will look at a few of them:

- Hooks can only be called from a functional component.
- Hooks must be called from the top level, not from within loops, conditional statements, and so on.
- You can create your own hooks, but you must follow the rules one and two.

`useState`

The **`useState`** method is one of the commonly used React hooks. This hook is classified under the controlled component in React. The **`useState`** method sets an initial value, which is updated as the user performs an operation. Let us use an example to explain this:

```

1. import React,{useState} from 'react'
2.
3.
4. const MyApp={()=>{
5.   const [counter,setCounter]=useState(0);
6.
7.   return (
8.     <>
9.       <p>Current count is {counter}</p>
10.      <button onClick={()=>setCounter(counter+1)}> increase
        counter </button>
11.      <button onClick={()=>setCounter(counter-1)}> decrease
        counter</button>

```

```
12.     </>
13.   )
14.
15. };
16.
17.
18. export default MyApp;
```

The preceding example is a simple counter application. The application has two buttons that increase and decrease the counter value. We imported the **useState** hook on line one from React library. We then declared the counter state and set its initial value to 0 on line 5. So on the click of either of the buttons, we call the **setCounter** and pass the current counter to it so we can increase or decrease the current counter value by **1**. The value of the counter will be displayed on line 9.

[useReducer](#)

The **useReducer** method is one of the commonly used React hooks. This hook is used when you have a complex changing state in your application. This is like **useState**, but you dispatch actions to update the state. Let us use an example to explain this:

```
1. import React,{useReducer,} from 'react'
2.
3.
4. const MyApp={()=>{
5.   const initialState={counter:0}
6.   const appReducer=(state, action)=>{
7.     switch(action.type){
8.       case 'increase':
9.         return {counter:state.counter ++};
10.      case 'decrease':
11.        return {counter:state.counter --};
```

```

12.     default:
13.         return {counter:state.counter};
14.     }
15. }
16. const [state, dispatch] = useReducer(appReducer,
    initialState);
17. return (
18.     <>
19.         <p>Current count is {state.counter}</p>
20.         <button onClick={()=>dispatch({type:
    'increase'})}>increase counter </button>
21.         <button onClick={()=>dispatch({type:'decrease'
    })}>decrease counter</button>
22.     </>
23. )
24. };
25.
26.
27. export default MyApp;

```

The preceding example is also a simple counter application. The application has two buttons that increase and decrease the **counter** value. We dispatch actions to the reducer to update the **counter** state. The reducer takes in two parameters (**state** and **action**). There is usually an initial state which, in this case, is a **counter** object that has its value set to 0. The preceding code will look like this in the browser:



Figure 4.1: Counter app

Clicking on the increased counter will increase the **counter** value by **+1**.

Current count is 1

increase counter decrease counter

Figure 4.2: Counter app

Also, clicking on the decrease counter will reduce the **counter** value to **-1**.

useEffect

The **useEffect** method is one of the commonly used React hooks. This hook is used for asynchronous operations that have two arguments (fat arrow function and an optional dependency array). This dependency array is a must because, without it, the component will have an infinite re-render. The dependency array can take as many values as possible, which means that for any values that change inside the dependency array, the **useEffect** method will run again. We also have another fat arrow function that is used to clean up (**componentWillUnmount**). The **useEffect** method hook is usually used when we make API calls before our component mounts. Let us look at an example of the **useEffect** method hook:

```
1. import React,{useEffect,useState} from 'react'
2.
3.
4. const MyApp={()=>{
5.   const [data,setData]=useState([]);
6.   useEffect(()=>{
7.     const fetchData=async()=>{
8.
9.       return
10.      fetch('https://jsonplaceholder.typicode.com/todos')
11.      .then(response => response.json())
12.      .then(json => setData(json))
13.    } ;
14.    fetchData();
15.  },[])
```

```
14.   return (  
15.     <ul>{  
16.       data.length >0 && data.map(item=>{  
17.         return <li key={item.id}>{item.title}</li>  
18.       })  
19.     }</ul>  
20.   )  
21. }  
22.  
23.  
24.  
25. export default MyApp;
```

In the preceding example, we used the **useState** hook and the **useEffect** hook. We made a call to the <https://jsonplaceholder.typicode.com/todos> API and stored the array data in the **useState** method hook. We then looped through the array data and displayed it inside the functional component on line 16. Let us see how this will appear on the browser:

- 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
- quo adipisci enim quam ut ab
- molestiae perspiciatis ipsa
- illo est ratione doloremque quia maiores aut
- vero rerum temporibus dolor
- ipsa repellendus fugit nisi
- et doloremque nulla
- repellendus sunt dolores architecto voluptatum
- ab voluptatum amet voluptas
- accusamus eos facilis sint et aut voluptatem
- quo laboriosam deleniti aut qui
- dolorum est consequatur ea mollitia in culpa
- molestiae ipsa aut voluptatibus pariatur dolor nihil
- ullam nobis libero sapiente ad optio sint
- suscipit repellat esse quibusdam voluptatem incidunt
- distinctio vitae autem nihil ut molestias quo
- et itaque necessitatibus maxime molestiae qui quas velit
- adipisci non ad dicta qui amet quaerat doloribus ea
- voluptas quo tenetur perspiciatis explicabo natus
- aliquam aut quasi
- veritatis pariatur delectus
- nesciunt totam sit blanditiis sit
- laborum aut in quam

Figure 4.3: useEffect rendered list shows our useEffect displayed list

useLayoutEffect

The **useLayoutEffect** method is one of the commonly used React hooks. This hook is used for synchronous operations that have two arguments (fat arrow function and an optional dependency array). This dependency array is a must because, without it, the component will have an infinite re-render. The dependency array can take as many values as possible, which means that for any values that change inside the dependency array, the **useEffect** method will run again. We also have another fat arrow function that is used to clean up (**componentWillUnmount**). The **useEffect** method hook is usually used when we make API calls before our component mounts. The hook usually runs after the component has been mounted. Let us look at an example of the **useEffect** method hook:

```

1. import React,{useLayoutEffect,useState} from 'react'
2.
3.
4. const MyApp={()=>{
5.   const [data,setData]=useState([]);
6.   useLayoutEffect(()=>{
7.     const fetchData=async()=>{
8.
8.                                     return
       fetch('https://jsonplaceholder.typicode.com/todos')
9.       .then(response => response.json())
10.      .then(json => setData(json))
11.    } ;
12.    fetchData();
13.  },[])
14.  return (
15.    <ul>{
16.      data.length >0 && data.map(item=>{
17.        return <li key={item.id}>{item.title}</li>
18.      })
19.    }</ul>
20.  )
21.}
22.
23.
24.
25. export default MyApp;

```

In the preceding example, we used the **useState** hook and the **useEffect** hook. We made a call to the <https://jsonplaceholder.typicode.com/todos> API and stored the array data in the **useState** hook. We then looped through the array data and displayed it inside the functional component on line 16.

useMemo

The **useMemo** method is one of the commonly used React hooks. The **useMemo** method is used for memoization in react. The **useMemo** method memoizes a computation and returns its value. This hook is very good for performance because it will cache the computed value and return it if the values in the dependency array do not change. If the values change, then the **useMemo** is run again, and the newly computed value is returned. Let us see the **useMemo** effect in action with an example:

```
1. import React,{useMemo} from 'react'
2.
3.
4. const MyApp={()=>{
5.   const result=useMemo(()=>{
6.     return 1+1;
7.
8.   },[])
9.
10.  return (
11.    <p>{result}</p>
12.  )
13. }
14.
15.
16.
17. export default MyApp;
```

useCallback

The **useCallback** method is one of the commonly used React hooks. The **useCallback** method is used for memoization in react. The **useCallback** method memoizes a computation and returns its function. This hook is very good for performance because it will cache the computed value and return it

if the values in the dependency array do not change. If the values change, then the **useCallback** is run again, and the new computed function is returned. Let us see the **useCallback** effect in action with an example:

```
1. import { useState, useCallback } from "react";
2.
3. const App = () => {
4.   const [count, setCount]=useState(0);
5.   const handleCount = useCallback(() => {
6.     setCount((currentValue) => currentValue + 1);
7.   }, []);
8.
9.   return (
10.     <>
11.
12.       <hr />
13.       <div>
14.         Count: {count}
15.         <button onClick={handleCount}>+</button>
16.       </div>
17.     </>
18.   );
19. };
20.
21. export default App;
```

[useRef](#)

The **useRef** method is one of the commonly used React hooks. The **useRef** method is used for the following two major reasons:

- Points to an element in the DOM.
- Used to store values of elements even after re-rendering.

Let us look at what the **useRef** method does with the aid of an example:

```
1. import { useRef } from "react";
2.
3. const App = () => {
4.   const inputRef=useRef();
5.
6.   const handleChange={()=>{
7.     inputRef.current.style.backgroundColor='green';
8.   }}
9.   return (
10.     <>
11.
12.       <hr />
13.       <div>
14.         <input type="text" ref={inputRef}      onChange=
           {handleChange}/>
15.       </div>
16.     </>
17.   );
18. };
19.
20. export default App;
```

The preceding example just shows us one of the ways to use the **useRef** hook. Here, we are pointing to an input element in the dom. When we have the **onChange** event called, we call the **handleChange** function, which we declared on line 6, and set the pointed element background color to green. Let us see how this looks in the browser:



Figure 4.4a: Showing our useRef example

This is before we key in values:


A screenshot of a web browser window. At the top, there is a green header bar with the name 'Anthony' written in white text. Below the header, there is a text input field containing the same name 'Anthony'.

Figure 4.4b: Showing our useRef example

This is after we enter values in the box. We will further look at the other scenario where we can persist values using **useRef** in-between re-renders with an example:

```
1. import { useRef } from "react";
2.
3. const App = () => {
4.   const inputRef=useRef();
5.
6.   const handleChange=()=>{
7.     console.log(inputRef.current.value);
8.   }
9.   return (
10.    <>
11.
12.      <hr />
13.      <div>
14.        <input type="text" ref={inputRef}      onChange=
          {handleChange}/>
15.      </div>
16.    </>
17.  );
18. };
19.
20. export default App;
```

The preceding example will print to the console on line 7 as the following:

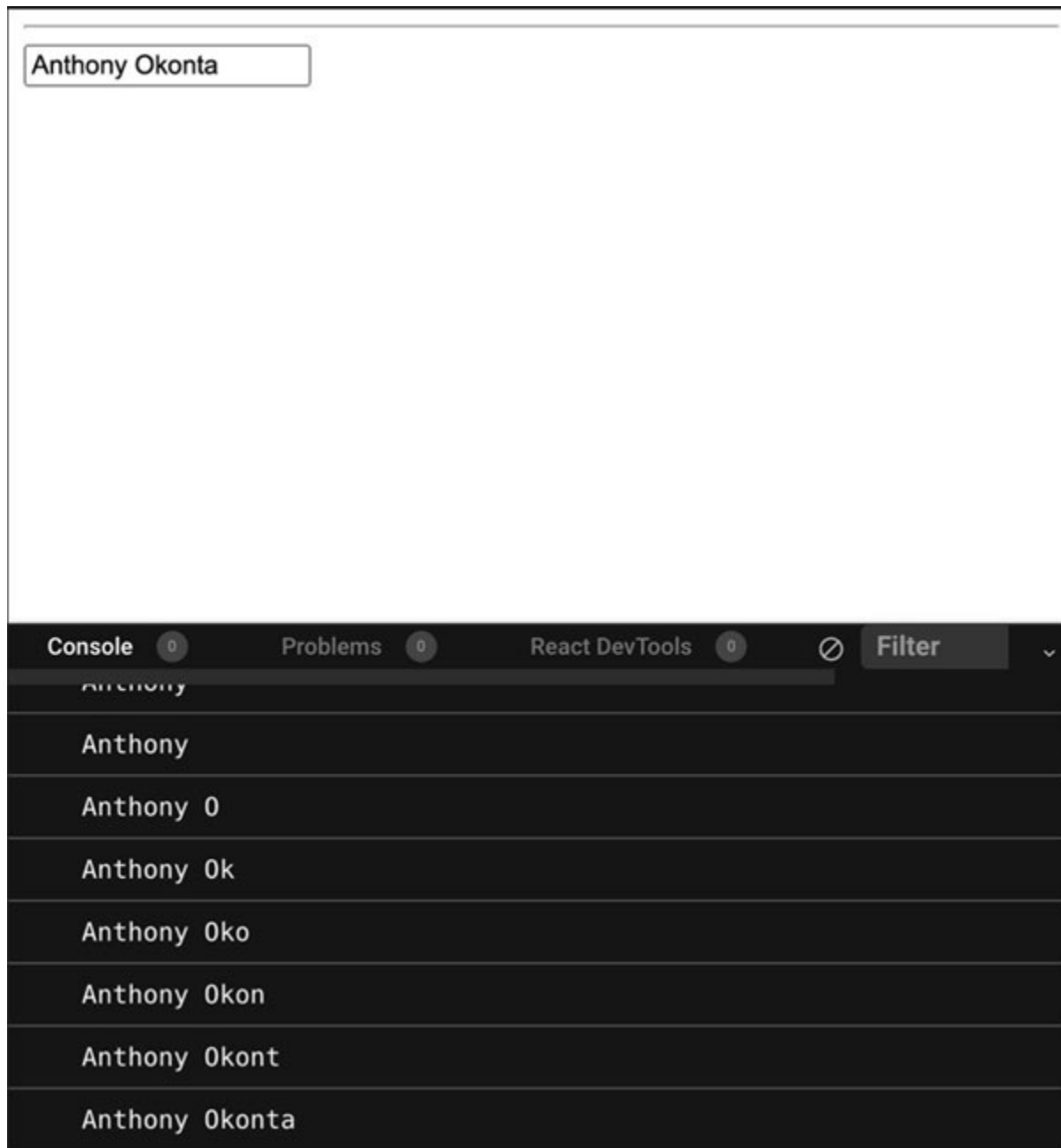


Figure 4.5: Showing our useRef example

You can see from the preceding code how could persist the value **Anthony Okonta** could be in between re-renders.

