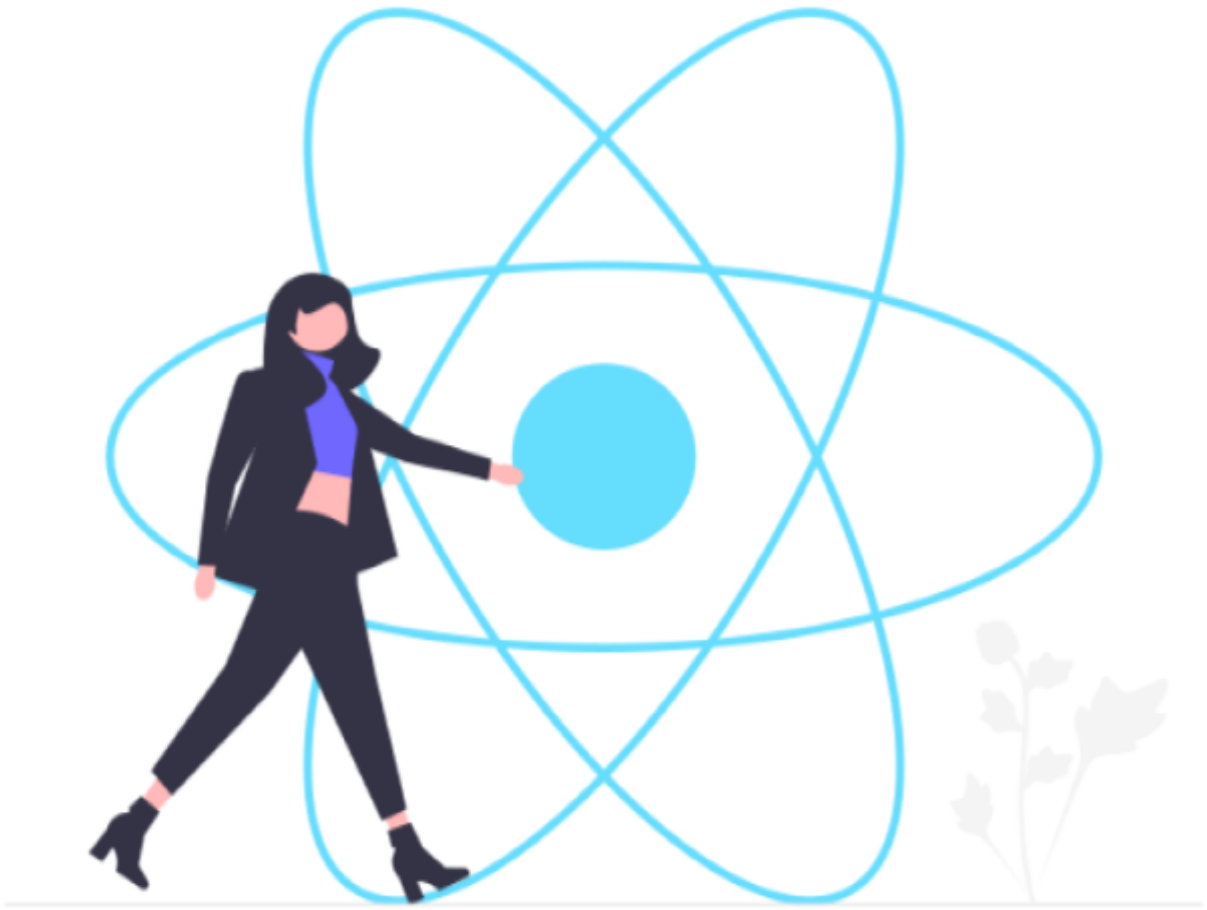


Frontend Web Development with ReactJS

Frontend Web Development with ReactJS

Learner's Guide

APTECH LIMITED
Contact E-mail: ov-support@onlinevarsity.com

First Edition - 2023

Onlinevarsity

Preface

ReactJS is a powerful and popular JavaScript library for building user interfaces. Its flexibility and scalability make it an ideal choice for modern frontend Web development. The guide aims to provide a comprehensive guide to using ReactJS to build beautiful and functional Web applications.

This book is the result of a concentrated effort of the Design Team, which is continuously striving to bring you the best and the latest in Information Technology. The process of design has been a part of the ISO 9001 certification for Aptech-IT Division, Education Support Services. As part of Aptech's quality drive, this team does intensive research and curriculum enrichment to keep it in line with industry trends.

We will be glad to receive your suggestions.

Design Team

MANY COURSES ONE PLATFORM

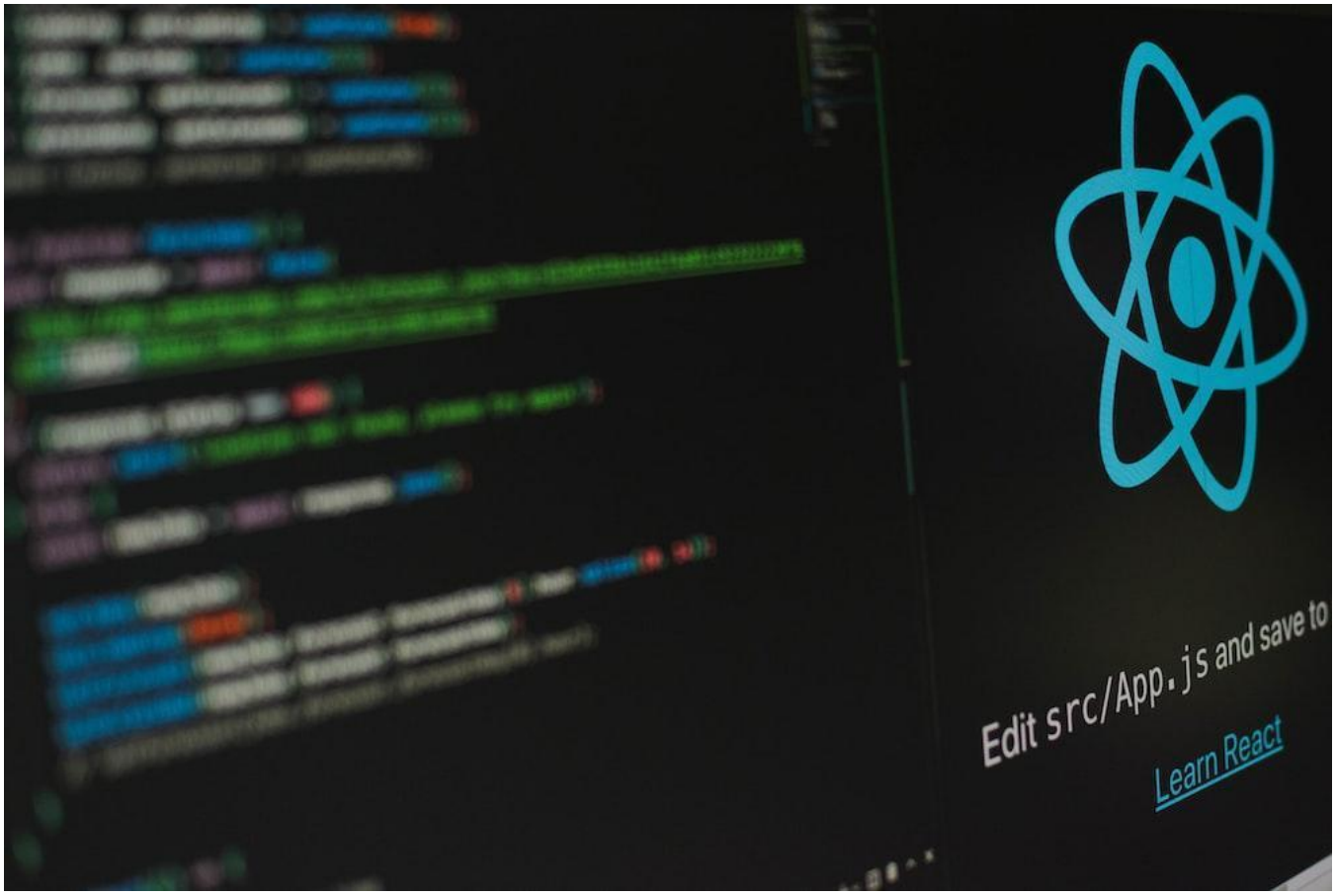
Onlinevarsity

Onlinevarsity App

for Android devices

Download from Google Play Store

Sessions



Session 1

Introduction to ReactJS

Session Overview

This session provides information about ReactJS. It also provides an overview of the ReactJS ecosystem, outlines the core concepts of ReactJS, and setting up a ReactJS development environment.

Objectives

In this session, students will learn to:

- Explain ReactJS
- Explain the ReactJS ecosystem
- Explain the core concepts of ReactJS
- Illustrate the setup of the ReactJS development environment
- Illustrate installation of the create-react-app library

1.1 What is ReactJS?

ReactJS is an open-source front-end JavaScript library, which is used for building component-based User Interfaces (UI). Also known as ReactJS.js, it is used to develop single-page Web and mobile applications or server-rendered applications with frameworks like Next.js.

Meta (formerly Facebook), assisted by a community of individual developers maintain ReactJS.

Note: ReactJS is only used to build and render user interface components to the Document Object Model (DOM). For creating ReactJS applications, developers would also require other libraries to cater to client-side functionalities and routing.

1.2 Why use ReactJS?

ReactJS is one of the most popular libraries because it helps in creating:

- Declarative views
- Component-based interactive UIs

Learning ReactJS helps developers in writing codes and creative UIs from anywhere easily.

Declarative views

ReactJS helps in designing simple declarative views. Advantages of declarative views are:

- Predictable code
- Easy to debug
- Helps in designing simple views for each state in an application
- As data changes, it facilitates updating and rendering the right components

Component-Based design

ReactJS facilitates building component-based interactive user interfaces. Advantages of component-based designing are:

- Components are capable of managing their own states
- Developers can design interactive and complex UIs using multiple components
- The logic uses JavaScript instead of templates, which helps in passing rich data through the application
- It allows in keeping the states out of the DOM

1.3 Overview of ReactJS Ecosystem

The ReactJS ecosystem has the following components:

- Babel
- Webpack
- Routing
- Styling
- State (Redux/ Context)

Table 1.1 provides information about the components of the ReactJS ecosystem.

Component	Description
Babel	Babel is a JavaScript compiler that converts JavaScript code from one format to another based on the specified configurations.
Webpack	Webpack, a module bundler, scans the codebase and bundles all the JavaScript modules and other dependent modules into a single file. This single file is bundled intelligently so that the browser can easily understand and use it.
Routing	ReactJS Router renders the specific ReactJS components depending on the Uniform Resource Locator (URL).
Styling	Users can style ReactJS applications through: ● Traditional styling ● ReactJS styling
	Traditional Styling ● Includes all the styling details in the index.css file. ● Users can include classes to define cascading. ● Users can also use Syntactically Awesome Style Sheets (SASS) or Cascading Style Sheet (CSS) pre-processors.
	ReactJS Styling ● This styling uses component model, wherein the styling is encapsulated within the component. ● Styled Components is the most popular CSS in the ReactJS ecosystem.
State (Redux/ Context)	ReactJS designs interactive and complex UI applications using multiple components. Each of these components are designed to hold information about the UI and also manage their own states. However, at times, in real-life scenarios, the component state must be shared with other related components. Sharing becomes complicated for larger and complex applications. Therefore, the following help in efficiently handling the sharing of component states: ● Redux state management library. ● Context API.
	Redux state management library Redux facilitates collating all the component state details in a single location. This single location is referred to as a 'Store'. Rules are created to manage the states within the 'Store'. The states in the 'Store' change through 'Actions'. **Note**: Redux is used within the ReactJS library and also with other view libraries.
	Context API Context API manages the state of a component, without depending on external libraries. The official ReactJS docs at Context – React (reactjs.org) defines Context as follows: "*Context provides a way to pass data through the component tree without having to pass props down manually at every level.*"

	Note: Context API was developed by the ReactJS team to avoid depending on external libraries such as Redux.

Table 1.1: Components of the ReactJS Ecosystem

1.4 Core Concepts of ReactJS

The core concepts of ReactJS include:

- Virtual DOM
- Components
- Properties and States
- JavaScript XML
- Lifecycle Methods

1.4.1 Virtual DOM

To understand virtual DOM, it's important to understand DOM and its functions.

What is Document Object Model (DOM)

When users request access to a Web page through a browser, the browser receives an HTML document from the server. The browser creates a structured and tree-based, hierarchical representation of the HTML document. This structured representation is referred to as Document Object Model (DOM).

What is the function of DOM?

JavaScript and other scripting languages cannot easily understand a HTML document. However, it can easily access and understand the structured format in DOM. Hence, it becomes easier to interact with the content on the Webpage through DOM because it acts as an interface to access the Web page.
Example: Developers use DOM APIs to:

- Add or remove elements.
- Modify appearance.
- Perform user actions on the Web elements.

What is Virtual DOM?

Virtual DOM is a copy, clone, or a virtual image of the DOM. ReactJS uses a virtual DOM. For every object in the DOM, there is an equivalent object in the virtual DOM.

What are the advantages of a virtual DOM?

As virtual DOM is a copy of the DOM, it is faster to access the virtual DOM and work or manipulate the objects in the virtual DOM. For example: If there is a change in the state of any element or data in an application, the change gets updated in the virtual DOM first.

How does Virtual DOM work faster?

Every element in the application is part of the virtual DOM, which is a structured tree. So, whenever a new element is added to an application, it is represented as a node of the structured tree.

If there is a change in the state of this element, then a new virtual DOM is created. This new virtual DOM is compared with the previous DOM to track the changes. The original DOM gets updated appropriately. When the changes are rendered on the Web page because the updated part only is rendered, it is updated faster.

How does Virtual DOM help ReactJS?

ReactJS is component-based and therefore everything is considered as a component. For example: Functional component, class component, and so on. ReactJS manages two Virtual DOM trees:

- Updated Virtual DOM.
- Pre-Update version of the Virtual DOM.

Each component in an application contains a state and every time the state changes, the ReactJS optimizes the process of updating and re-rendering the UI through the following steps:

1. It updates the Virtual DOM tree.
2. It compares the two versions (Updated virtual DOM and pre-updated version of the virtual DOM) to figure out the changes in the components.
3. ReactJS updates only the relevant changes in the original DOM, as shown in Figure 1.1.
4. ReactJS also uses batch updates to update the real DOM instead of updating after every single change.

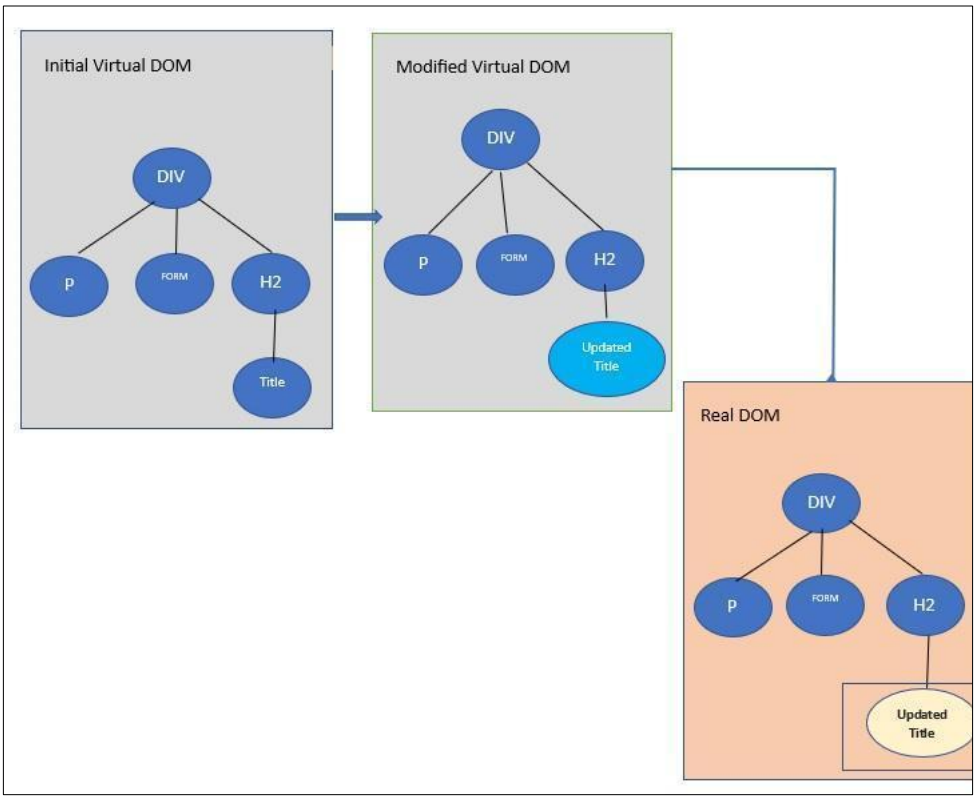


Figure 1.1: Reconciliation Process

Important Terms

Diffing - The process of comparing the pre-updated version of the virtual DOM and the updated version.

Reconciliation - The process of updating changes to the real DOM.

1.4.2 Components

Components are the core building blocks of ReactJS because every application is a group of components, bundled together.

For example: Each part of the UI is a combination of multiple components. Therefore, developers can work on each component independently and then bundle them together as the final User Interface.

In technical terminology, each component in ReactJS returns a part of JSX code, which indicates what must be rendered on the screen.

Table 1.2 lists the types of components in ReactJS.

Component Type	Description
Functional	Functional components are JavaScript functions. These functions can receive or not receive data as parameters. To create a functional component in ReactJS, developers must write a JavaScript function. Functional components are not aware of other components in the program.
Class	Class components work together. Therefore, users can pass data from one class component to the other. JavaScript ES6 classes are used to create class-based components in ReactJS.

Table 1.2: ReactJS Component Types

1.4.3 Properties and States

Properties and States play an important role in ReactJS.

Properties - Used to transfer data between ReactJS components. Properties are also referred to as props.

States - A built-in object that facilitates components to create and manage their data.

Notes:

In ReactJS, data flow is unidirectional, wherein data flow is from parent to child. Components cannot transfer data with the state. Components have the capability of creating and managing data internally.

Table 1.3 lists the difference between Properties (props) and States.

Properties (Props)	States
Components receive data.	Components create and manage their own data.
Used to transfer or pass data.	Used to manage data.
Data is read-only and the component receiving it cannot modify or change the received data.	• The data created by the component can modify it. However, it is private to the component and other components cannot access it. • The `setState( )` method is used to modify a state.

Data flow is unidirectional, wherein props can be passed from parent to child only.	NA

Table 1.3: Difference between Props and States

1.4.4 JavaScript XML

The JavaScript Extension (JSX) is a ReactJS extension, which allows developers to write JavaScript codes with an HTML/XML-like syntax. So, like HTML, JSX tags have tag names, attributes, and children.

Advantages of using JSX

The advantages of JSX include following:

- The co-existence of JavaScript/ ReactJS code.
- Helping pre-processors (like Babel) to read and convert the syntax to compatible standard JavaScript objects.
- Writing HTML/XML-like structures in the same file where JavaScript code exists, which are later converted by preprocessors into the actual JavaScript code. Example: DOM-like tree structures.
- Faster processing when compared to the regular JavaScript because optimization occurs while translating the code to JavaScript.
- ReactJS using components that contain the markup and logic, without separating them as independent files.
- Creating templates easily.
- Finding errors during compiling (Type-safe).

1.4.5 Lifecycle Methods

The ReactJS Web applications are a group of independent components that run according to the interactions made with them. Each ReactJS component has a lifecycle of its own.

The lifecycle of a component is a series of methods that are invoked in different stages of the component's existence. The lifecycle of a ReactJS component has four stages as shown in Table 1.4 and Figure 1.2.

Stage	Description
Initialization	In this stage, the component is created and initialized with default props and state. This is done when the component is first mounted onto the DOM.
Mounting	In this stage, the component is mounted onto the DOM, and the `componentDidMount()` method is called. This is the point where the component has access to the DOM and can manipulate it.
Updating	Here, the state of a component is updated, and the application is re-rendered.
Unmounting	In this stage, the component is removed from the DOM. This happens when the component is no longer needed, or the parent component is removed from the DOM.