useContext

The `useContext` method is one of the commonly used React hooks. The `useContext` method is used to pass values down the component tree from a

parent component to a child component far down the component tree. This **useContext** hook solves the issue of **props** drilling, where you must pass values from one component to the other irrespective of the component needing the value. Let us look at this with the aid of an example. We would look at a problem of props drilling and look at how to solve it with context API.

```
1. import { useState,useRef,useContext,createContext } from
   "react";
2.
3. const App = () => {
4.   const appBackgroundContext=createContext();
5.   const [defaultColor, setDefaultColor] = useState("red");
6.
7.   const ComponentTwo=(value)=>{
8.     const {color}=value
9.     return <div style={{backgroundColor:color}}>
10.       hello
11.     </div>
12. }
13. const ComponentThree=(value)=>{
14.   const {color}=value
15.   return <div style={{backgroundColor:color}}>
16.     hello
17.   </div>
18. }
19. const ComponentFour=(value)=>{
20.   const {color}=value
21.   return <div style={{backgroundColor:color}}>
22.     hello
23.   </div>
24. }
```



```

25.  const ComponentFive=(value)=>{
26.    const {color}=value
27.    return <div style={{color:color}}>
28.      hello {color}
29.    </div>
30.  }
31.  return (
32.    <>
33.
34.    {
35.      <appBackgroundContext.Provider value={defaultColor}>
36.        <ComponentTwo color="red"/>
37.        <ComponentThree color="red"/>
38.        <ComponentFour color="red"/>
39.        <ComponentFive color="red"/>
40.      </appBackgroundContext.Provider>
41.    }
42.    </>
43.  );
44. };
45.
46. export default App;

```

In the preceding example, we have an issue passing parameters to **ComponentTwo**, **ComponentThree**, **ComponentFour**, and **ComponentFive**. We are passing parameters as props to all these components, which are not really needed because this will cause props to drill. To solve this, we would use the **useContext** hook to solve the following problem:

```

1. import { useState,useRef,useContext,createContext } from
   "react";
2.

```

```
3. const App = () => {
4.   const appBackgroundColor=createContext();
5.   const [defaultColor, setDefaultColor] = useState("red");
6.
7.   const ComponentTwo={()=>{
8.     const context=useContext(appBackgroundColor)
9.     return <div style={{backgroundColor:context}}>
10.       hello
11.     </div>
12.   }
13.   const ComponentThree={()=>{
14.     const context=useContext(appBackgroundColor)
15.     return <div style={{backgroundColor:context}}>
16.       hello
17.     </div>
18.   }
19.   const ComponentFour={()=>{
20.     const context=useContext(appBackgroundColor)
21.     return <div style={{backgroundColor:context}}>
22.       hello
23.     </div>
24.   }
25.   const ComponentFive={()=>{
26.     const context=useContext(appBackgroundColor);
27.     return <div style={{color:context}}>
28.       hello {context}
29.     </div>
30.   }
31.   return (
32.     <>
```

```

33.
34.    {
35.        <appBackgroundColor.Provider value={defaultColor}>
36.            <ComponentTwo/>
37.            <ComponentThree/>
38.            <ComponentFour/>
39.            <ComponentFive/>
40.        </appBackgroundColor.Provider>
41.    }
42. </>
43. );
44. };
45.
46. export default App;

```

Custom hooks

Custom hooks are basically writing your own hooks. These hooks are functions that start with the word use. These custom hooks follow the same principle of reacting hooks that we mentioned earlier. Let us look at an example of the same:

```

1. import {useEffect,useState} from 'react';
2.
3. const useApi=(endpoint)=>{
4.   const [data,setData]=useState([]);
5.
6.   useEffect(()=>{
7.     const ApiCall=async()=>{
8.       return fetch(endpoint).then(response=>
9.         response.json()).then(result=>{

```

```

10.     })
11.   };
12.   ApiCall();
13. },[endpoint]);
14.   return {data}
15. }
16.
17. export default useApi

```

The preceding code is a custom hook that takes in an endpoint as a generic API, runs the **useEffect** asynchronous hook, and returns the data to the calling component:

```

1. import useApi from './hooks'
2. const App = () => {
3.                                     const
   {data}=useApi('https://jsonplaceholder.typicode.com/todos')
   ;
4.   return (
5.     <ul>
6.       {
7.         data && data.map(item=>{
8.           return <>
9.             <li>{item.id} {item.title} </li>
10.          </>
11.
12.        })
13.      }
14.    </ul>
15.  )
16. };
17.

```

18. export default App;

The preceding code calls the **useApi** custom hook defined in the previous code and passes the data down to the functional component. Kindly view the result of the functional component when returned.

- 1 delectus aut autem
 - 2 quis ut nam facilis et officia qui
 - 3 fugiat veniam minus
 - 4 et porro tempora
 - 5 laboriosam mollitia et enim quasi adipisci quia provident illum
 - 6 qui ullam ratione quibusdam voluptatem quia omnis
 - 7 illo expedita consequatur quia in
 - 8 quo adipisci enim quam ut ab
 - 9 molestiae perspiciatis ipsa
 - 10 illo est ratione doloremque quia maiores aut
 - 11 vero rerum temporibus dolor
 - 12 ipsa repellendus fugit nisi
 - 13 et doloremque nulla
 - 14 repellendus sunt dolores architecto voluptatum
 - 15 ab voluptatum amet voluptas
 - 16 accusamus eos facilis sint et aut voluptatem
 - 17 quo laboriosam deleniti aut qui
 - 18 dolorum est consequatur ea mollitia in culpa
 - 19 molestiae ipsa aut voluptatibus pariatur dolor nihil
 - 20 ullam nobis libero sapiente ad optio sint
 - 21 suscipit repellat esse quibusdam voluptatem incidunt
 - 22 distinctio vitae autem nihil ut molestias quo
 - 23 et itaque necessitatibus maxime molestiae qui quas velit
 - 24 adipisci non ad dicta qui amet quaerat doloribus ea
 - 25 voluptas quo tenetur perspiciatis explicabo natus
 - 26 aliquam aut quasi
 - 27 veritatis pariatur delectus
 - 28 nesciunt totam sit blanditiis sit
 - 29 laborum aut in quam
 - 30 nemo perspiciatis repellat ut dolor libero commodi blanditiis omnis
 - 31 repudiandae totam in est sint facere fuga
-

Figure 4.6: Showing our custom useEffect hook to display our list

Custom components exist in so many ways. You can decide to use it for extraction of logic, and it can make you not repeat yourself in a functional component but remember the rules before using them.

Conclusion

In this chapter, we looked into what React hooks are, their advantages, and their rules. We also looked at how to write custom hooks which are very helpful in extracting logic.

In the upcoming chapter, we will look at styling and how to properly style a react component differently.

CHAPTER 5

Styling

Introduction

This chapter will cover how to style React components. This will include preprocessors like Scss and Less. We will also cover styled-components and inline styles. Also, we will cover modular CSS.

Structure

The following topics are to be covered:

- Styling a component
- Types of styling (inline styling, preprocessors (Sass and Less), CSS, and styled-components
- Pros and cons of different styling
- Conventions for writing styles

Objectives

In this chapter, we will look at how to style a component and the different types of styling. We will also look at the pros and cons of the different types of styling. We will conclude by looking at some conventions for writing styles.

Styling a component

Styling is very important in every Web application because styling makes it very attractive to the user and gives the user a good experience. There are different ways to achieve this in React. We are going to look at some of the common ways of styling the React component to give a great view to the user.

Types of styling

So, in React and generally in website or Web applications, there are different types/ways of styling the application. We will look at some of them with detailed examples.

- Inline CSS
- External/referential CSS
- Sass
- Bootstrap
- Styled components
- Tailwind

Inline CSS

Inline CSS is one of the ways of styling websites or Web applications. It involves writing your styles on the HTML elements directly. We will look at this with the aid of the following example:

```
1. export default function App() {  
2.   return (  
3.     <div>  
4.       <p style=  
        {{color: 'green', fontSize: '30px'}}>Welcome</p>  
5.     </div>  
6.   )  
7. }
```

The preceding example shows us a typical example of an inline style. We styled the paragraph by making the **color** of the text **green**, and the size grew by **30px**. This has some pros and cons, which we will look at subsequently. Let us see the following code and how it looks like in the browser:

Welcome

Figure 5.1: Demonstration of inline CSS

Pros

- Easy to write.
- The browser loads and applies the styles faster.

Cons

- Styles would be duplicated.

External CSS

External CSS is one of the ways of styling websites or Web applications. It involves writing all your styles in classes and IDs inside a file and referencing it inside our application or component. We will look at this with the aid of the following example:

```
1. .container{  
2.   display:flex;  
3.   justify-content:center;  
4. }
```

```
5. .header{
6.   color:green;
7.   font-size:50px;
8.
9. }
```

Here, we declared our style inside a separate file and named it **App.css**. We added a class named **header** and wrote **color** and **font-size** for it. We also declared a **container** class and used a flexbox for the **display** property. We want to **center** the content of our container. Let us look at how we would make use of the preceding file in the following application:

```
1. import './App.css'
2.
3. export default function App() {
4.   return (
5.     <div className="container">
6.       <p className="header">Welcome</p>
7.     </div>
8.   )
9. }
```

We imported the **App.css** file, which contains our CSS classes. We imported it from our directory on line 1 and applied it on lines 5 and 6 using the **className** parameter. Let us see how the code will look on the following browser:



Welcome

Figure 5.2: Demonstration of external CSS

Pros

- Easy to write.
- Keeps component clean because styles are abstracted to a file.
- Styles would not be repeated and can be applied in multiple places.

Cons

- Slow to load because the file would have to be downloaded by the browser first.

Sass

Sass (Synthetically Awesome Style Sheet) is a preprocessor. CSS is one method for styling React applications. It has the extension `.scss`. It has some great features, such as:

- Mixins
- Inheritance
- Nesting

We would look at how to use sass in our React application with the aid of an example. To get started using sass, please run the following command:

```
npm install sass
```

or

```
yarn add sass
```

Please make sure you have installed node.js before running the preceding command:

```
1. .container{
2.   display:flex;
3.   justify-content:center;
4. }
5. .header{
6.   color:green;
7.   font-size:50px;
8.
9. }
```

Here, we declared our style inside a separate file and named it **scss.css**. We added a class named **header** and wrote **color** and **font-size** for it. We also declared a container class and used a flexbox for the **display** property. We want to **center** the content of our container. Let us look at how we will make use of the preceding file in the following application:

```
1. import './App.scss'
2.
3. export default function App() {
4.   return (
5.     <div className="container">
6.       <p className="header">Welcome</p>
```

```
7.     </div>
8.   )
9. }
```

We imported the **App.scss** file that contains the CSS classes. We imported it from our directory in line 1 and applied it in lines 5 and 6 using the **className** parameter. Let us see how the code will look on the following browser:

A screenshot of a web browser displaying the word "Welcome" in a large, green, serif font. The text is centered on a plain white background.

Figure 5.3: Demonstration of scss CSS

Pros

- Breaks down complex styles into simple styles.

Cons

- Not easy to learn.

Bootstrap

Bootstrap is one of the popular open sources of CSS frameworks used to style the React Web application. Bootstrap has already inbuilt styles and classes, which can be applied to your application instantly. These inbuilt styles and classes are already responsive by default, so you do not need to bother about their responsiveness.

To get started using bootstrap, please run the following command:

```
npm install react-bootstrap bootstrap or yarn add react-bootstrap bootstrap
```


Please make sure you have installed node.js and yarn to be able to use yarn before running the preceding command.

Let us look at how this bootstrap works with the following example. Here, we just decided to declare a div that we will align all our HTML elements to the **center** of the page.

```
1. import './styles.css';
2. import 'bootstrap/dist/css/bootstrap.min.css';
3. export default function App() {
4.   return (
5.     <div className="d-flex justify-content-center ">
6.       <button className="btn btn-primary">Hello welcome to
       our Project</button>
7.     </div>
8.   );
9. }
```

Here, we imported the **styles.css** file on line 1, so we can use the **Container** class we created in it in line 5 since it is the **div** that will hold the entire component. We imported the bootstrap CSS on line two that was added when we ran the **npm install react-bootstrap bootstrap** or **yarn add react-bootstrap bootstrap** code. We added a button to our component and gave it a **className** of **btn btn-primary**, which gave it a nice blue background color.