Table 1.4: Stages of the ReactJS Component Lifecycle

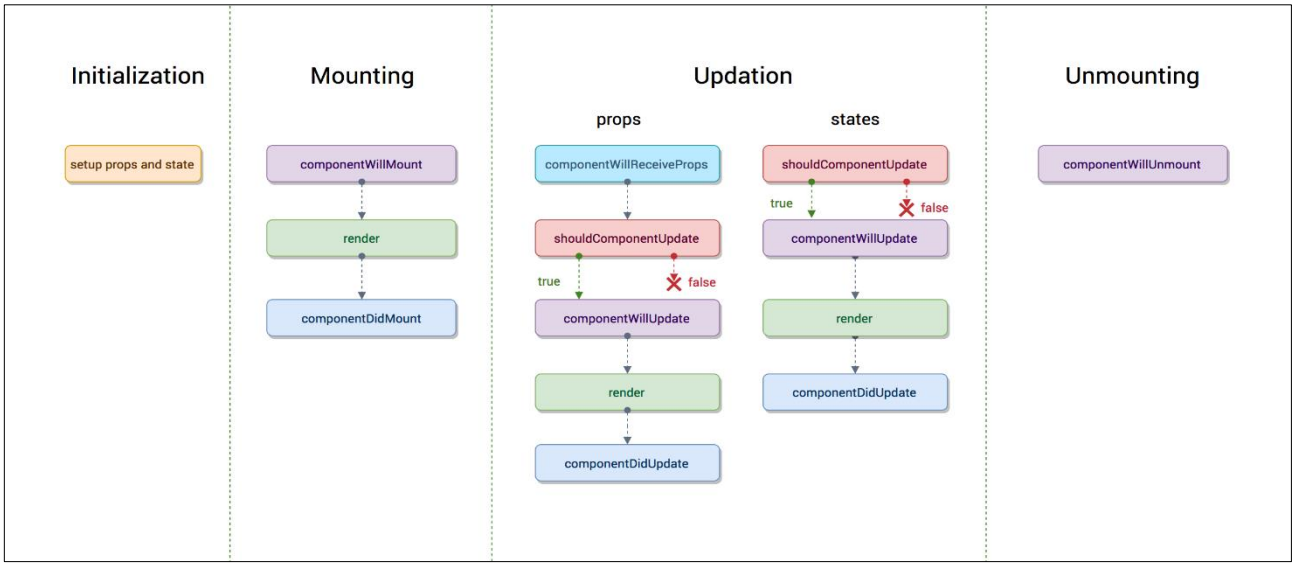


Figure 1.2: Stages: ReactJS Component Lifecycle

1.5 Setting Up the ReactJS Development Environment

To set up the ReactJS development environment, users must install the following:

- JavaScript Runtime - Install NodeJS as a JavaScript Runtime is required in a ReactJS development environment.
- Integrated Development Environment (IDE) – Install Visual Studio Code as an IDE. There are other available IDEs such as Sublime Text, Visual Studio Code, WebStorm, and so on.
- Create-react-app library – Install the create-react-app library..

1.5.1 Installing NodeJS

To install NodeJS, the steps are as follows:

1. Download the **Node.js 18.14.2 '.msi'** installer from the following link: https://nodejs.org/en/download/
2. Open the installer. A dialog box as shown in Figure 1.3 appears.

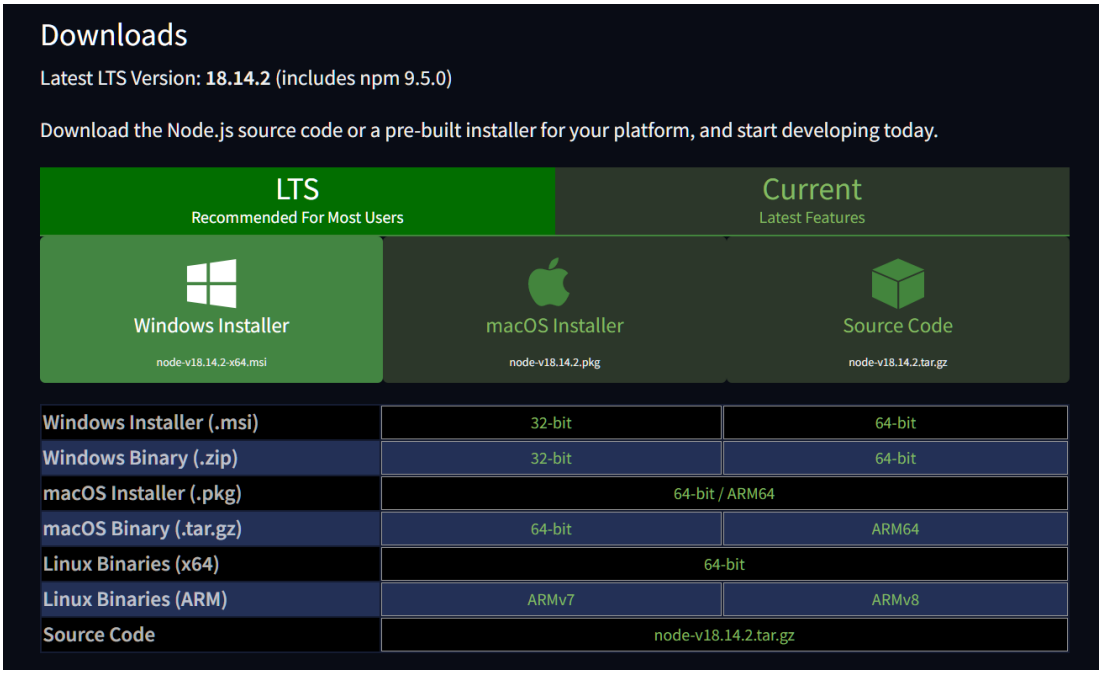


Figure 1.3: Installing NodeJS

3. Run the installer and click **Next** as shown in Figure 1.4 to accept the end-user license agreement.

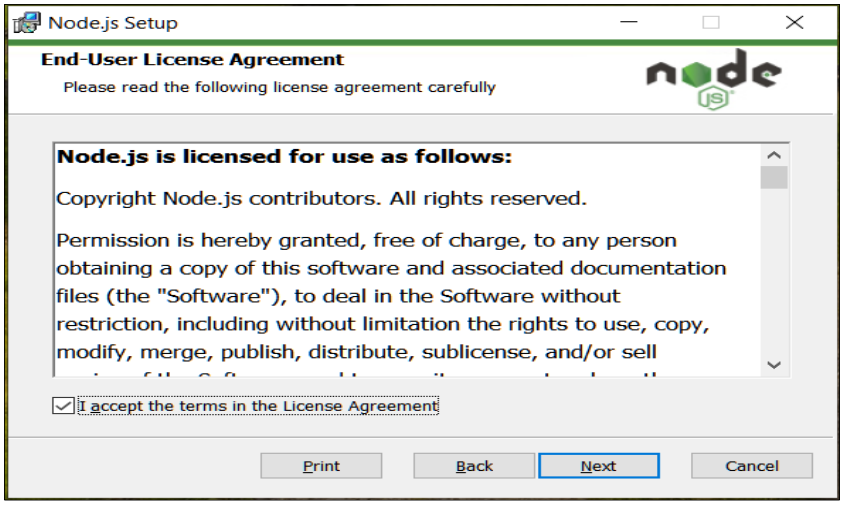


Figure 1.4: Accepting the License Agreement

4. Change the installation directory as shown in Figure 1.5, if required, and click **Next**.
Note: It is advisable to retain the default directory.

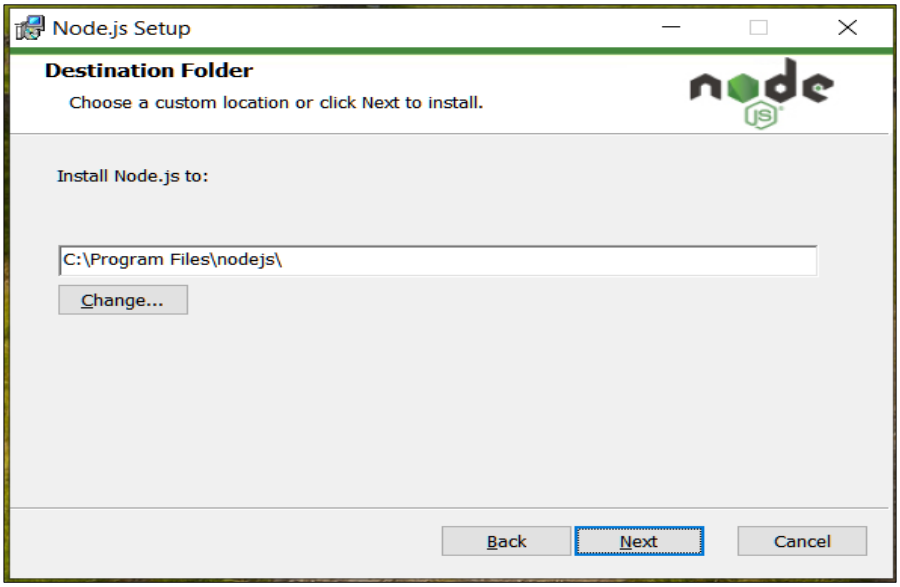


Figure 1.5: Destination Folder

5. Retain the custom setup options as shown in Figure 1.6 and click **Next**.

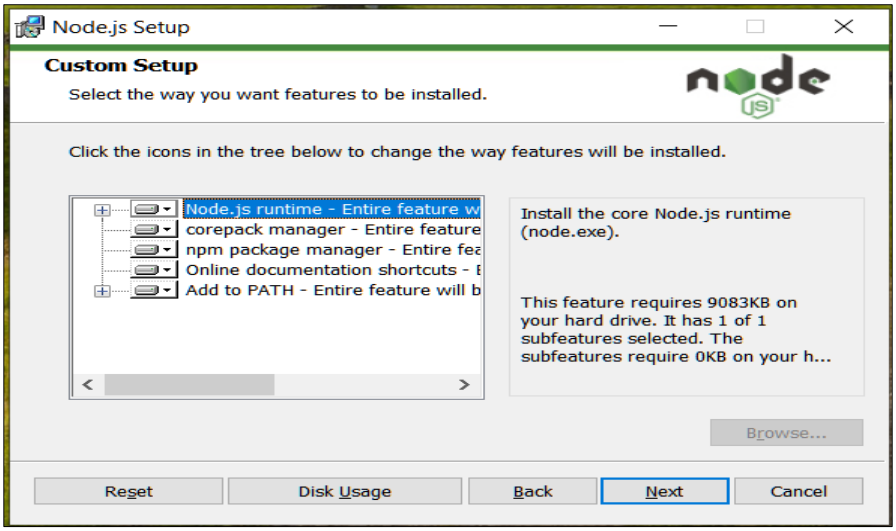


Figure 1.6: Custom Setup

6. Select the **Automatically install the necessary tools** option as shown in Figure 1.7, only if you want to install the tools. It is optional to install these tools. If you do not want to install the tools, do not select the option. Click **Next**.

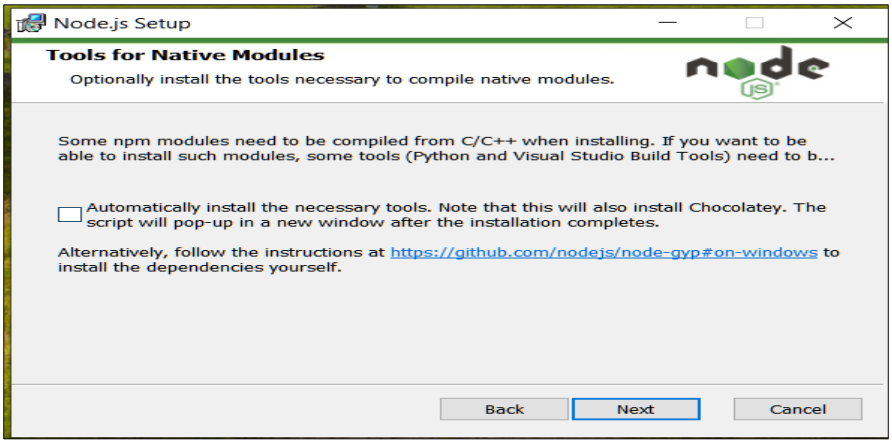


Figure 1.7: Optional Tools for Native Modules

7. Click **Next** on the Setup wizard as shown in Figure 1.8 and follow the onscreen instructions to install NodeJS. The Installation success dialog box appears after successful installation.

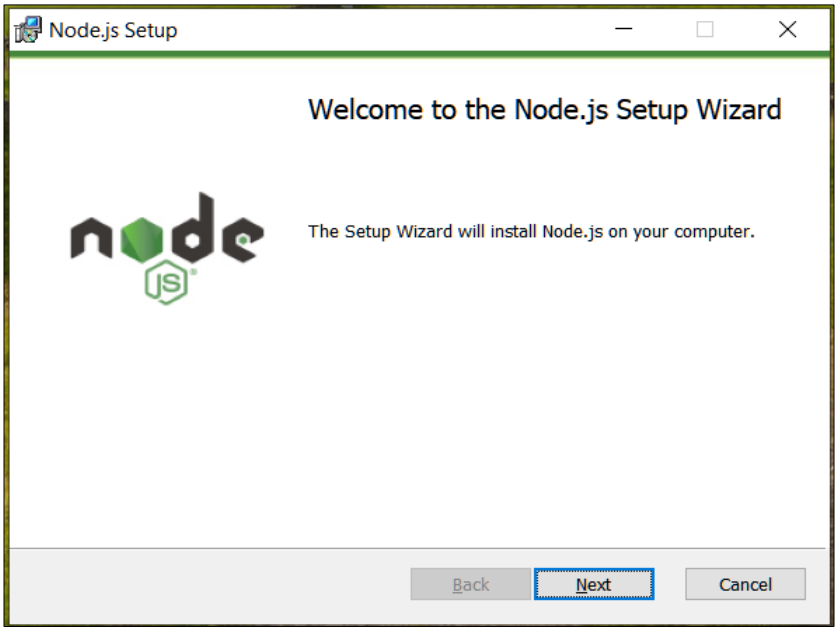


Figure 1.8: Installation Wizard

1.5.2 Installing the Visual Studio Code IDE

Using an IDE like Visual Studio Code to write ReactJS code helps in the following:

- Reducing the development time.
- Auto-generation of base codes such as definitions of class components, functional components, constructors, and so on.

To install the Visual Studio Code IDE, the steps are as follows:

1. Download Visual Studio Code from this link: https://code.visualstudio.com/download. Click the appropriate option based on the operating system used. Figure 1.9 shows different operating systems.

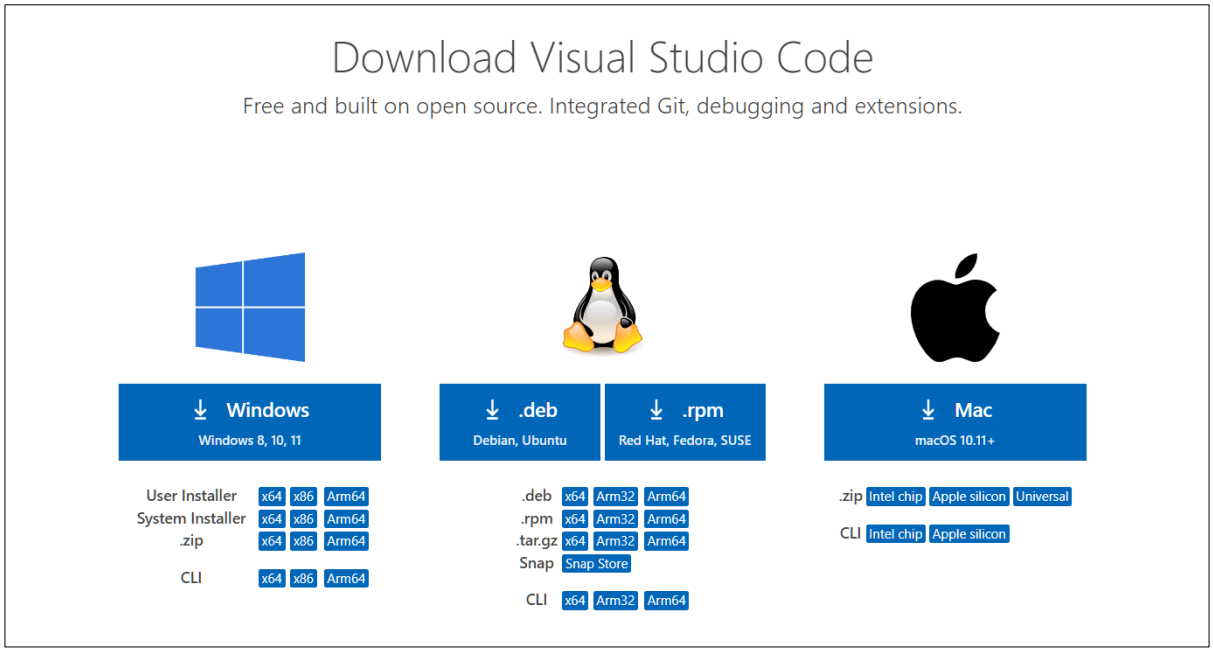


Figure 1.9: Downloading Visual Studio Code

2. Run the installer after downloading. Select **I accept the agreement** click **Next** to accept the License agreement as shown in Figure 1.10.

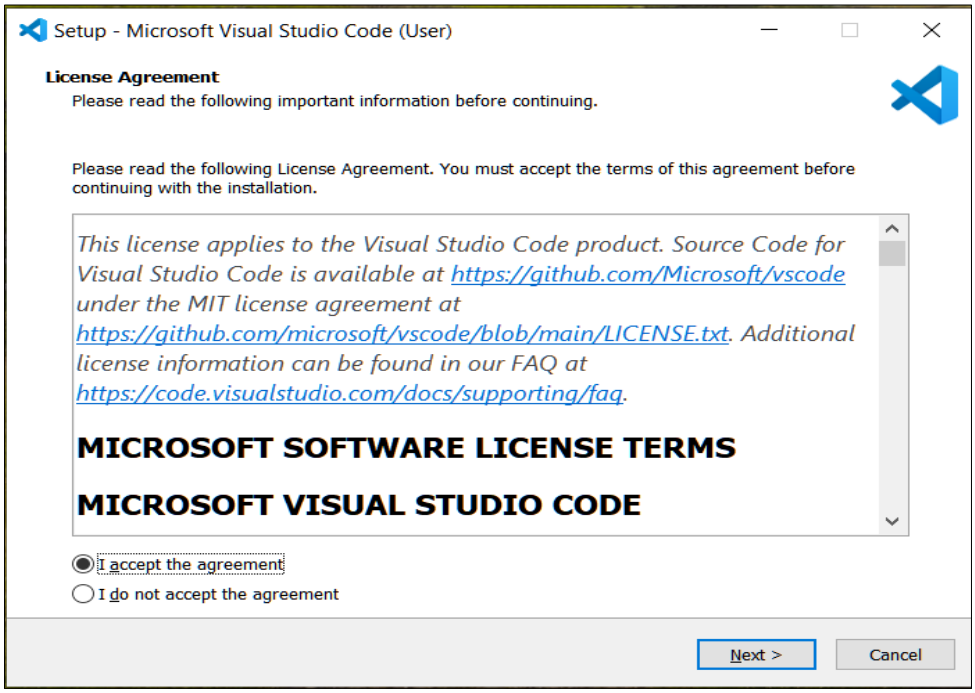


Figure 1.10: License Agreement

3. Select the required options from the Select Additional Tasks dialog box and click **Next**.
Note: Do not select the Add to path option until you know to how to add the PATH to environment variables manually.