Let us see how our code will look on the following browser:



Hello welcome to our Project

Figure 5.4: Demonstration of bootstrap CSS

Pros

- Already inbuilt styles for your use.
- Has responsive styles by default.

Cons

- Overriding a style can be difficult.

Styled components

Styled components are one of the commonly used ways of styling a react application. It involves the writing of styles in JavaScript. We will look at this in the following example. Before we look at this example, please first run the following command:

`npm install styled-components` or `yarn add styled-components`

Then add the following lines of code to your project:

```
1. import React,{useState} from 'react';
2. import styled from 'styled-components';
3.
4. const Button=styled.button`
5. background-color:#f4ebff;
6. width:50%;
7. height:100px;
8. border-radius:10px;
9. border-color:none;
10. border-width:0px;
11.
12. `;
13.
14. const Div=styled.div`
15. display:flex;
16. justify-content:center;
17. align-items:center;
18. flex-direction:column;
19. gap:20px;
20. margin-top:100px
21. `;
22.
23. const TextBox=styled.input.attrs({ type: 'text' })`
24. width:50%;
25. height:50px;
26. `
27. export default function App() {
28.   const [name,setName]=useState('');
29.   const onWelcomeClick=()=>{
30.     if(name){
```



```
31.     alert(`Welcome ${name}`)
32.   }
33.   else{
34.     alert('please enter your name')
35.   }
36. }
37. return (
38.   <Div>
39.     <label>Please enter your name</label><TextBox onChange=
      {(event)=>setName(event.target.value)}/>
40.     <Button onClick={onWelcomeClick}>Welcome</Button>
41.   </Div>
42. );
43. }
```

So, we imported our styled component from the library on line 3. So, we declared three styled components, namely, **Button**, **Div**, and **TextBox**, and gave them some CSS properties. We then applied the three styled components inside our component. Let us see how this looks on the browser. See [figure 5.5](#):



Figure 5.5: Demonstration of styled component css

This is how our application would look inside our browser. Let us look at another use case of styled components. Now, styled components can accept props and can be used to change the style. Let us look at this in code and in the browser.

```
1. import React,{useState} from 'react';
2. import styled from 'styled-components';
3.
4. const Button=styled.button`
5. background-color:#f4ebff;
6. width:50%;
7. height:100px;
8. border-radius:10px;
9. border-color:none;
```

```
10. border-width:0px;
11. color:${props => (props.title==='test' ? 'green'
    : 'black')}};
12. `;
13.
14. const Div=styled.div`
15. display:flex;
16. justify-content:center;
17. align-items:center;
18. flex-direction:column;
19. gap:20px;
20. margin-top:100px
21. `;
22.
23. const TextBox=styled.input.attrs({ type: 'text' })`
24. width:50%;
25. height:50px;
26. `
27. export default function App() {
28.   const [name,setName]=useState('');
29.   const onWelcomeClick={()=>{
30.     if(name){
31.       alert(`Welcome ${name}`)
32.     }
33.     else{
34.       alert('please enter your name')
35.     }
36.   }
37.   return (
38.     <Div>
```

```
39.         <label>Please enter your name</label>
        <TextBox onChange={(event)=>setName(event.target.value)}/>
40.         <Button title="test"onClick=
        {handleClick}>Welcome</Button>
41.     </Div>
42. );
43. }
```

Here, we added a **props.title** condition in line 11 to see if the parameter **title=== 'test'** then the color would be green; else, it would be black. In line 40, we can see the props value title added and the test assigned to it to meet our condition in line 11. Let us see how this will look like on the browser with or without the test. Refer to [figure 5.6](#):



Figure 5.6: Styled component with props

The preceding [figure 5.6](#) has the props **title='test'** in line 40:

```
<Button title="test" onClick={onWelcomeClick}>Welcome</Button>
```



Figure 5.7: Demonstration of the styled component with props:

The preceding picture has no props title in line 40:

```
<Button onClick={onWelcomeClick}>Welcome</Button>
```

Pros

- Reusable
- No **className** conflicts
- Dynamic styling
- React native support

Cons

- Performance can become an issue

Tailwind CSS

Many developers these days are going the way of Tailwind CSS because it is quicker to write and maintain in your react application. Tailwind CSS is a utility-first framework. It has classes that can give you control of your application, such as padding, margin, color, and so on. We will look at how this Tailwind CSS works.

```
npm install -D tailwindcss postcss autoprefixer and npx  
tailwindcss init -p
```

For more on how tailwind works, please visit <https://tailwindcss.com/docs/guides/create-react-app>.

Conventions of writing styles

We have looked at the different kinds of styles in our preceding topic, but now, we are going to look at some common conventions in writing styles. These include the following:

BEM

Block Element Modifier (BEM) is a common convention when writing CSS. It is used to break down large interfaces into independent blocks. We will look at the three blocks that make up the BEM in the following details:

Block

Block describes a purpose. They are usually represented by a class. It tells us what that class is doing at first glance. Let us look at the following example:

```
1. <div className="success"></div>
```

The preceding code is a simple **div** with a **className** of success. This **div** will show up on the success of an operation. The block has the following two rules:

- Blocks can be nested in each other.
- You can have any number of nesting levels.

Element

Element also describes a purpose. They are usually represented by a class. The element name is separated from the block with a double underscore (__). Let us look at the following example:

```
1. <div className="form">
2.   <input type='text' className='form__input' />
3.   <input type='submit' className='form__button' />
4. </div>
```

The preceding code is a simple parent **div** with a **className** of form. This **div** has two inputs inside of it which are text and submit. They both have **classNames** preceding with **form__button** and **form__input**, which signifies that this is an element of the block form.

Modifier

A modifier also describes the appearance, state, or behavior of a block or element. They are usually separated from the block or element name by a single underscore (_).

```
1. <div className="form">
2.   <input type='text' className='form__input_disabled' />
3.   <input type='submit' className='form__button_big' />
4. </div>
```

Conclusion

In this chapter, we learned how to style components and the benefits of styling components. We also learned the different types of styles and their pros and cons. We also looked at a naming convention when writing styles in react.

In the upcoming chapter, we will look at the server-side rendering and its benefits.

CHAPTER 6

Server-Side Renderings

Introduction

In this chapter, we will look at what server-side rendering is, when to use it, and its pros and cons. We will also look at a popular server-side framework (Next.js) and how it differs from the client-side react library.

Structure

The following topics are to be covered:

- What is server-side rendering?
- When to use server-side rendering like Next.js
- Pros and cons of using Next.js
- Pre-rendering in Next.js
- Next.js versus React.js
- Server-side rendering with Node.js and Express.js.

Objectives

In this chapter, we will learn how to use server-side rendering frameworks like Next.js. We will learn the differences, pros, and cons of Next.js versus React.js. We will look at some use cases on when to use server-side rendering like Next.js.

What is Server-Side rendering?

Server-side rendering (SSR) is basically rendering HTML pages to the client that are served from the server. The client makes a request to the server based on the user event; the server makes the page ready and returns

it to the client, which then renders the page. This request is done by the Web browser on the server-side.

When to use server-side rendering?

Server-side rendering is not used on every application. The following are some of the reasons why you will choose SSR:

- Better user experience
- Rapid feature development
- Performance

Pros of using Next.js

The following are some of the pros of using Next.js:

- **Great for SEO:** This is another benefit of using Next.js. Search engine optimization helps you increase traffic visits to your site, and Next.js provides this. It has the built-in functionality to make this happen.
- **Great community support:** Next.js has a very large community because the framework is still very much evolving and growing popular by the day.
- **Fast refresh:** When code changes are made on the Next.js application, the changes reflect within seconds on the browser.
- **Speed:** Next.js is very fast because most of the pages are pre-rendered on the server-side and served on the client side. This is a very big advantage of using next.js.
- **Image optimization:** This is one of the main benefits of using Next.js. This feature automatically optimizes images. It now supports AVIF images, enabling 20% smaller images compared to WebP.
- **Typescript support:** This is one of the main benefits of using Next.js Automatic TypeScript configuration and compilation.
- **Zero config**
- **Great developer experience:** One of the main benefits of using NextJS for our team is the developer experience. With NextJS, we

spend more time writing features and less time struggling with building tools and Webpack.

We also really enjoy how NextJS structures its projects into pages and how there is one way of doing routing.

- **Data fetching:** Rendering content in different ways, depending on the app's use case. This includes both pre-rendering with server-side rendering or static site generation and updating or creating content at runtime with incremental static regeneration.
- **Middleware:** This feature enables the use of code over configuration, so you can run code before a request is completed. You can change the response to requests and redirect the user from one route to another.
- **Data security:** Next.js does not freeze the browser to download and execute a whole lot of JavaScript code at once; it has the potential to dramatically improve metrics such as **total blocking time (TBT)**. It measures the amount of time blocked by the scripts before the user can interact with the Web app, and it should ideally be below 300 ms. The better your TBT, the quicker your Web app becomes useful for your users. That, in turn, makes it more probable to convert them into customers. Waiting for a page to interact is a significant turn-off, which might increase your bounce rate (the number of users that leave your page on the first URL before 30 seconds). That is another SEO-relevant factor.
- **Performance**

Cons of using Next.js

The following are some of the cons of using Next.js:

- **Development and management:** If you want to use NextJS to build an online store, yet you do not have an in-house team of developers, you will need a dedicated person to handle the development and management
- **Routing problems:** Since a file-based router limits what Next.js can do in routing through nodes, you will have to use the Node.js server if your project requires dynamic routes.

- **No state manager:** No built-in state manager is available in the framework Next.js. To get the most out of one, you will need another tool to do it.
- **Limited building time:** The time it takes to build or build websites and apps is restricted. Adding new pages to a website or app is no problem for Next.js. However, the static creation of the entire website means that the build time can be extremely long for apps with many pages.

Pre-rendering in Next.js

Next.js pre-renders every page. This means that HTML for each page is generated in advance, instead of it all being done by the client-side. There are two types of Pre-rendering, namely:

- Static generation
- Server-side rendering

Static generation

The HTML is generated at build time and will be reused on each request. To make a page, use Static Generation, either export the page component or export **getStaticProps** (and **getStaticPaths** if necessary). It is great for pages that can be pre-rendered ahead of a user's request. You can also use it with client-side rendering to bring in additional data. We will look at ways you can generate static which are as follows:

- Static page without data
- Static page with data

Static page without data

Next.js pre-renders these pages without fetching data from any source or backend. Let us look at this with the aid of an example:

```
1. function Contact(){  
2.   return <div>Contact</div>  
3. }
```

- 4.
5. export default Contact

The preceding example does not fetch any data from the backend or any source.

Static page with data

Some pages require fetching external data for pre-rendering. There are two ways this can be done:

- Content depends on external data
- Paths depend on external data

Content depends on external data

Your page might need to get data from an API from an external source. Let us look at an example of this:

```
1. function UserList({ users }) {  
2.   return (  
3.     <ul>  
4.       {users.map((user) => (  
5.         <li>{user.firstname}</li>  
6.         <li>{user.lastname}</li>  
7.       )}}  
8.     </ul>  
9.   )  
10. }  
11. export default UserList
```

The preceding example fetches this data on pre-render. Let us look at how this **getStaticProps** work with the following code:

```
1. function UserList({ users }) {  
2.   return (  
3.     <ul>{users.map((user) => (  
4.       <li>{user.firstname}</li>  
5.       <li>{user.lastname}</li>  
6.     )}}  
7.   )  
8. }
```

```

4.      <li>{user.firstname}</li>
5.      <li>{user.lastname}</li>)))}
6.    </ul>)
7.  }
8.
9. export async function getStaticProps() {
10.   const res = await fetch('https://.../users')
11.   const users = await res.json()
12.
13.   return {
14.     props: {
15.       users,
16.     },
17.   }
18. }
19.
20. export default UserList

```

Paths depend on data

For example, create a **.js** file called **users/[id].js** to show a single user detail based on its **id**. We need to export an **async** function called **getStaticPaths** from our **users/[id].js** file. Let us look at this with the aid of an example:

```

1. function UserList({ users }) {
2.   return (
3.     <ul>{users.map((user) => (
4.       <li>{user.firstname}</li>
5.       <li>{user.lastname}</li>)))}
6.     </ul>)
7.

```

```

8.  )
9. }
10. export async function getStaticPaths() {
11.   const response = await fetch('https://.../users')
12.   const users = await response.json()
13.
14.   const paths = users.map((user) => ({
15.     params: { id: user.id },
16.   }))
17.   return { paths, fallback: false }
18.
19. }
20. export async function getStaticProps({ params }) {
21.
22.   const response = await fetch(`https://.../
    users/${params.id}`)
23.   const users = await response.json()
24.
25.
26.   return { props: { users } }
27. }
28.
29. export default UserList

```

[GetServerSideProps](#)

The HTML is generated on each request by exporting the async function called **getServerSideProps**. Let us look at how this works with the aid of an example:

```

1. function UserList({ users }) {
2.   return (

```

```

3.    <ul>{users.map((user) => (
4.        <li>{user.firstname}</li>
5.        <li>{user.lastname}</li>))}
6.    </ul>)
7.  )
8. }
9.
10. export async function getServerSideProps() {
11.
12.   const res = await fetch(`https://.../users`)
13.   const data = await res.json()
14.
15.   return { props: { data } }
16. }
17.
18. export default UserList

```

[Next.js versus React.js](#)

Both Next.js and React.js are very good to use when building applications. We are now going to compare both in the following ways:

- Maintainability
- TypeScript
- Configuration
- Server-side rendering
- Image optimization

[Maintainability](#)

React App	Next.js
CRA is opinionated.	NextJs is also well maintained.

TypeScript support

React App	Next.js
<code>npx create-react-app my-app --template typescript</code> is used to add typescript from scratch with the app.	Start with configurations for typescript with <code>touch tsconfig.json</code>

Configuration

React App	Next.js
Configurations like Web pack config cannot be changed.	Almost everything is configurable.

Server-side rendering

React App	Next.js
CRA does not support SSR out of the box.	NextJs has different types of SSR. It supports SSR out of the box.

Image optimization

React App	Next.js
CRA does not have much to offer in image optimization.	Next.js has a built-in image component called <code>next/image</code> .

Server-side rendering with Node.js and Express.js.

Server-side rendering with Node.js and Express.js is another way to render react pages from the server side. The express.js creates a route that we can use to request specified pages based on the route from the user. You can find more details in the following link:

<https://www.digitalocean.com/community/tutorials/react-server-side-rendering>

Conclusion

We have successfully looked at what server-side rendering is and when to use it. We also looked at Next.js and its pros and cons with detailed examples. We concluded by looking at some of the differences between creating react app and Next.js.

In the upcoming chapter, we will look at data fetching.

CHAPTER 7

Data Fetching

Introduction

In this chapter, we will look at how to fetch data in our React application. We will also look at the different types of libraries/APIs we can use to fetch data remotely into our application. We will also look at the difference between REST API and GraphQL and how to cache the fetched data and use it in our application. We will also look at promises, their benefits, and what it solves in our react application.

Structure

The following topics are to be covered:

- Ways to fetch data from a server in React
- Rest API versus GraphQL
- Caching data from API and use in your application
- Promises

Objectives

In this chapter, we will learn how to use the different JavaScript APIs/libraries to fetch data from a server. We will look at the difference between Rest API and GraphQL also caching data in our application with detailed examples. We will also use promises to resolve fetched data in our application.

Fetching data in React

In most React applications, data from an API or server is likely needed for that application to work. We submit data from our application to the server to receive some sort of response. There are several ways to fetch/send this

data to the API or server, and we can do this by inbuilt API or external **npm** packages. The following are some of them, and we will look at them with the aid of an example:

- Fetch
- Axios

Fetch

This is a commonly used API in JavaScript and React applications. Fetch provides a generic definition of Request and Response objects (and other things involved with network requests). This will allow them to be used wherever they are needed in the future, whether it is for service workers, Cache API, and other similar things that handle or modify requests and responses or any kind of use case that might require you to generate your responses programmatically (that is, the use of computer program or personal programming instructions).

The **fetch()** method takes one mandatory argument, the path to the resource you want to fetch. It returns a Promise that resolves the response to that request. Let us look at this fetch in action with examples.

We are going to use <https://jsonplaceholder.typicode.com> as our API for this example:

```
1. import React,{useState,useEffect} from 'react'
2. export default function App() {
3.   const [postsArr,setPostsArr]=useState([]);
4.   const baseUrl='https://jsonplaceholder.typicode.com';
5.   useEffect(()=>{
6.     const fetchPosts=async()=>{
7.                                     return
       fetch(`${baseUrl}/posts`).then(res=>res.json()).then(respon
       se=>{
8.         setPostsArr(response);
9.       })
10.    };

```

```
11.     fetchPosts();
12.   }, [])
13.   return (
14.     <div className="App">
15.       {
16.         postsArr.map(post=>{
17.           return <ul key={post.id}>
18.             <li>{post.title}</li>
19.           </ul>
20.         })
21.       }
22.     </div>
23.   );
24. }
```

The preceding code shows how to fetch all data from <https://jsonplaceholder.typicode.com/posts> and displays it inside our React application using fetch. The fetch takes in the URL and path to the resource and pulls the data from the server.

Let us look at how this data is rendered in a browser.

-
- sunt aut facere repellat provident occaecati excepturi optio reprehenderit
 - qui est esse
 - ea molestias quasi exercitationem repellat qui ipsa sit aut
 - eum et est occaecati
 - nesciunt quas odio
 - doloreum magni eos aperiam quia
 - magnam facilis autem
 - dolore dolore est ipsam
 - nesciunt iure omnis dolore tempora et accusantium
 - optio molestias id quia eum
 - et ea vero quia laudantium autem
 - in quibusdam tempore odit est dolore
 - dolorum ut in voluptas mollitia et saepe quo animi
 - voluptatem eligendi optio
 - eveniet quod temporibus
 - sint suscipit perspiciatis velit dolorum rerum ipsa laboriosam odio
 - fugit voluptas sed molestias voluptatem provident
-

Figure 7.1: Shows Post list rendered in our application

So, what if we want to fetch a single Post from our Post API? We can do this with the following code:

```
1. import React,{useState,useEffect} from 'react'
2. export default function App() {
3.   const [post,setPost]=useState({});
4.   const baseUrl='https://jsonplaceholder.typicode.com';
```

```

5.   useEffect(()=>{
6.     const fetchPosts=async()=>{
7.                                     return
       fetch(`${baseUrl}/posts/1`).then(res=>res.json()).then(response=>{
8.         setPost(response);
9.       })
10.    };
11.    fetchPosts();
12.  }, [])
13.  return (
14.    <div className="App">
15.      <p>Title: {post.title}</p>
16.      <p>Body: {post.body}</p>
17.    </div>
18.  );
19. }

```

Title: sunt aut facere repellat provident occaecati excepturi optio reprehenderit

Body: quia et suscipit suscipit recusandae consequuntur expedita et cum
reprehenderit molestiae ut ut quas totam nostrum rerum est autem sunt rem eveniet
architecto

Figure 7.2: Shows a single Post rendered in our application.

Let us see in the following code how we can add a new post:

```

1. import React, {useEffect} from 'react'
2. export default function App() {

```

```

3.   const baseUrl='https://jsonplaceholder.typicode.com';
4.
5.   useEffect(()=>{
6.     const postBody=JSON.stringify({
7.       title:"This is a test",
8.       body:"quia et suscipit\nsuscipit recusandae"
9.     })
10.    const addPosts=async()=>{
11.      return fetch(`${baseUrl}/posts`,{
12.        method:'POST',
13.        body:postBody
14.      }).then(response=>{
15.        console.log(response)
16.      })
17.
18.    };
19.    addPosts();
20.  },[])
21.  return (
22.    <div className="App">
23.
24.    </div>
25.  );
26. }

```

Axios

Axios is a promise-based HTTP Client for node.js and the browser. It is isomorphic (= it can run in the browser and nodejs with the same codebase). On the server-side, it uses the native node.js HTTP module, whereas on the client (browser), it uses **XMLHttpRequests**.

Let us see how we can use Axios to fetch data from <https://jsonplaceholder.typicode.com>.

Before we use the Axios, run the following command:

npm install axios

or

yarn add axios

```
1. import React, {useEffect, useState} from 'react';
2. import axios from 'axios';
3.
4. const baseUrl='https://jsonplaceholder.typicode.com'
5.
6. export default function App() {
7.   const [postList, setPostList]=useState([]);
8.   useEffect(()=>{
9.     const getAllPosts=async()=>{
10.       return axios.get(`${baseUrl}/posts`).then(response=>{
11.         setPostList(response.data)
12.       })
13.     };
14.     getAllPosts()
15.   }, [])
16.   return (
17.     <div className="App">
18.       {
19.         postList.map(item=>{
20.           return <ul><li>{item.title}</li></ul>
21.         })
22.       }
23.     </div>
24.   );
```


25. }

The preceding code shows how to fetch all data from <https://jsonplaceholder.typicode.com/posts> and displays it inside our React application using Axios. The fetch takes in the URL and path to the resource and pulls the data from the server.

Let us look at how this data is rendered in a browser.

-
- sunt aut facere repellat provident occaecati excepturi optio reprehenderit
 - qui est esse
 - ea molestias quasi exercitationem repellat qui ipsa sit aut
 - eum et est occaecati
 - nesciunt quas odio
 - doloreum magni eos aperiam quia
 - magnam facilis autem
 - dolore dolore est ipsam
 - nesciunt iure omnis dolore tempora et accusantium
 - optio molestias id quia eum
 - et ea vero quia laudantium autem
 - in quibusdam tempore odit est dolore
 - dolorum ut in voluptas mollitia et saepe quo animi
 - voluptatem eligendi optio
 - eveniet quod temporibus
-

Figure 7.3: Shows all Post rendered in our application.

To fetch a single Post from our Post API, we can use the following code:

1. `import React, { useEffect, useState } from "react";`
2. `import axios from "axios";`
- 3.
4. `const baseUrl = "https://jsonplaceholder.typicode.com";`
- 5.

```

6. export default function App() {
7.   const [post, setPost] = useState({});
8.   useEffect(() => {
9.     const getAllPosts = async () => {
10.                                     return
      axios.get(`${baseUrl}/posts/1`).then((response) => {
11.       setPost(response.data);
12.     });
13.   };
14.   getAllPosts();
15. }, []);
16. return (
17.   <div className="App">
18.     <p>Title: {post.title}</p>
19.     <p>Body: {post.body}</p>
20.   </div>
21. );
22. }

```

The preceding code shows how to fetch all data from <https://jsonplaceholder.typicode.com/posts/1> and displays it inside our React application using Axios. The fetch takes in the URL and path to the resource and pulls the data from the server.

Let us have a look at how this data is rendered in a browser:

Title: sunt aut facere repellat provident occaecati excepturi optio reprehenderit

Body: quia et suscipit suscipit recusandae consequuntur expedita et cum
reprehenderit molestiae ut ut quas totam nostrum rerum est autem sunt rem
eveniet architecto

Figure 7.4: Shows a single Post rendered in our application.

GraphQL

GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools.

Send a GraphQL query to your API and get exactly what you need, nothing more and nothing less. GraphQL queries always return predictable results. Apps using GraphQL are fast and stable because they control the data they get, not the server.

Rest API versus GraphQL

GraphQL	Rest API
A query language for solving common problems when integrating APIs.	An architectural style is largely viewed as a conventional standard for designing APIs.
Deployed over HTTP using a single endpoint that provides the full capabilities of the exposed service.	Deployed over a set of URLs where each of them exposes a single resource.
Uses a client-driven architecture.	Uses a server-driven architecture.
Allows for schema stitching and remote data fetching.	Simplifying work with multiple endpoints requires expensive custom middleware.

Caching data

In most reach applications, it is normal that we compute data or fetch data from an API. However, this data cannot be computed all the time or fetched all the time when there is a render of the component. There are several ways to achieve the caching of data in React. Let us look at them with the following examples:

- Memoization
- State management

Memoization

Memoization is an optimization technique used primarily to speed up computer programs by **storing the results of expensive function calls** and returning the cached result when the same inputs occur again.

Memoizing, in simple terms, means **memorizing** or storing in memory. A memoized function is usually faster because if the function is called subsequently with the previous value(s), then instead of executing the function, we will be fetching the result from the cache.

In React, we can achieve memorization in the following two ways.

- `useMemo`
- `useCallback`

useMemo

React has a built-in hook called **useMemo** that allows you to memoize expensive functions so that you can avoid calling them on every render. You simply pass in a function and an array of inputs, and **useMemo** will only recompute the memoized value when one of the inputs has changed. Let us look at **useMemo** with the following example:

```
1. import React, {useMemo} from 'react'
2. const values = [3, 9, 6, 4, 2, 1];
3. const MyApp=()=>{
4.
5.   const result=useMemo(()=>{
6.
7.     return values.sort().join(',')
8.   },[])
9.
10.  return (
11.    <p>{result}</p>
12.  )
13. }
14. export default MyApp;
```

The preceding code will return 1, 2, 3, 4, 6, and 9 as a result in line 9 because the data has already been computed and will not recompute until the values array changes in line 2.

useCallback

React has a built-in hook called **useCallback** that allows you to memoize expensive functions so that you can avoid calling them on every render. You simply pass in a function and an array of inputs, and **useCallback** will only recompute the memoized function when one of the inputs has changed. Let us look at **useCallback** with the following example:

```
1. import React, {useCallback} from 'react'
```

```

2. const values = [3, 9, 6, 4, 2, 1];
3. const MyApp={()=>{
4.
5.   const callBackResult=useCallback(()=>{
6.
7.     return values.sort().join(',')
8.   },[])
9.
10.  return (
11.    <p>{callBackResult()}</p>
12.  )
13. }
14. export default MyApp;

```

The preceding code will return 2 as a result in line 11 because the data has already been computed and will not recompute until the values array changes in line 2.