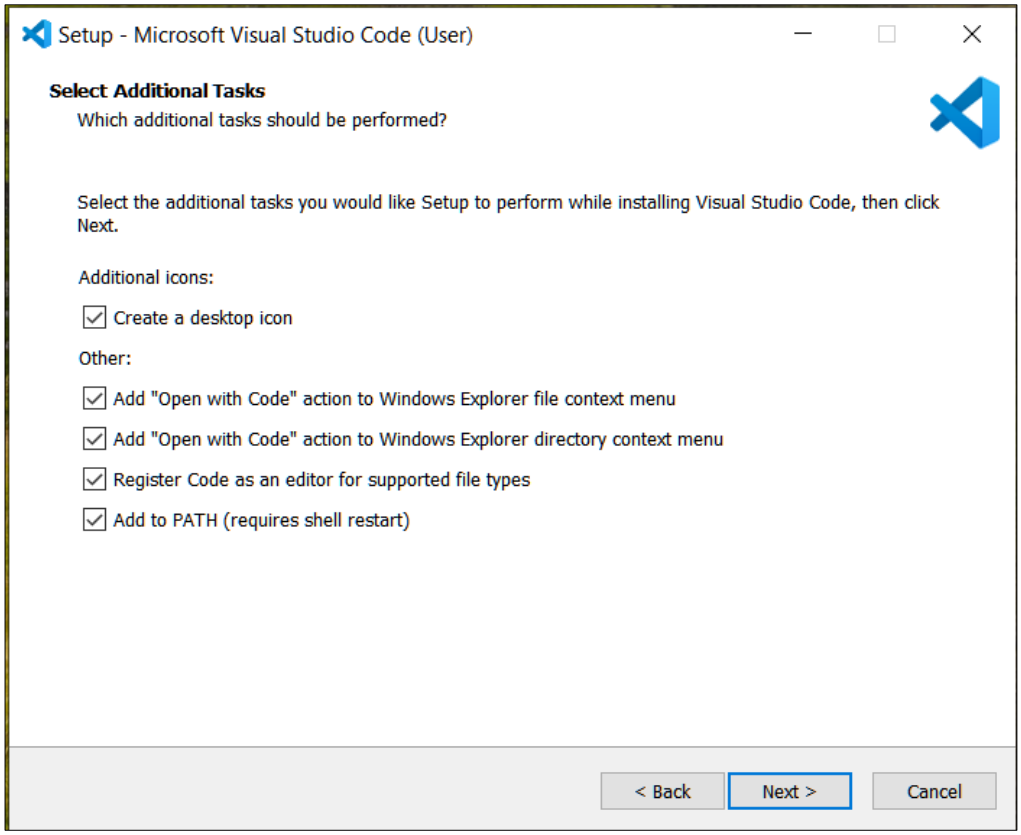


Figure 1.11: Selecting Additional Tasks

4. The Ready to Install dialog box appears as shown in Figure 1.12. Click **Install** to begin installing the Visual Studio Code.

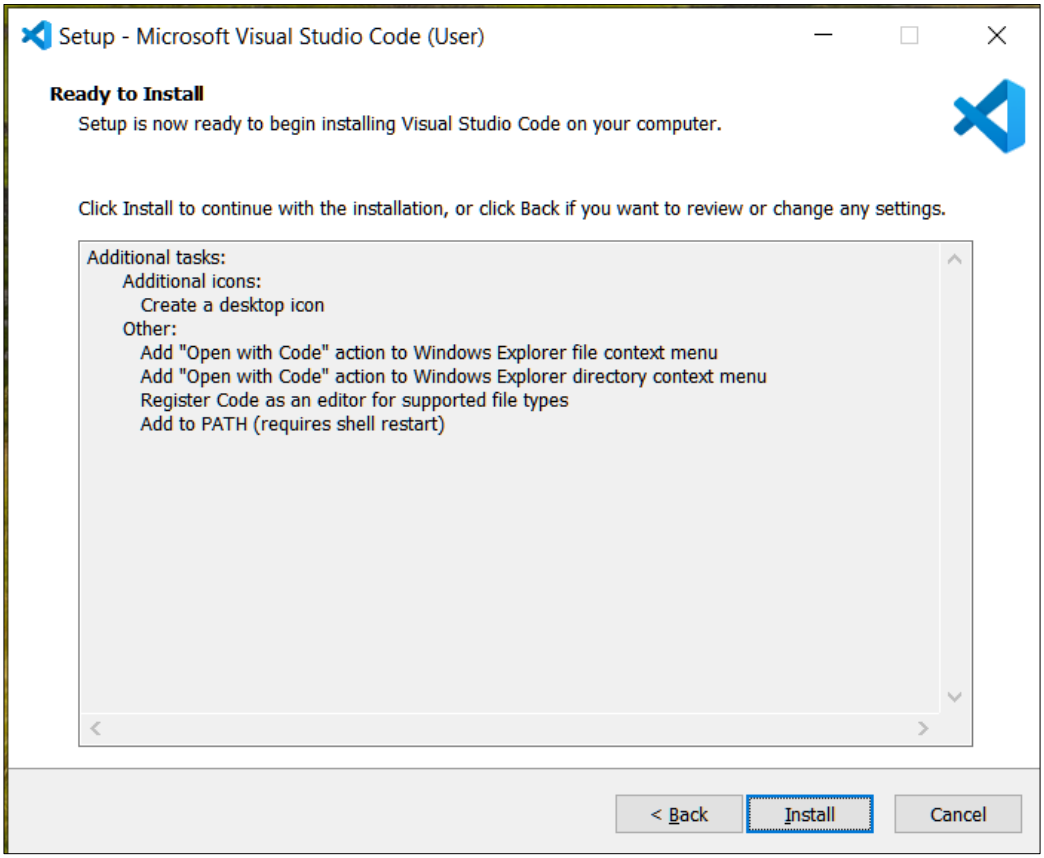


Figure 1.12: Installing Visual Studio Code

1.5.3 Installing the create-ReactJS-app Library

Figure 1.13 provides information about the command used to install the **create-ReactJS-app** library and the version number.

Figure 1.13: Version and Command Details

To install the create-react-app Library, the steps are as follows:

1. Open the command prompt.
2. Run the global command : `npm i -g create-react-app`

   Running this command installs the utility with global access to every user, who is part of the system. It facilitates creating ReactJS applications easily. It also ensures that the latest version of ReactJS is installed.

   Note: The latest version of ReactJS is ReactJS 18.x or above. This session uses 18.2.0.

1.6 Creating a Sample ReactJS App

To create a ReactJS App, the steps are as follows:

1. Right-click on the folder you want to work with and click **Open with Code** as shown in Figure 1.14.

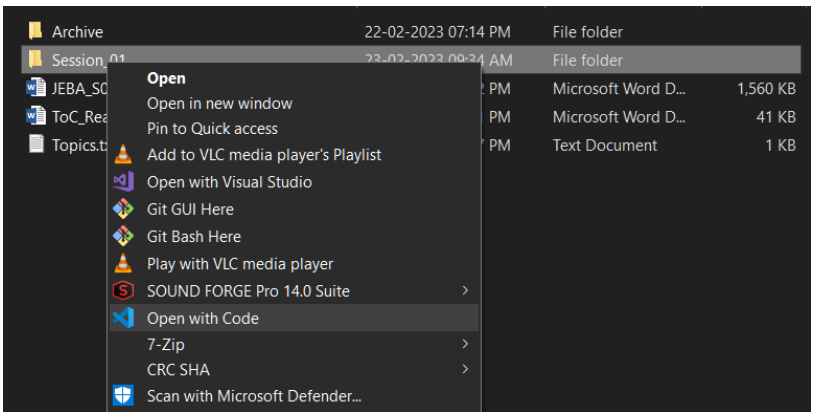


Figure 1.14: Folder to Create ReactJS App

2. Create a new folder named **Code** as shown in Figure 1.15 (or with a name of choice). This folder is to store all the practice-related codes.

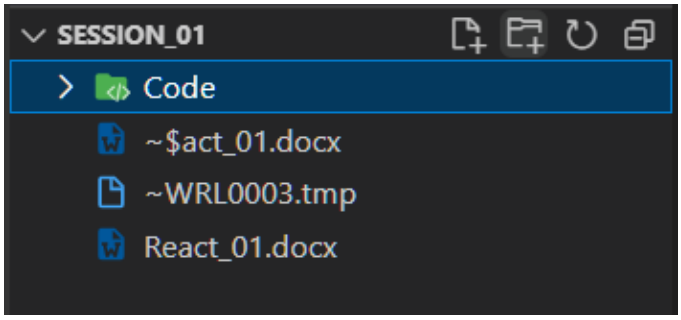


Figure 1.15: Folder for Storing Code Samples

3. Right-click the folder **Code** and select **Open in Integrated Terminal** as shown in Figure 1.16.

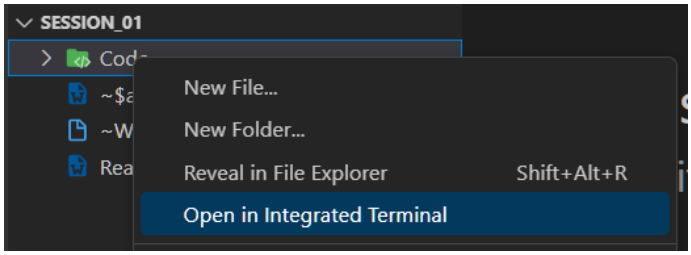


Figure 1.16: Opening the Integrated Terminal

4. A terminal window appears as shown in Figure 1.17.
   a. Run the command: `npx create-react-app hello-world`
   b. Type `y` for yes and press **Enter.**

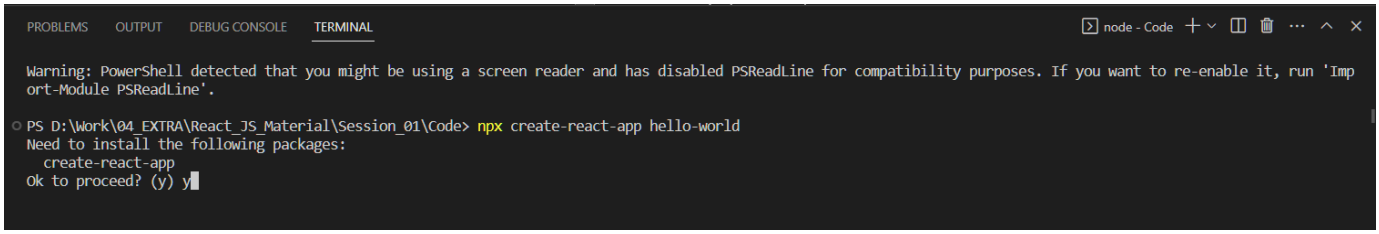


Figure 1.17: Terminal to Run Commands

5. After successful installation of the ReactJS app, type the following as shown in Figure 1.18:

```
cd hello-world
<Press Enter>
npm start
<Press Enter>
```

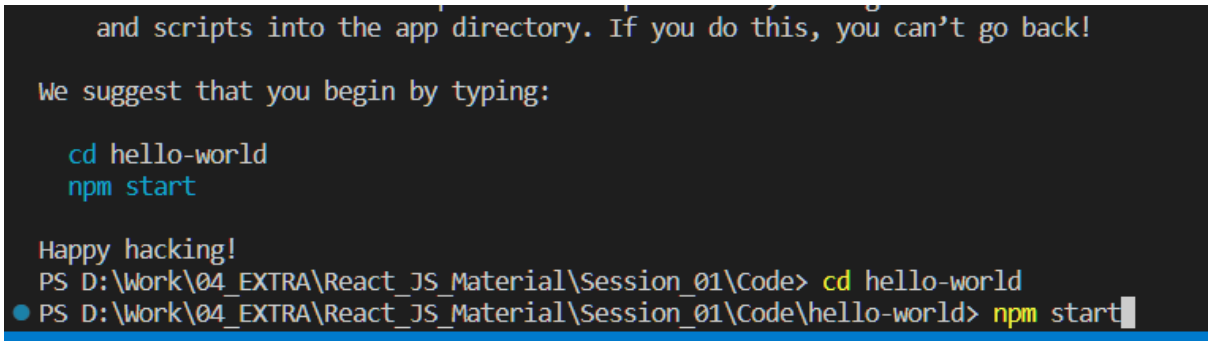


Figure 1.18: Commands for Starting the Server and Creating the ReactJS App

6. The server starts and the first ReactJS App is created. The ReactJS Welcome page appears as shown in Figure 1.19.

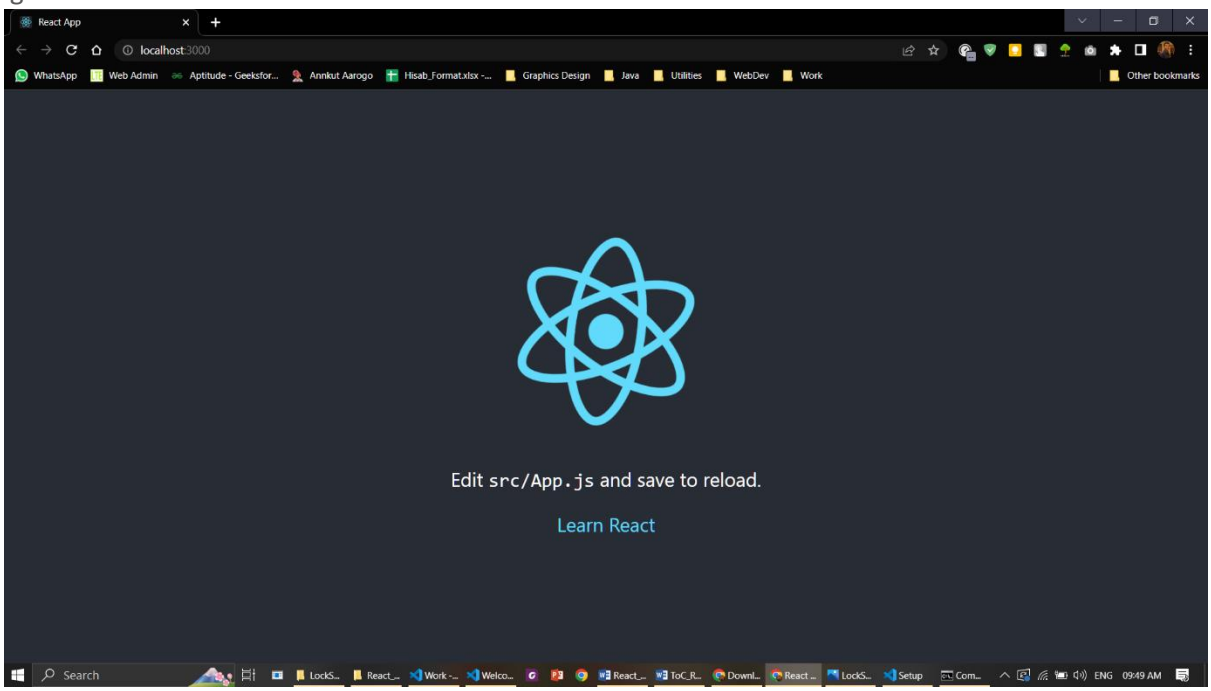


Figure 1.19: Welcome Page of ReactJS Application

7. To stop the ReactJS application, press **Ctrl+c** in the terminal.

1.7 Summary

- ✓ ReactJS is an open-source front-end JavaScript library, which is used for building component-based User Interfaces (UI).
- ✓ ReactJS helps in designing simple declarative views and facilitates building component-based interactive user interfaces.
- ✓ The ReactJS ecosystem has the following components:
  - Babel
  - Webpack
  - Routing
  - Styling
  - State (Redux/Context)
- ✓ Virtual DOM is a copy, clone, or a virtual image of the DOM. ReactJS uses a virtual DOM. For every object in the DOM, there is an equivalent object in the virtual DOM.
- ✓ The process of comparing the pre-updated version of the virtual DOM and the updated version is known as Diffing.
- ✓ To set up the ReactJS development environment, users must install JavaScript Runtime, Integrated Development Environment (IDE), and the create-react-app library.

1.8 Check Your Knowledge

1. Which of these are ReactJS components?

A	Functional
B	Class
C	A and B
D	None of these

2. To setup the ReactJS development environment, which library must the user install?

A	JavaScript
B	IDE
C	`create-react-app`
D	None of these

3. What is Babel?

A	Used to transfer data between ReactJS components
B	A built-in object
C	A JavaScript compiler
D	None of these

4. What is Webpack?

A	Used to transfer data between ReactJS components
B	Router
C	JavaScript compiler
D	Module bundler

5. _____ facilitates collating all the component state details in a single location.

A	Redux
B	Babel
C	Webpack
D	Module bundler

6. _____ is a copy of the DOM.

A	Virtual DOM
B	JavaScript
C	ReactJS
D	Context API

Answers

1	C
2	C
3	C
4	D
5	A
6	A

Try It Yourself

Set up a ReactJS development environment and create a simple ReactJS component. Ensure following tools are installed:

- Node.js
- NPM
- Code Editor

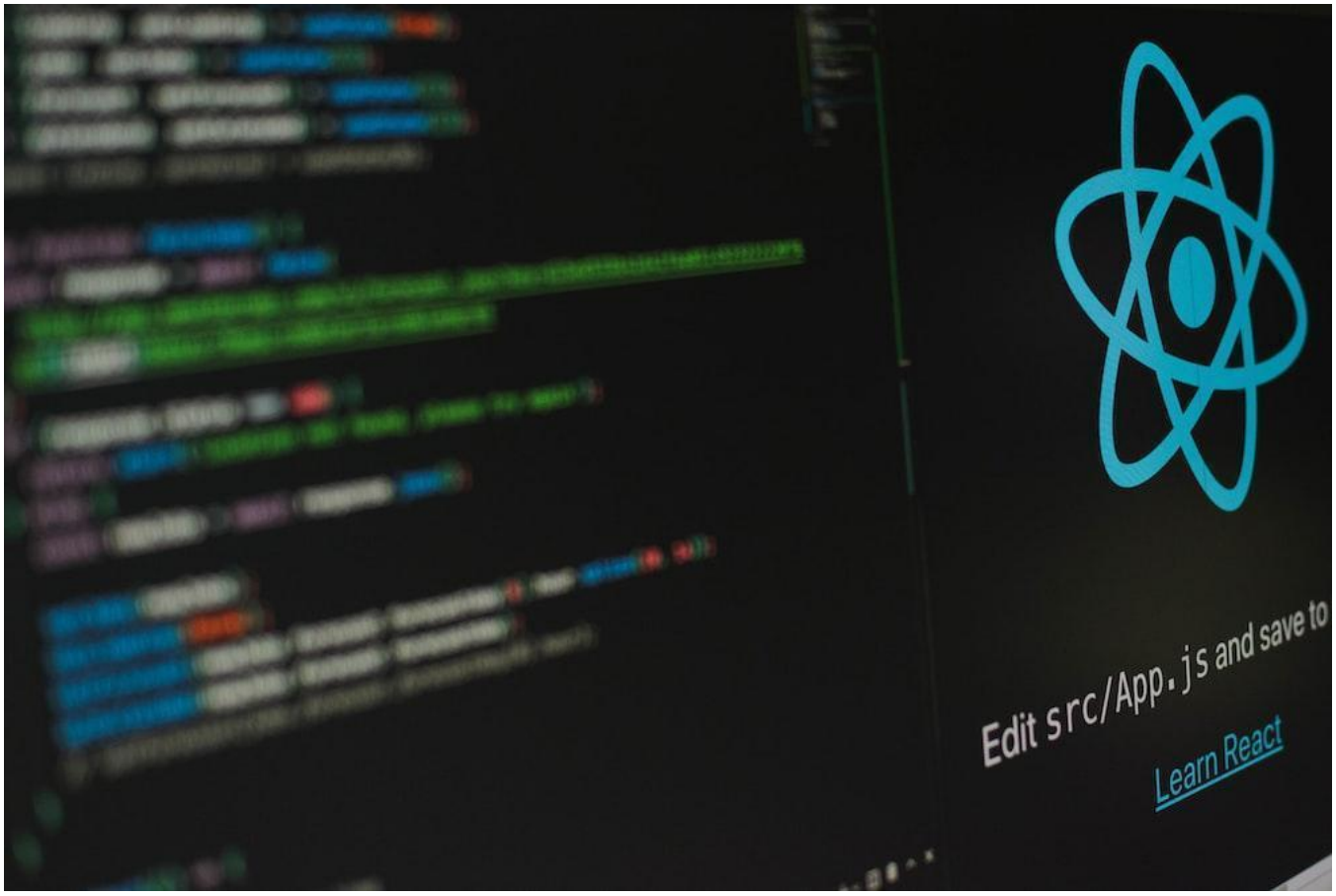
Perform the following:

a) Install Node.js and NPM on the computer.
b) Open the terminal and create a new folder called "**my-react-app**".
c) Navigate into the "**my-react-app**" folder using the terminal.
d) In the terminal, initialize a new **Node.js** project to create a new **package.json** file to track project's dependencies.
e) Install the ReactJS and React-DOM packages to create a ReactJS application.
f) Create a new file called "**index.js**" in the "**my-react-app**" folder that renders a "**Hello, world!**" message.
g) Create a new file called "**index.html**" in the "**my-react-app**" folder to define a basic HTML file with a div element that will be used to mount the ReactJS component, as well as a script tag that links to the "**index.js**" file.
h) Start the development server and launch the ReactJS application in the browser.