State management

State management is another way of caching data and using it in your react application. The data that is cached from an API can be stored in our state management like and used in our application globally. Although the data is cached but on refreshing the page, it will lose the data, and you will need to re-fetch it again. Also, you can fetch the data and store it in React application state.

Promises

A promise is used to handle asynchronous calls in JavaScript. They are effective in managing multiple asynchronous calls, which can lead to callback hell. When an asynchronous call has been made, the promise returned is an object, and it can be in four states which are as follows:

- **Fulfilled:** This shows that the asynchronous call was successful

- **Rejected:** This shows that the asynchronous call has an error
- **Pending:** It neither fulfilled nor rejected
- **Settled:** It is either fulfilled or rejected.

Conclusion

In this chapter, we learnt about data fetching in React and the different libraries we can use to fetch data from an API or resource. We also looked at the difference between GraphQL and Rest API. We also looked at caching data and using it in our application and concluded by looking at promises.

In the upcoming chapter, we will look at building scalable applications in React

CHAPTER 8

Building Scalable Applications

Introduction

This chapter will cover ways to build scalable and high-performance applications in React. We will look at some best practices, concepts, and benefits of building such applications in React.

Structure

The following are topics to be covered:

- Conditional rendering
- Memoization
- Typescript
- File organization
- Separation of concerns
- Higher order components
- Solid principles

Objectives

In this chapter, we will look at how we can achieve scalable and high-performance applications in React. We will look at how we can organize our file structure and its benefits. We will also look at conditional rendering and ways to achieve this. We will also look at TypeScript and the separation of concerns. Finally, we will look at HOC and solid principles.

Conditional rendering

In building React application, there will always be times when you do not want to re-render the component except for some props or value changes, or

a condition is met; this is called **conditional rendering**. Let us look at ways to make conditional rendering in React:

- Ternary operation
- AND logical operation
- If else operation

Ternary operation

The ternary operator is commonly used in JavaScript for conditional operations. It is denoted by a single question mark ? followed by the first condition and : followed by the next condition. It basically tests if the first condition is met and, if yes, ignores the second condition and vice versa. Let us look at the following. It shows how we can use this ternary operator in our React component:

```
1. import React from 'react';
2.
3. const ComponentOne={()=>{
4.
5.   return (
6.     <div>
7.       Component One
8.     </div>
9.   )
10.
11. };
12. export default ComponentOne
```

The preceding code is a component that renders a **div** with **ComponentOne** as a text inside of it

```
1. import React from 'react';
2.
3. const ComponentTwo={()=>{
```

```
4.  
5.   return (  
6.     <div>  
7.       Component Two  
8.     </div>  
9.   )  
10.  
11. };  
12. export default ComponentTwo
```

The preceding code is a component that renders a **div** with **ComponentTwo** as text inside of it:

```
1. import {useState} from 'react'  
2. import ComponentOne from "./ComponentOne";  
3. import ComponentTwo from "./ComponentTwo";  
4. import "./styles.css";  
5.  
6. export default function App() {  
7.     const [isComponentVisible, setComponentVisible]=useState(false);  
8.   return (  
9.     <div className="App">  
10.      {  
11.        isComponentVisible?<ComponentOne/>  
12.        :    <ComponentTwo/>  
13.      }  
14.  
15.    </div>  
16.  );  
17. }
```

The preceding code will display only **Component Two** because the first condition is not met in line 11. Let us see what this would look like in our browser:

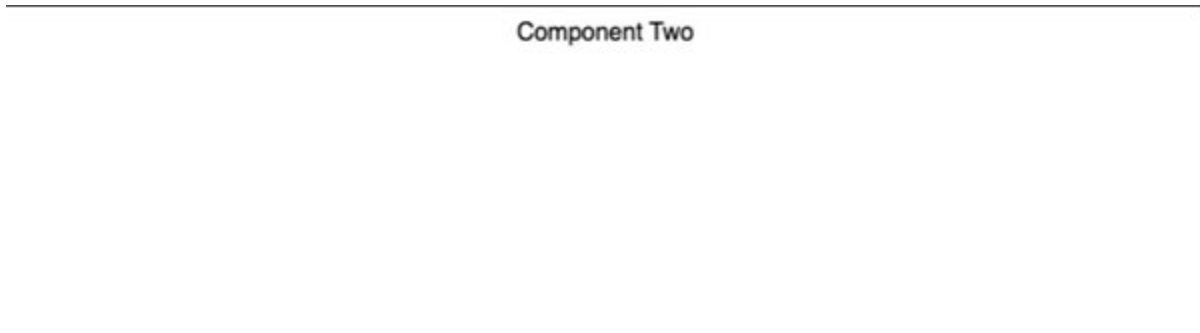


Figure 8.1: Showing our component after the Ternary operation

Logical operation

AND logical operator is commonly used in JavaScript for conditional operations. It is denoted by a double `&&`. It basically tests if the first condition is met and, if yes, runs the code after. Let us look at how we can use this `&&` logical operator in our React component as follows:

```
1. import React from 'react';
2.
3. const ComponentOne={()=>{
4.
5.   return (
6.     <div>
7.       Component One
8.     </div>
9.   )
10.
11. };
12.
13. export default ComponentOne
```

The preceding code is a component that renders a **div** with **Component_One** as text inside it

```
1. import React from 'react';
2.
3. const ComponentTwo={()=>{
4.
5.   return (
6.     <div>
7.       Component Two
8.     </div>
9.   )
10.
11. };
12.
13. export default ComponentTwo
```

The preceding code is a component that renders a **div** with **Component_Two** as text inside it

```
1. import {useState} from 'react'
2. import ComponentOne from './ComponentOne';
3. import ComponentTwo from './ComponentTwo';
4. import './styles.css';
5.
6. export default function App() {
7.   const [isComponentVisible,
8.     setIsComponentVisible]=useState(false);
9.   return (
10.     <div className="App">
11.       {
12.         isComponentVisible && <ComponentOne/>
```

```

13.     }
14.     {
15.         !isVisible && <ComponentTwo/>
16.     }
17.
18.
19. </div>
20. );
21. }

```

The preceding code will display only **Component Two** because the first condition is not met on line 11. The line 15 will be displayed because line 7 has a default value of false.

Let us see what this will look like in our browser:

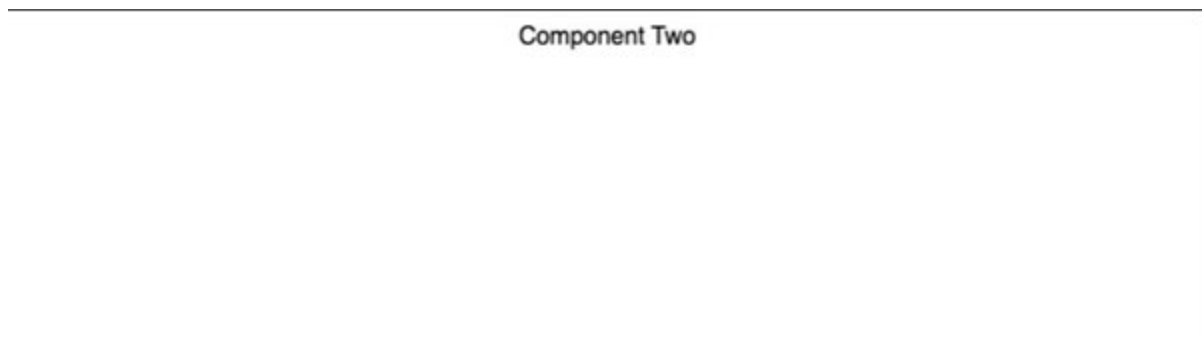


Figure 8.2: Showing our component after the logical operation

If else operation

If else operation is commonly used in JavaScript for conditional operations. It is denoted by the **if...else** syntax. It basically tests if the first condition is met and, if yes, runs the code after. Let us look at how we can use this **&&** logical operator in our React component as follows:

```

1. import React from 'react';
2.
3. const ComponentOne={()=>{
4.

```

```
5.   return (  
6.     <div>  
7.       Component One  
8.     </div>  
9.   )  
10.  
11. };  
12.  
13. export default ComponentOne
```

The preceding code shows a component that renders a **div** with **ComponentOne** as text inside of it:

```
1. import React from 'react';  
2.  
3. const ComponentTwo={()=>{  
4.  
5.   return (  
6.     <div>  
7.       Component Two  
8.     </div>  
9.   )  
10.  
11. };  
12.  
13. export default ComponentTwo
```

The preceding code shows a component that renders a **div** with **ComponentTwo** as text inside of it.

```
1. import {useState} from 'react'  
2. import ComponentOne from './ComponentOne';  
3. import ComponentTwo from './ComponentTwo';
```

```

4. import "./styles.css";
5.
6. export default function App() {
7.     const [isComponentVisible,
    setIsComponentVisible]=useState(false);
8.   let component;
9.   if(isComponentVisible){
10.     component=<ComponentOne/>
11.   }
12.   else{
13.     component=<ComponentTwo/>
14.   }
15.   return (
16.     <div className="App">
17.       {component}
18.
19.
20.     </div>
21.   );
22. }

```

The preceding code in line 9 tests if the **isComponentVisible** is a truthy value, and if yes, it assigns **componentOne** to the component declared in line 8, and if not, it assigns **componentTwo** to the component value declared in line 8. Let us see how this will look in our browser:

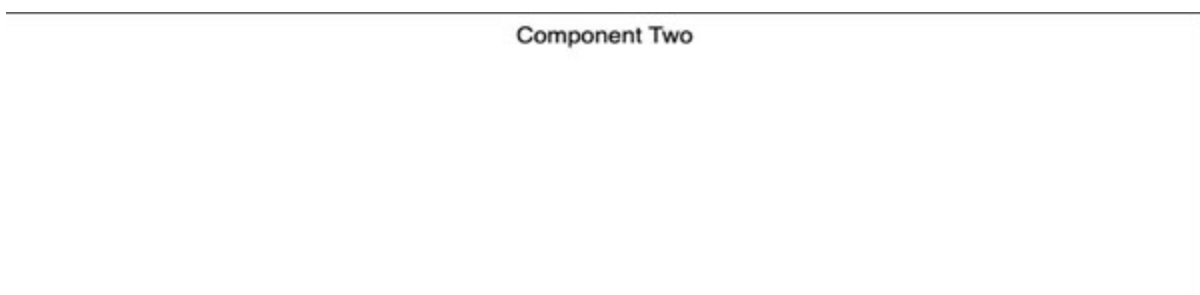


Figure 8.3: Showing component two after the condition is met

Memoization

Memoization is a way of preventing unnecessary re-computation of operations in programming. It helps make applications faster because it helps to cache computed results in memory and shows the same result when demanded instead of recomputing it every time. In React, there are two major ways to do this, which are as follows:

- `useMemo`
- `useCallback`

useMemo

If you want to memoize a value in React.js, use **`useMemo`** because it caches the computed value and returns it when needed. Let us see how this works in React.

```
1. import {useMemo} from 'react'
2. import "./styles.css";
3.
4. const numbersArr=[1,2,3,4,5,6];
5. export default function App() {
6.
7.   let sum=useMemo(()=>{
8.
9.     return numbersArr.reduce((curr,acc)=>curr+acc,0)
10.   },[])
11.   return (
12.     <div className="App">
13.       {sum}
14.
15.     </div>
16.   );
17. }
```


The preceding code is a demonstration of **useMemo**. Here, we do a summation of the **numbersArr** and memorize its value in line 7. The result of our calculation is **21**, which is displayed in line 13.

[useCallback](#)

If you want to memoize a function in React.js, use **useCallback** because it caches the computed function and returns it when needed. Let us see how this works in React.

```
1. import {useCallback} from 'react'
2. import "./styles.css";
3.
4. const numbersArr=[1,2,3,4,5,5,4,5];
5. export default function App() {
6.
7.                                     const
   callBackFunction=useCallback(()=>numbersArr.reduce((current
   Val,accum)=> currentVal+accum,0), [])
8.   return (
9.     <div className="App">
10.       <h1>{callBackFunction()}</h1>
11.     </div>
12.   );
13. }
```

The preceding code is a demonstration of **useCallback**. Here, we do a summation of the **numbersArr** and memorize its function in line 7. The result of our calculation is 29, which is displayed in line 10.

[TypeScript](#)

TypeScript is a superset of JavaScript. TypeScript helps to build scalable applications because it is strongly typed. TypeScript compiles to JavaScript at runtime with the help of Babel. We have used some TypeScript examples in this book so far, but we would look at the concepts and things to note

when building applications with TypeScript. To create a react TypeScript application, we use the following commands:

```
npx create-react-app myapp --template typescript
```

Here, **myapp** is the name of our application. It is forbidden to use any capital letter when stating the name of the app.

TypeScript has some advantages that can make your application scalable, which include the following:

- Its strongly typed features reduce errors
- Data types are easily defined

File organization

Yes, we know React is an opinionated library that gives you the opportunity to design it how you like, but if you want to achieve a good application, it is best you follow some best practices on how to name and organize your files and folders for your application. We will look at some suggested ways in which you can organize your application folder and files.