- Explore the possibilities of adding following components in the app: navigation, card, form, modal, and table.

Make sure that each component has its own file and that it is imported and used in the App.js file.

Note: Feel free to use any external resources or tutorials to help you complete the assignment.



Session 2

ReactJS Components

Session Overview

This session provides information about the types of ReactJS components. It also provides information about passing data between components.

Objectives

In this session, students will learn to:

- Explain ReactJS components and the types of ReactJS components
- Distinguish between Functional and Class components
- Explain how to pass data between ReactJS components

2.1 Components

The ReactJS team came up with the brilliant idea of components in Web development. In JavaScript, developers have been reusing the logic through functions. Functions are the building blocks of JavaScript because functions can be declared once and reused ‘n’ number of times. The team effectively extended this concept of reusing.

The team decided to reuse views (HTML+CSS), in addition to reusing logic. This led to the birth of components. In simple terms:

For example: Consider components to be Lego blocks. Users can build a large application by putting these Lego blocks together as shown in Figure 2.1.

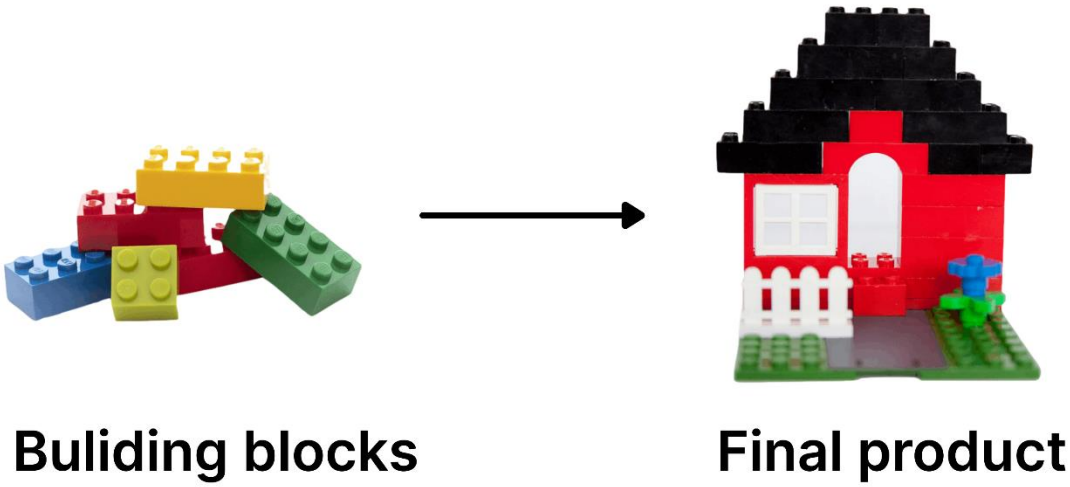


Figure 2.1: Joining Lego Blocks

You can reuse Lego blocks many times, as required. Similarly, users can build a component once and reuse it many times, as shown in Figure 2.2.

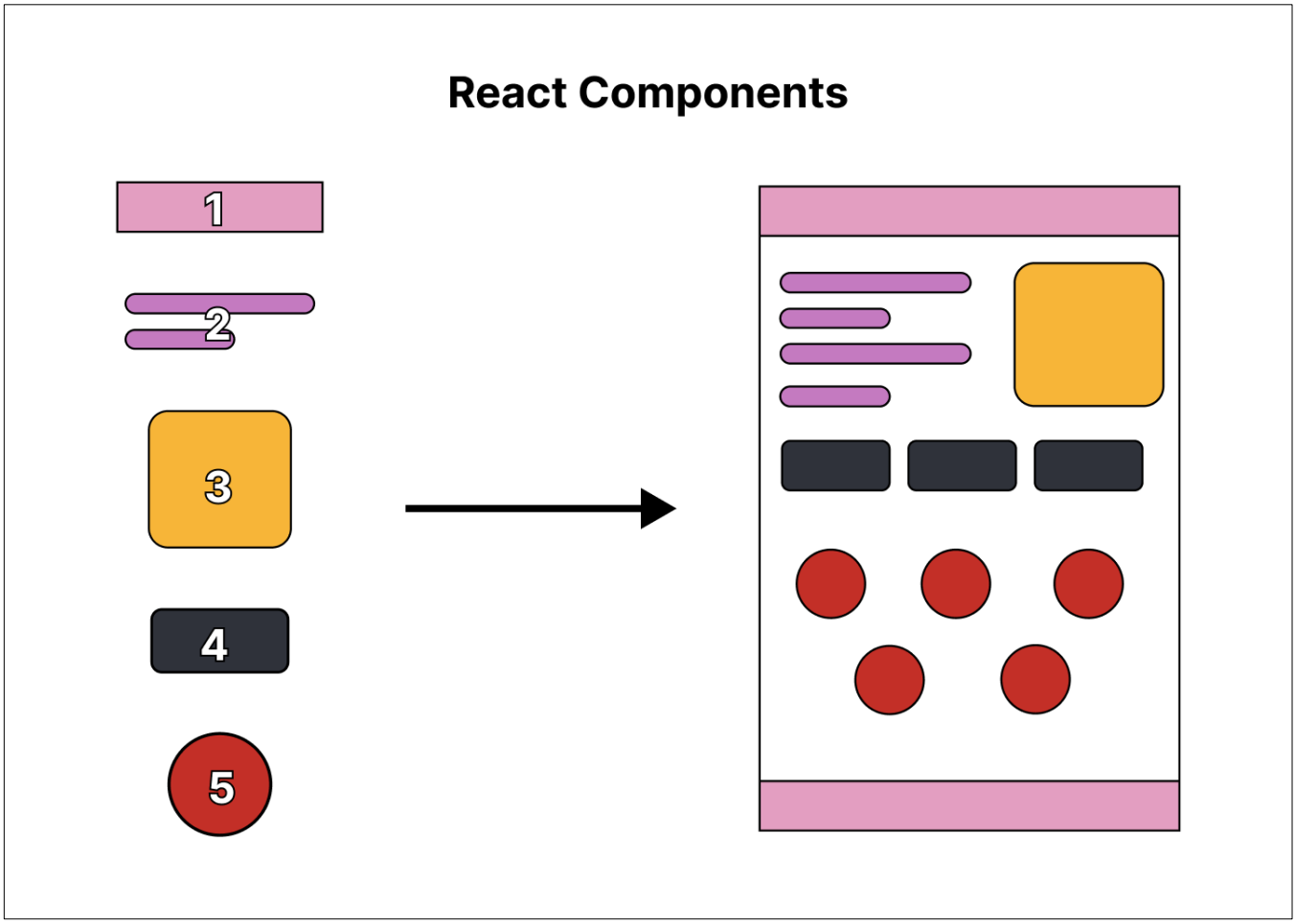


Figure 2.2: Reusing Components

Figure 2.3 shows some real-world examples of components.

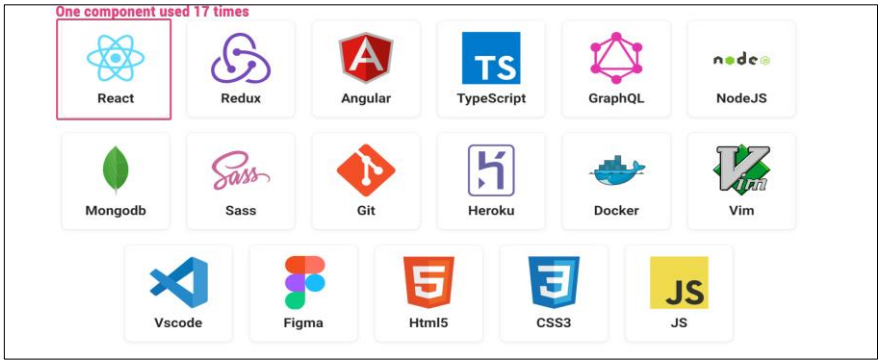


Figure 2.3: Real-World Example of Components

2.2 Types of Components

There are two types of components:

- Functional components
- Class components

2.3 Using Functional Components

Functional components are more declarative. The two conditions for normal functions to be considered as functional components by ReactJS as shown in Code Snippet 1 are:

- The name of the function must start with a capital letter.
- The function must return a JavaScript XML (JSX) element.

> **Pro Tip:**
> To type emojis hold **win** + (win is the Windows key on the keyboard)**.** The emoji panel opens and allows students to choose an emoji.

All the Code Snippets mentioned are written in App.js.

Code Snippet 1:

```
function Welcome() {
  return (
    <section>
      <h1>Hello World 🎉🎉</h1>
    </section>
  );
}
```

The `Welcome` component is called inside the `App` component as shown in Code Snippet 2.

Code Snippet 2:

```
function App() {
    return (
      <div className="App">
        <Welcome />
      </div>
    );
}
```

The output of the code is shown in Figure 2.4.

Figure 2.4: Output of Code Snippet 2

Component Reusability

Component reusability is shown in Code Snippet 3.

Code Snippet 3:

```
function App() {
    return (
      <div className="App">
        <Welcome />
        <Welcome />
        <Welcome />
      </div>
    );
  }
```

The output of the code is shown in Figure 2.5.