- **Components:** Just as the name components is pronounced, you need to put all your components in this folder. This folder would contain your JSX files, CSS files, and types (if you are using TypeScript).
- **Assets:** Assets are where we have media files such as images, some CSS (if you like), videos, and so on.
- **Helpers:** The helper folder is where we put common reusable functions. These functions are called whenever they are needed around the application
- **Constants:** The constants folder is where you put things that will not change.
- **API:** A folder is where you put your calls to your API and its functions.
- **Hooks:** The hooks folder is where one puts custom hooks.

Separation of concerns

Separation of concerns in software development is building an application in different modules, and each module is doing only one thing or addresses a concern. This will make your application more robust and scalable. So let us assume we have an application that has two components, namely, Login, Register, and an API to call. We want to separate these two components from each other so that they can be independent and do what they are created for, which is Login and Register a user by calling the API. This will help our application to be loosely coupled:

```
1. .login{
2.   display:flex;
3.   flex-direction:column;
4.   gap:20px;
5. }
```

The preceding code is CSS for our login component that follows-

```
1. import React from 'react';
2. import "./styles.css";
3.
4. export default function Login() {
5.
6.   return (
7.     <div>
8.       <form className="login">
9.         <label htmlFor="username">Username:</label>
10.          <input id="username" type="text"
11.            placeholder="username"/>
12.          <label htmlFor="password">Password:</label>
13.          <input id="password" type="password"
14.            placeholder="password"/>
15.          <input type="submit" value="Login"/>
16.        </form>
17.      </div>
18.    );
19.  }
20. }
```

```
16.     </div>
17.   )
18. }
```

The preceding component is for **Login**. This component does one thing alone, which is handle user login. This is how our login component will look on rendering, as shown in [figure 8.4](#):

Username:

Password:

Figure 8.4: Our login component

Let us now look at the CSS for our registration component:

```
1. .register{
2.   display:flex;
3.   flex-direction:column;
4.   gap:10px;
5. }
```

The preceding code is CSS for our registration component that follows:

```
1. import React from 'react';
2. import "./styles.css";
3.
4. export default function Register() {
5.
6.   return (
7.     <div>
8.       <form className="register">
9.         <label htmlFor="firstName">FirstName:</label>
```

```
10.         <input id="firstName" type="text"
placeholder="firstName"/>
11.         <label htmlFor="lastName">LastName:</label>
12.         <input id="lastName"
type="text" placeholder="lastName"/>
13.         <label htmlFor="username">Username:</label>
14.         <input id="username"
type="text" placeholder="username"/>
15.         <label htmlFor="password">Password:</label>
16.         <input id="password"
type="password" placeholder="password"/>
17.
18.         <input type="submit" value="Sign Up"/>
19.     </form>
20. </div>
21. )
22. }
```

The preceding code component is for **Register**. This component does one thing alone, which is handle user registration. This is how our registration component will look as follows on rendering:

FirstName:
<input type="text" value="firstName"/>
LastName:
<input type="text" value="lastName"/>
Username:
<input type="text" value="username"/>
Password:
<input type="password" value="password"/>
<input type="submit" value="Sign Up"/>

Figure 8.5: *Our Register component*

So, for us to use these components in our projects, we should import them into the place we want to use them as against writing inside the same file. Let us look at what I mean by this:

```
1. import React, { useState } from "react";
2. import Login from "../Login";
3. import SignUp from "../SignUp";
4. import "../styles.css";
5.
6. const App = () => {
7.   const [activePath, setActivePath] = useState("login");
8.
9.   const onNav=(e)=>{
10.     setActivePath(e);
11.   }
12.
13.   return (
14.
15.     <div>
16.       <div className="nav">
17.         <button onClick={()=>onNav("login")}>Login</button>
18.         <button onClick={()=>onNav("Register")}>
19.           Register</button>
20.       </div>
21.
22.
23.       <div className="login">
24.         {activePath === "login" && <Login/>}
25.         {activePath !== "login" && <SignUp />}
```

```
26.     </div>
27.   </div>
28. );
29. };
30.
31. export default App;
```

LoginRegister

Username:

username

Password:

password

Sign Up

Figure 8.6: Our component, when rendered the first time

This is how our application would look on render and will look like as shown as follows when the Register button is clicked:

LoginRegister

FirstName:

firstName

LastName:

lastName

Username:

username

Password:

password

Sign Up

Figure 8.7: Component when the register button is clicked

So let us make a call to our API to login or register users with different paths instead of calling the API on individual files. Let us see this with the following example:

```
1. const baseUrl = "http://api.com";
2.
3. const httpJsonResponseResolver = (response) => {
4.   if (!response.ok) {
5.     return Promise.reject(response.statusText);
6.   }
7.   return Promise.resolve(response.json());
8. };
9.
10. export const Post = (path, data) => {
11.   return fetch(`${baseUrl}/${path}`, JSON.stringify(data))
12.     .then(httpJsonResponseResolver)
13.     .then((response) => {
14.       return response;
15.     });
16. };

```

So, the preceding code has a **Post** method, which will be used to post data from our app to our API from everywhere in the app as against calling the API from every component in our app. So, we can import this into our **Login** or **Register** components and pass the required parameters to it. Let us look at this in action for login:

```
1. import React, {useState} from 'react';
2. import {Post} from './api';
3. import "./styles.css";
4.
5. export default function Login() {

```



```
6.  const [username, setUsername]=useState('');
7.    const [password, setPassword]=useState('');
8.    const onSubmit= async(event)=>{
9.      event.preventDefault();
10.     const data={
11.       username,
12.       password
13.     };
14.     const login= await Post(`login`,data)
15.       .then(response=>response.json())
16.       .then(data=>{
17.         if(data){
18.
19.         }
20.       })
21.   }
22.   return (
23.     <div>
24.       <form className="login" onSubmit={onSubmit}>
25.         <label htmlFor="username">Username:</label>
26.         <input id="username"onChange={(e)=>
setUsername(e.target.value)}    type="text"
placeholder="username"/>
27.         <label htmlFor="password">Password:</label>
28.         <input id="password"onChange=
{(e)=>set
                                     Password
(e.target.value)}    type="password"placeholder="password"/
>
29.
30.         <input type="submit" value="Login"/>
31.       </form>
```

```
32.     </div>
33.   )
34. }
```

The preceding code explains that **onSubmit** of the form details, the API **Post** method is called because we have already exported this from our API file, and the path and the data is passed as an argument, which will return a promise.

Higher order component

A higher order component is a function that takes in a component and returns a new component. Let us look at an example of how this HOC works:

```
1. import React from 'react'
2.
3. const higherOrderComponent = Component => {
4.   function HOC =()=> {
5.     return <Component />
6.   }
7.   return HOC
8. }
```

In the preceding example, **higherOrderComponent** is a function that takes a component called **Component** as an argument. We have created a new component called **HOC** which returns the **<Component/>** from its render function.

We can invoke the **HOC** as follows:

```
const HOC = higherOrderComponent(MyComponent);
```

Solid principles

Solid principles are object design principles that were written by Robert Martin in 2000. Solid is an acronym for the following:

- **Single Responsibility Principle (SRP)**

- **Open/Closed Principle (OCP)**
- **Liskov Substitution Principle (LSP)**
- **Interface Segregation Principle (ISP)**
- **Dependency Inversion Principle (DIP)**

These principles were designed for objects, but they can be used in other languages, such as JavaScript. Let us see how SOLID can be used in React.js:

- **Single Responsibility Principle (SRP):** This principle implies that every component should do one thing. When designing your react application, keep this principle in mind because it will make your code more readable and maintainable.
- **Open/Closed Principle (OCP):** This principle says that your code should be extendable without having to break or rewrite a module. This is important so that we can add features to our application without reworking it.
- **Liskov Substitution Principle (LSP):** This principle states that every subclass should be a substitute for its base class. This simply means that if we have a class called Make and it extends to another class called Car, we should be able to extend the class Make without touching the functionality of the Car class. This is possible with class components or when using TypeScript in React.
- **Interface Segregation Principle (ISP):** This principle states that one should only use interfaces that they need. In React, this can be said to be props. Props is very vital in every React application. These props, when passed down from parent to child components, one should only pass down what they need as against everything in the props. We can achieve this by destructuring our props before passing them.
- **Dependency Inversion Principle (DIP):** This principle states that we should hide code implementation and just interact with them by abstraction.

Conclusion

In this chapter, we have seen the different ways to achieve conditional re-rendering in React. We also looked into memoization and its benefits. We

also looked at TypeScript and its benefits. We also looked at file organization and concluded with a separation of concerns and HOCs.

In the upcoming chapter, we will look at testing.

CHAPTER 9

Testing

Introduction

This chapter will cover the testing of react components. We will be using Jest and Cypress for this chapter. Here, we will cover functional testing, integration testing, and end-to-end testing.

Structure

The following topics are to be covered:

- How to test components?
- Using Jest and React testing library to test applications
- How to do a regression Test?
- How to do the e2e test with Cypress?
- How to do an integration test with Jest?

Objectives

In this chapter, we will look at testing react components with Cypress, Jest, and React testing libraries. We will also look at mocking API with Jest and how to do a regression test with Jest. We will also look at the e2e test with Cypress.

How to test components?

Testing is very important in every Web application. Even in React, testing is very important, especially on its components. There are two ways to achieve this, which include the following:

- Functional test
- Rendering test

Functional test

A functional test is a test that is carried out to verify the functions work as they should. This test usually carries out assertiveness based on what should be the supposed outcome of the function being tested.

Rendering test

A rendering test is a test that is carried out to verify that your component renders correctly.

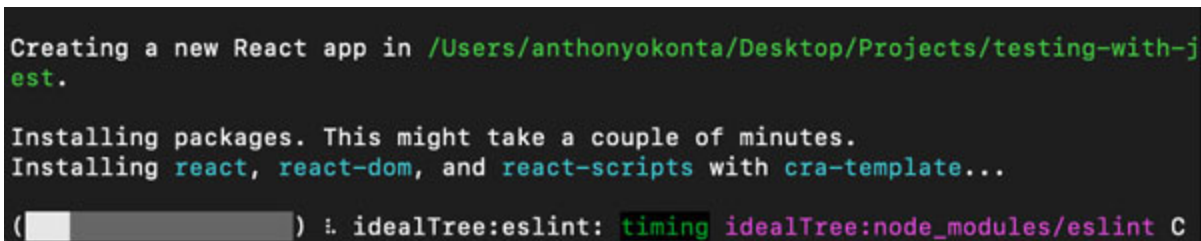
Testing with Jest

Jest is a very popular testing library when it comes to testing react applications. Jest is mostly used for running functional tests, but we can also use it for rendering tests. Let us see how we can use Jest to test some functions in our components.

So, we will create a simple application to demonstrate how we can use Jest. Let us spin up our application. To do this, use the command as follows:

```
npx create-react-app testing-with-jest
```

Our application is being created:

A terminal window with a dark background and light green text. The text shows the process of creating a new React application. It starts with 'Creating a new React app in /Users/anthonyokonta/Desktop/Projects/testing-with-jest.' followed by 'Installing packages. This might take a couple of minutes.' and 'Installing react, react-dom, and react-scripts with cra-template...'. At the bottom, there is a progress bar and the text '(idealTree:eslint: timing idealTree:node_modules/eslint C'.

```
Creating a new React app in /Users/anthonyokonta/Desktop/Projects/testing-with-jest.  
Installing packages. This might take a couple of minutes.  
Installing react, react-dom, and react-scripts with cra-template...  
( [progress bar] ) : idealTree:eslint: timing idealTree:node_modules/eslint C
```

Figure 9.1: Creating our react application

Hurray, our application is ready!

```
Success! Created testing-with-jest at /Users/anthonyokonta/Desktop/Projects/testing-with-jest
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm run build
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm run eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd testing-with-jest
  npm start

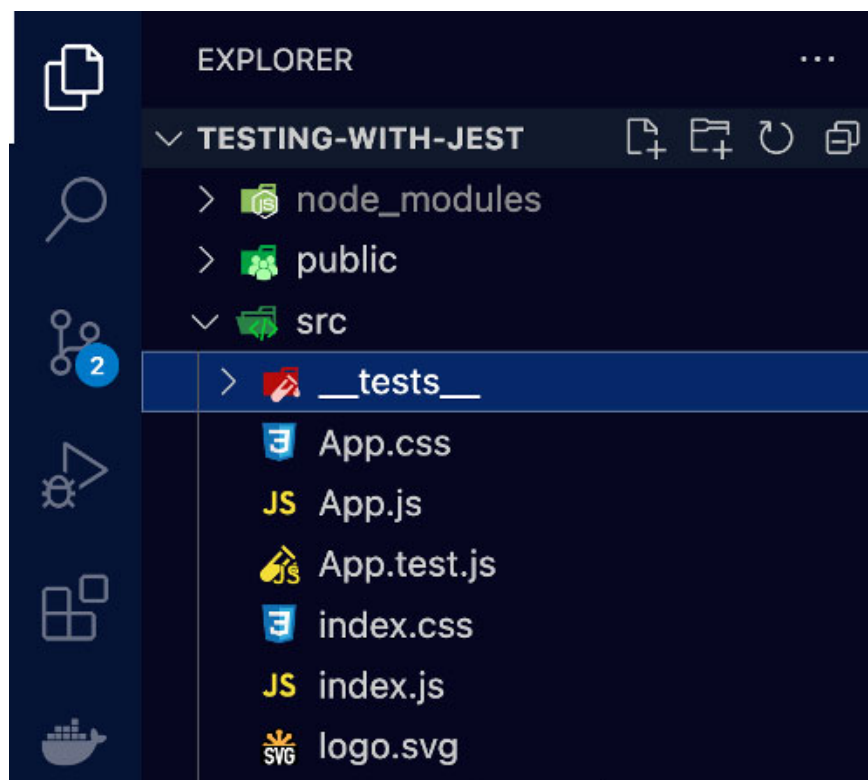
Happy hacking!
anthonyokonta@Anthonys-MacBook-Air Projects %
```

Figure 9.2: Application installed

Before we begin, please install the Jest package with the following command inside the testing-with-jest project:

npm install --save-dev jest or yarn add --dev jest

Now, we have installed our Jest package, and we can now start testing. There are different ways to organize your test files, but for this project, we will put them inside our `__tests__` folder, which we will create:



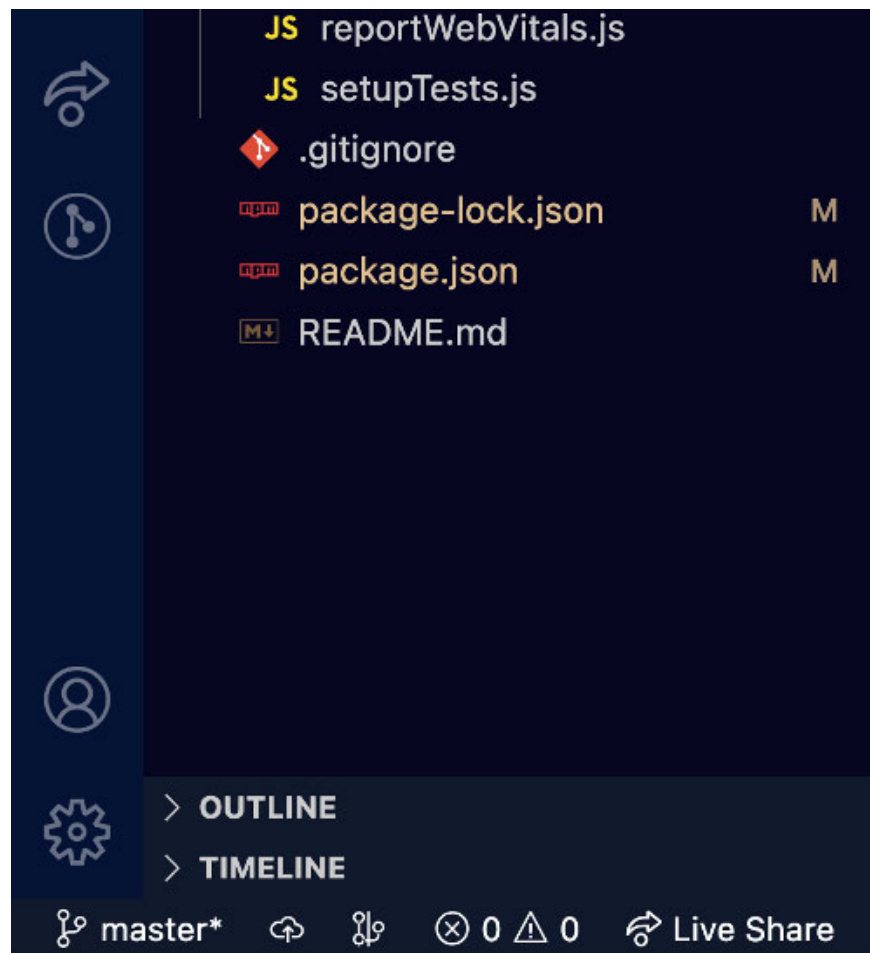


Figure 9.3: File structure where we added tests

After we have created this file, we can now start putting our test files into the folder. The test files usually end with **.spec.js** or **.test.js**. Whichever naming convention you want to do is completely up to you. Do not worry, Jest will find either of them or execute the tests inside of them. For the project, we will use the **.test.js**. So, we are going to create a simple file called **Arithmetic.test.js** and add four tests, which are added, subtracted, divided, and multiplied to test their assertiveness on them passing and failing the test criteria, which we will set for them. Let us look at the following:

1. export function Addition(a,b) {
2. return a+b
3. }
- 4.


```
5. export function Subtraction(a,b) {  
6.   return a-b  
7. }  
8.  
9. export function Multiplication(a,b) {  
10.  return a*b  
11. }  
12.  
13. export function Division(a,b) {  
14.  return a/b  
15. }
```

Here, we just declared our four functions of addition, subtraction, multiplication, and division. This is exported so we can use it in our test file. Please note that this preceding file is put inside the constant folder.

```
1. import { Addition, Division, Multiplication, Subtraction}  
   from '../constants/Arithmetic';  
2.  
3.  
4. test('it should add two numbers', () => {  
5.   expect (Addition(1,2)).toBe(3)  
6. })  
7.  
8. test('it should subtract two numbers', () => {  
9.   expect (Subtraction(2,1)).toBe(1)  
10. })  
11.  
12. test('it should Multiply two numbers', () => {  
13.   expect (Multiplication(1,2)).toBe(2)  
14. })  
15.
```

```
16. test('it should divide two numbers', () => {
17.   expect (Division(2,1)).toBe(2)
18. })
```

The preceding file depicts the tests that we have written for our addition, subtraction, multiplication, and division functions that were written in the previous file.

As you can see from the example, we started with a test for each of the test cases, which takes into two arguments. The first argument is the description of the test and the second one is the function that tests whatever we want to test. It is the code that will be executed and assert if it is true or false. For us to execute this test, we will run the following command:

npm test or yarn test

This is what will be produced on the successful run of this test.

```
1. PASS  src/__tests__/Arithmetic.test.js
2.   ✓ it should add two numbers (2 ms)
3.   ✓ it should subtract two numbers (1 ms)
4.   ✓ it should Multiply two numbers (2 ms)
5.   ✓ it should divide two numbers (2 ms)
6.
7. Test Suites: 1 passed, 1 total
8. Tests:      4 passed, 4 total
```

For the preceding code, it shows that all our tests passed. What if we want to make all of them fail? Let us edit the code and see what happens:

```
1. import { Addition, Division, Multiplication, } from
   '../constants/Arithmetic';
2.
3. test('it should add two numbers', () => {
4.   expect (Addition(1,2)).toBe(2)
5. })
6. test('it should subtract two numbers', () => {
7.   expect (Subtraction(2,1)).toBe(0)
```

```
8. })
9.
10.
11. test('it should Multiply two numbers', () => {
12.   expect (Multiplication(1,2)).toBe(4)
13. })
14.
15. test('it should divide two numbers', () => {
16.   expect (Division(2,1)).toBe(6)
17. })
```

So, we have successfully edited our code **toBe** values for addition, subtraction, multiplication, and division with wrong values to see if our tests will fail. Now, let us run the code with the **npm test** or **yarn test** and see the following output:

```
1. FAIL  src/__tests__/Arithmetic.test.js
2.   ✕ it should add two numbers (6 ms)
3.   ✕ it should subtract two numbers (1 ms)
4.   ✕ it should Multiply two numbers (1 ms)
5.   ✕ it should divide two numbers (2 ms)
6.
7.
8. Test Suites: 1 failed, 1 total
9. Tests:      4 failed, 4 total
10. Snapshots: 0 total
11. Time:      1.201 s
```

Now, you can see in line 75 that all our tests failed because the values passed inside our **toBe** are wrong. The error is descriptive as to why it failed. For example, our division test failed because it should be **2** since we said **2/1=2**, but we passed 6, which is wrong.

So, in testing, it is best you write tests that fail before you write the ones that pass. Let us extend this test example to take care of some issues.

```
1. export function Addition(a, b) {
2.   //check if the data is not null and both parameters are
   'number's
3.   if ((a && b) &&(typeof a==='number' && typeof
   b==='number'))
4.   {
5.     return a+b
6.   }
7. }
8.
9. export function Substraction(a, b) {
10.  //check if the data is not null and both parameters are
    'number's
11.  if ((a && b) &&(typeof a==='number' && typeof
    b==='number')) {
12.    return a - b
13.  }
14. }
15.
16. export function Multiplication(a, b) {
17.  //check if the data is not null and both parameters are
    'number's
18.  if ((a && b) &&(typeof a==='number' &&
    typeof b==='number')) {
19.    return a * b
20.  }
21. }
22.
23. export function Division(a, b) {
24.  //check if the data is not null and both parameters are
    'number's
```

```

25.      if ((a && b) &&(typeof a==='number' &&
    typeof b==='number')) {
26.      return a / b
27.  }
28. }

```

We have modified our function to successfully handle some edge cases. These edge cases will handle null and empty strings and enforce that both parameters passed are numbers and nothing else. We are going to test this by running our test again with **npm test** or **yarn test** and see the result as follows:

```

1. FAIL  src/__tests__/Arithmetic.test.js
2.  ✕ it should add two numbers (3 ms)
3.  ✕ it should subtract two numbers (1 ms)
4.  ✕ it should Multiply two numbers (1 ms)
5.  ✕ it should divide two numbers
6. Test Suites: 1 failed, 1 total
7. Tests:      4 failed, 4 total
8. Snapshots:  0 total
9. Time:       1.33 s
10. Ran all test suites related to changed files.
11.
12. Watch Usage: Press w to show more.

```

Our test failed here because the values were incorrect. How about us passing parameters that are null or not numbers and seeing if our modified function will work?

In the following, we will change the parameters to null and some to object:

```

1. import { Addition, Division, Multiplication, Substraction }
    from '../constants/Arithmetic';
2.
3. test('it should add two numbers', () => {

```

```

4.   expect(Addition(null, {})).toBe(3)
5. })
6.
7.
8. test('it should subtract two numbers', () => {
9.   expect (Substraction(2,1)).toBe(1)
10. })
11.
12. test('it should Multiply two numbers', () => {
13.   expect (Multiplication(1,2)).toBe(2)
14. })
15.
16. test('it should divide two numbers', () => {
17.   expect (Division(2,1)).toBe(2)
18. })

```

And our results after running the **npm test** will be as the following:

```

1. FAIL  src/__tests__/Arithmetic.test.js
2.   ✕ it should add two numbers (3 ms)
3.   ✓ it should subtract two numbers
4.   ✓ it should Multiply two numbers (1 ms)
5.   ✓ it should divide two numbers
6.
7.   • it should add two numbers
8.
9.     expect(received).toBe(expected) // Object.is equality
10.    Expected: 3
11.
12.    Received: undefined
13.      2 |

```

```

14.      3 |   test('it should add two numbers', () => {
15.    > 4 |       expect(Addition(null, {})).toBe(3)
16.      |                                   ^
17.      5 |   })
18.      6 |
19.      7 |
20.
21.                at Object.<anonymous> (src/__tests__/
    Arithmetic.test.js:4:36)
22.
23. Test Suites: 1 failed, 1 total
24. Tests:      1 failed, 3 passed, 4 total
25. Snapshots:  0 total
26. Time:       0.36 s, estimated 1 s
27. Ran all test suites related to changed files.
28.
29. Watch Usage: Press w to show more.

```

From the preceding code, there is only one test case that failed because we passed null and empty objects as the two parameters to the Addition function, and since they are not numbers, they will return undefined.

For more information on how to use Jest, please visit <https://jestjs.io/>.

[React testing library](#)

React testing library is the library used to test components. This library works on DOM nodes. This library actually looks for elements in the node by a data-tested so as to test them. We can also use this library to mock APIs and assert them for their truthfulness. For more on React testing library, visit <https://testing-library.com/>

[Regression test](#)

Regression testing is making sure that everything tested before a change or changes were made is still intact. When something changes in an application, there is a tendency for something to have broken in the application. Regression is here to make sure that everything is still working as before. We can achieve this regression test by using snapshot testing in Jest.

Before we run this example, please install this **npm** package as follows:

```
npm i react-test-renderer
```

Let us look at how this would work with a small example:

```
1. import renderer from 'react-test-renderer';
2.
3.
4. it('renders correctly', () => {
5.   const testTree = renderer
6.     .create(<label>LinkedIn</label>)
7.     .toJSON();
8.   expect(testTree).toMatchSnapshot();
9. });
```

When we run **npm test**, Jest creates a snapshot file of the preceding code. We get something like this as follows after the snapshot file has been created.

```
1. // Jest Snapshot v1, https://goo.gl/fbAQLP
2.
3. exports[`renders correctly 1`] = `
4. <label>
5.   LinkedIn
6. </label>
7. `;
```

When something changes in the tree, Jest will compare the snapshot file that was created from the first code and look for any changes. Let us try to change it from `linkedin.com` to `WhatsApp` and see the following:


```

1. FAIL   src/__tests__/Snapshot.js
2.   ✕ renders correctly (11 ms)
3.
4.   • renders correctly
5.
6.     expect(received).toMatchSnapshot()
7.
8.     Snapshot name: `renders correctly 1`
9.
10.    - Snapshot   - 1
11.    + Received   + 1
12.
13.      <label>
14.    -   LinkedIn
15.    +   WhatsApp
16.      </label>
17.
18.      6 |       .create(<label>WhatsApp</label>)
19.      7 |       .toJSON();
20.    > 8 |     expect(testTree).toMatchSnapshot();
21.        |                               ^
22.      9 |   });
23.
24.                               at      Object.<anonymous>
      (src/__tests__/Snapshot.js:8:20)

```

The test failed because it could not match the snapshot file created when we ran the previous code. To correct this, let us change it back and see the following:

1. Interactive Snapshot Result
2. › 1 snapshot reviewed, 1 snapshot updated

- 3.
4. Watch Usage
5. › Press Enter to return to watch mode.
- 6.
7. Watch Usage: Press w to show more.

We changed it back from WhatsApp to LinkedIn, and we are fine now.

End-to-end test with Cypress

When it comes to end-to-end tests, Cypress leads the way among other frameworks/libraries. To use Cypress, run the following command in your React application.

```
npm install --save-dev cypress
npx cypress open
```

When it is done, add the following code below to the scripts section in your `package.json` file:

```
"e2e-test": "cypress open."
```

Then run `npm run e2e-test` in the terminal and wait. A new window will open, showing you some pre-configured tests you can use.

To learn more about Cypress, please visit <https://docs.cypress.io/guides/component-testing/quickstart-react>

Integration test with Jest

In most of our React applications, we usually integrate with external API to post and fetch data from and to our application. We can use Jest to test the behavior of our API to see the expected and unexpected results. Let us look at a simple example here:

1. `test('it should be greater than length 0', async () => {`
2. `const data = await`
`fetch('https://jsonplaceholder.typicode.com/posts').then(re`
`sponse => response.json()).then(data => {`
3. `return data`
- 4.

```
5.   })  
6.   expect(data.length).toBeGreaterThan(0)  
7. })
```

The preceding code just checks our posts API to see if it returns the data we expect, and we check the length to make sure it is greater than 0. Let us run the **npm test** and see the following result:

```
1. PASS   src/__tests__/API.js  
2.   ✓ it should be greater than length 0 (258 ms)  
3.  
4. Test Suites: 1 passed, 1 total  
5. Tests:      1 passed, 1 total  
6. Snapshots:  0 total  
7. Time:       8.77 s  
8. Ran all test suites matching /API/i.  
9.  
10. Watch Usage: Press w to show more.
```

API mocking with Jest

Mocking is very important in our React application. We use mocking to either spy(view) how our functions were called or use it to test a single function or an entire module. There are the following three types of mock functions in Jest.

- Jest.fn (used to mock a single function)
- Jest.mock (used to mock a whole module)
- Jest.spyOn (used to see how a function was called)

For more information, visit <https://jestjs.io/docs/mock-functions>.

Conclusion

In this chapter, we learned how to test components and types of testing. We also looked at how to use Jest to test functions and integrate with API. We

also looked at React testing library for UI testing. We looked at Cypress, which helps us to do end-to-end testing.

React is constantly updated by the facebook team and because of these changes, it is advisable to always check <https://reactjs.org/docs/getting-started.html> or <https://beta.reactjs.org/> to see the latest updates.

Index

A

API mocking
 with Jest [143](#)
Axios [102-104](#)

B

block
 about [84](#)
 rules [85](#)
Block Element Modifier (BEM) [84](#)
bootstrap
 about [78](#), [79](#)
 cons [79](#)
 pros [79](#)

C

caching data
 about [106](#)
 memoization [106](#)
 state management [108](#)
Camel case [26](#)
class component
 about [33](#)
 ComponentDidMount method [34](#)
 ComponentDidUpdate method [34](#), [35](#)
 example [33](#)
clean code
 business continuity [44](#)
 code readable [44](#)
 implementing [44-52](#)
 writing [44](#)
component
 styling [74](#)
ComponentDidMount method [34](#)
ComponentDidUpdate method [34](#)
component testing
 about [130](#)
 functional test [130](#)
 rendering test [130](#)
conditional rendering
 about [110](#)

- if else operator [114-116](#)
- logical operator [112-114](#)
- memoization [116](#)
- ternary operator [110-112](#)
- useCallback [117](#), [118](#)
- useMemo [116](#), [117](#)
- Consumer method [38](#)
- context API
 - about [38](#)
 - Consumer method [38](#)
 - Provider method [38](#)
- Continuous Delivery (CD) [51](#)
- Continuous Integration (CI) [51](#)
- controlled component
 - versus uncontrolled component [41](#), [42](#)
- custom hooks [70-72](#)
- Cypress
 - used, for end-to-end test [141](#), [142](#)

D

- data
 - fetching, in react application [98](#)
- data sharing components
 - about [36](#)
 - context API [38](#)
 - props [37](#), [38](#)
 - redux library [38](#)
 - useReducer method [39](#)
- Dependency Inversion Principle (DIP) [128](#)
- design patterns [52](#), [53](#)
- Document Object Model (DOM) [30](#)
- do not repeat yourself (DRY) technique [22](#)

E

- ECMA SCRIPT [6](#) (es6) [23](#)
- element
 - about [85](#)
 - example [85](#)
- end-to-end test
 - with Cypress [141](#), [142](#)
- errors
 - checking [53](#)
 - logging, for bug track and fixes [53](#)
- Express.js
 - server-side rendering [95](#)
- external CSS
 - about [75](#), [76](#)
 - cons [76](#)

pros [76](#)

F

fetch

about [98-101](#)

react application [100](#)

file/folder structure [17](#)

file organization

about [118](#)

API [119](#)

assets [119](#)

components [118](#)

constants [119](#)

helpers [119](#)

hooks [119](#)

functional component

about [27-29](#)

useEffect method [32](#), [33](#)

useRef method [30](#), [31](#)

useState method [29](#), [30](#)

functional test [130](#)

G

getServerSideProps [93](#)

GraphQL [105](#)

about [105](#)

versus Rest API [105](#)

H

higher order component [126](#)

hook method [29](#)

I

if else operator [114-116](#)

inline CSS

about [2](#), [3](#), [74](#), [75](#)

cons [75](#)

pros [75](#)

integration test

with Jest [142](#)

Interface Segregation Principle (ISP) [127](#)

J

JavaScript

- benefits [22](#), [23](#)
- using, in React application [22](#)

Jest

- about [130](#)
- testing with [130](#), [131](#), [132](#), [133](#), [134](#), [135](#), [136](#), [137](#), [138](#), [139](#)
- used, for API mocking [143](#)
- used, for integration test [142](#)

L

LESS preprocessor [17](#)

Lint

- rules [19](#)
- using [19](#)

Liskov Substitution Principle (LSP) [127](#)

log errors [20](#)

logical operator [112](#), [113](#), [114](#)

M

maintainable code

- design patterns [52](#), [53](#)
- errors, checking [53](#)
- errors, logging for bug track and fixes [53](#)
- implementing [44-52](#)
- testable codes, writing [53](#)
- writing [52](#)

memoization

- about [106](#), [116](#)
- useCallback [107](#)
- useMemo [106](#), [107](#)

mock functions

- reference link [143](#)
- types [143](#)

modifier [85](#)

modular CSS [16](#)

N

naming convention

- about [17](#), [18](#), [26](#)
- Camel case [26](#)
- Pascal case [27](#)

naming convention type

- Camel case [10](#)
- Pascal case [10](#)

Next.js

- configuration [95](#)
- cons [90](#)
- getServerSideProps [93](#)

- image optimization [95](#)
- maintainability [94](#)
- pre-rendering [90](#)
- pros [88](#), [89](#)
- server-side rendering [95](#)
- static generation [90](#)
- TypeScript [94](#)
- versus React.js [94](#)

Node.js

- server-side rendering [95](#)

O

Open/Closed Principle (OCP) [127](#)

P

Pascal case [21](#), [27](#)

Prettier

- using [19](#)

promise [108](#)

promise states

- fulfilled [108](#)
- pending [108](#)
- rejected [108](#)
- settled [108](#)

props [38](#)

props drilling

- about [5](#)
- access, avoiding [15](#), [16](#)
- ambiguous naming [10](#)
- div, avoiding [14](#)
- HTML tags style, avoiding [13](#), [14](#)
- index key, using in map loop [7](#), [8](#)
- map loop key, avoiding [6](#)
- mutate state, avoiding [11-13](#)
- nested ternary statement [9](#), [10](#)
- pass all props, avoiding [5](#)

Provider method [38](#)

Pull Requests (PR) [51](#)

R

react application

- Axios [102-104](#)
- data, fetching [98](#)
- fetch [98-101](#)
- GraphQL [105](#)
- JavaScript, using [22](#)
- TypeScript, using [23](#)

- react application architect
 - about [16](#)
 - code testing [21](#)
 - file/folder structure [17](#)
 - LESS preprocessor [17](#)
 - Linters, using [19](#)
 - log errors [20](#)
 - naming convention [17](#), [18](#)
 - Prettier, using [19](#)
 - React error boundary [20](#)
 - SASS preprocessor [17](#)
 - styling [16](#), [17](#)
 - try...catch, using [19](#)
 - useCallback, rendering [18](#), [19](#)
 - useMemo, rendering [18](#), [19](#)
- react application, bad practices
 - building [2](#)
 - inline CSS [2](#), [3](#)
 - large component [3-5](#)
 - props drilling [5](#)
- react component structure [35](#), [36](#)
- React error boundary [20](#)
- React hooks
 - about [56](#)
 - rules [56](#)
 - useCallback method [63](#)
 - useContext method [66](#), [68](#)
 - useEffect method [59](#), [60](#)
 - useLayoutEffect method [61](#), [62](#)
 - useMemo method [62](#)
 - useReducer method [57](#), [58](#)
 - useRef method [63-66](#)
 - useState method [56](#), [57](#)
- React.js
 - configuration [95](#)
 - image optimization [95](#)
 - maintainability [94](#)
 - server-side rendering [95](#)
 - TypeScript [94](#)
 - versus Next.js [94](#)
- React testing library
 - about [139](#)
 - reference link [139](#)
- redux library
 - about [38](#)
 - actions [39](#)
 - reducer [39](#)
 - store [39](#)
- regression test [139-141](#)
- rendering test [130](#)

- Rest API
 - about [105](#)
 - versus GraphQL [105](#)
- reusable logic
 - extracting [22](#)
- rops [37](#)

S

- SASS preprocessor [17](#)
- scalable application
 - building, with TypeScript [54](#)
- semantic HTML [14](#)
- separation of concerns [119-126](#)
- server-side rendering
 - with Express.js [95](#)
 - with Node.js [95](#)
- server-side rendering (SSR)
 - about [88](#)
 - using [88](#)
- Single Responsibility Principle (SRP) [127](#)
- solid principles
 - about [127](#)
 - Dependency Inversion Principle (DIP) [128](#)
 - Interface Segregation Principle (ISP) [127](#)
 - Liskov Substitution Principle (LSP) [127](#)
 - Open/Closed Principle (OCP) [127](#)
 - Single Responsibility Principle (SRP) [127](#)
- state management [108](#)
- static generation
 - about [90](#)
 - static page with data [91](#)
 - static page without data [91](#)
- static page with data
 - about [91](#)
 - external data content [91](#)
 - external data path [92](#)
- static page without data [91](#)
- style conventions
 - Block Element Modifier (BEM) [84](#)
 - writing [84](#)
- styled component [16](#)
- styled components
 - about [79-83](#)
 - cons [84](#)
 - pros [83](#), [84](#)
- styling [16](#), [17](#)
- styling techniques
 - modular CSS [16](#)
 - styled component [16](#)

- styling types
 - about [74](#)
 - bootstrap [78](#), [79](#)
 - external CSS [75](#), [76](#)
 - inline CSS [74](#), [75](#)
 - styled components [79](#), [81](#), [83](#)
 - Synthetically Awesome Style Sheet (Sass) [76](#), [78](#)
 - Tailwind CSS [84](#)
- Synthetically Awesome Style Sheet (Sass)
 - about [76-78](#)
 - cons [78](#)
 - pros [78](#)

T

- Tailwind CSS [84](#)
- ternary operator [110-112](#)
- testable codes
 - writing [53](#)
- total blocking time (TBT) [89](#)
- try catch
 - using [19](#)
- TypeScript
 - about [118](#)
 - advantages [118](#)
 - benefits [23](#)
 - using, in react application [23](#)
 - using, to build scalable application [54](#)

U

- uncontrolled component
 - versus controlled component [41](#), [42](#)
- useCallback
 - about [117](#), [118](#)
 - rendering with [18](#), [19](#)
- useCallback method [63](#)
- useContext method [66](#), [68](#)
- useEffect method
 - about [32](#), [33](#), [59](#), [60](#)
 - parameters [32](#)
- useLayoutEffect method [61](#), [62](#)
- useMemo
 - about [106](#), [107](#), [116](#), [117](#)
 - rendering with [18](#), [19](#)
- useMemo method [62](#)
- useReducer method [39](#), [57](#), [58](#)
- useRef method [30](#), [31](#), [63-66](#)
- useState method [29](#), [30](#), [56](#), [57](#)