Figure 2.5: Output of Code Snippet 3

2.4 Using Class Components

Class components serve the same purpose as that of functional components. When declaring a class component, as shown in Code Snippet 4, the `React.Component` is used to extend `Welcome`. Thus, the `React.Component` serves as the base class for creating a component. A special `render ( )` method is used to display the view on the screen.

Code Snippet 4:

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello World 🎉🎉</h1>;
  }
}
```

The `Welcome` component is called inside the `App` component as shown in Code Snippet 5.

Code Snippet 5:

```
class App extends React.Component {
    render() {
      return <Welcome />;
```

```
    }
}
```

The output of the code is shown in Figure 2.6.

Figure 2.6: Output of Code Snippet 5

ReactJS started with class components, but later in ReactJS version **16.8** (2018), the team introduced functional components. From that time onwards, the ReactJS ecosystem moved from **Class** to **Function**. This movement was fast-tracked by the introduction of **hooks,** which will be discussed in later chapters.

2.4.1 Drawbacks of Using class Components

Writing class components leads to larger boilerplate code. It generally needs mastery of in-depth concepts like `this` keyword. Hence, this led to lesser use of class components over the years. Currently, in companies, only functional components are used.

In Figure 2.7, both the code snippets have exactly the same functionality, but functional components could do to it with 40% lesser code. This comparison shows the power of functional components.

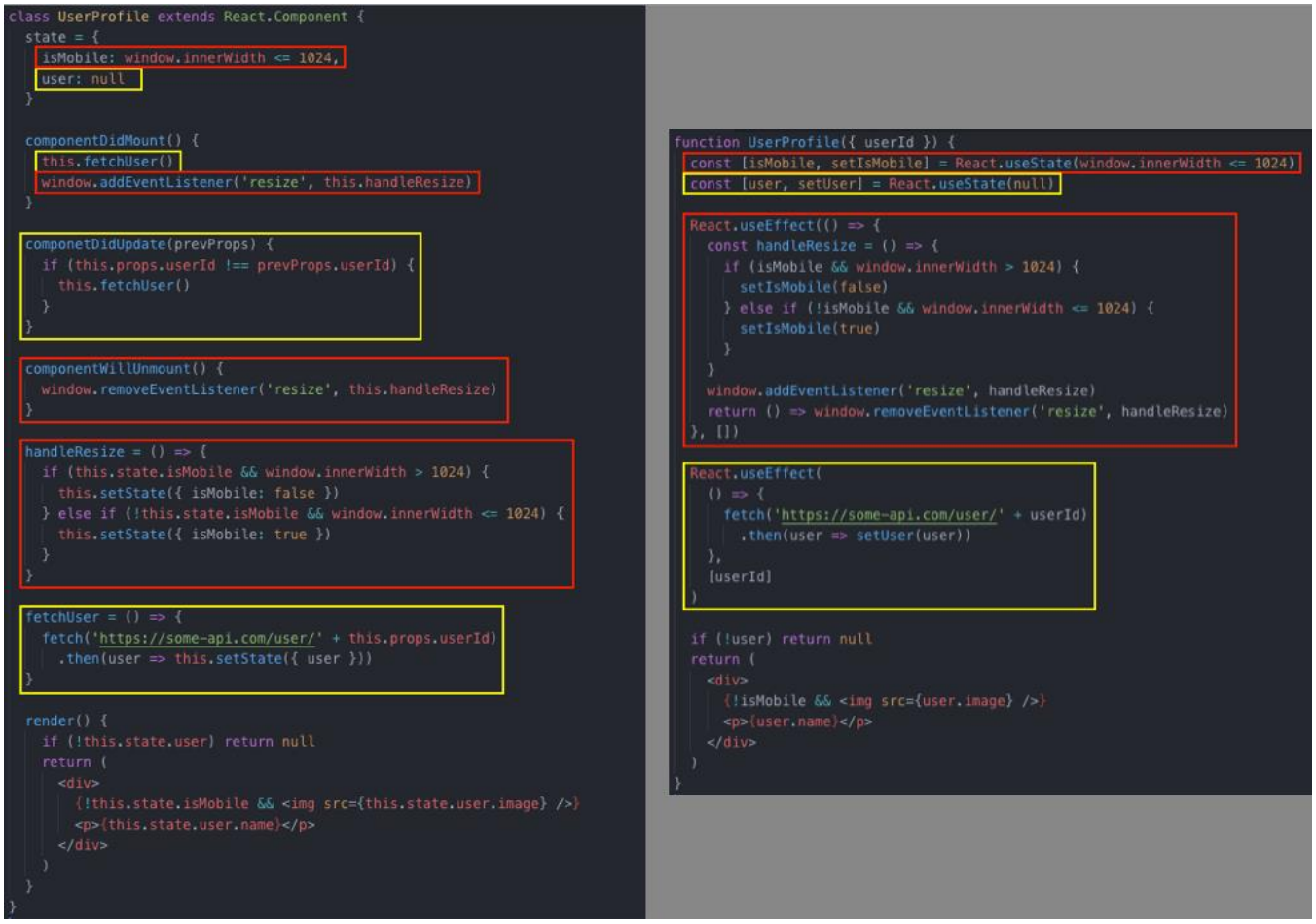


Figure 2.7: Code Snippets Using Class Component and Functional Component

2.5 Passing Data Between Components

In ReactJS, data flows only in one direction, wherein the data flows only from the parent to child. The data that gets passed is called as props or properties.

2.5.1 Parent → Child Component

Code Snippet 6 and Code Snippet 7 show the flow of data from parent to child.

Code Snippet 6:

```
function App() {
  return (
    <div className="App">
      <Welcome name="Neo" />
    </div>
  );
}
```

Code Snippet 7:

```
function Welcome(props) {
  return (
    <section>
      <h1>
```

```
        Hello, <span>{props.name}</span> 🎉🎉
      </h1>
    </section>
  );
}
```

The output of the Code Snippets is shown in Figure 2.8. The data flows from App to the Welcome component.

Figure 2.8: Output of Code Snippets 6, 7, and 8

2.5.2 Destructuring Props

Code Snippet 7 can be shortened by using the destructuring feature in ES6. The props have been destructured to get the `name` key out as shown in Code Snippet 8, which is an equivalent code of Code Snippet 7.

Code Snippet 8:

```
function Welcome({ name }) {
    return (
      <section>
        <h1>
          Hello, <span>{name}</span> 🎉🎉
        </h1>
      </section>
    );
  }
```

2.5.3 Child → Parent

There is no direct way to pass data from child to parent. Hooks are a workaround to pass data from child to parent. This is especially useful when one has to inform the parent component about changes in the child component.

2.6 Higher Order Components

Higher-Order Components (HOC) wrap components. This wrapping makes it easier to pass logic to components. HOC acts like a template component for building new components. It is especially useful in theming and customizing a component.

2.6.1 Using Higher Order Components

An example of using Higher Order Components is shown in Figure 2.9.

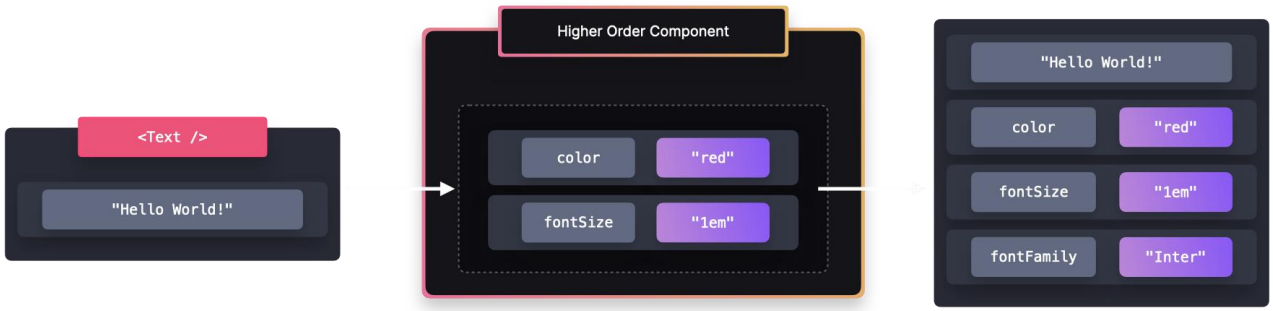


Figure 2.9: Higher Order Component

`withStyles` is the HOC component, which is used to create new components by wrapping the existing component, as shown in Code Snippets 9, 10, and 11.

Code Snippet 9:

```
function withStyles(Component) {
    return (props) => {
      const style = {
        color: "red",
        fontSize: "1em",
        // Merge props
        ...props.style,
      };

      return <Component {...props} style={style} />;
    };
}
}
```

`<StyledText />` is the new component that is created with the existing component `<Text />` using HOC `withStyles()` as shown in Code Snippet 10 and Code Snippet 11.

Code Snippet 10:

```
const Text = ({ style = {} }) => (
  <p style={{ ...style, fontFamily: "Inter" }}>Hello world!</p>
); // existing component

const StyledText = withStyles(Text); // new Component
```

Code Snippet 11:

```
function App() {
  return (
    <div className="App">
      <Welcome name="Neo" />
      <Text />
      <StyledText />
    </div>
```

```
    );
}
```

The output displaying HOC style is shown in Figure 2.10.

Figure 2.10: Output Containing HOC Style

2.7 Summary

- ✓ Components are the building blocks of ReactJS apps.
- ✓ Two types of ReactJS components are class components and functional components.
- ✓ The functional component is the preferred one.
- ✓ Props flow in one direction.
- ✓ HOC are template components for building new components.

2.8 Check Your Knowledge

1. Which is the building block of ReactJS apps?

A	Functional component
B	Class component
C	None
D	Both Class and Functional components

2. Props data flow is bidirectional.

A	True
B	False

3. Component = ___?

A	View + CSS
B	View + HTML
C	View + Logic
D	Logic + CSS

4. Which is the right way to call a component, `Hello`?

A	`Hello()`
B	`<Hello />`
C	`<Hello></Hello>`
D	Both B and C

5. Which of the options is not an advantage of the functional component?

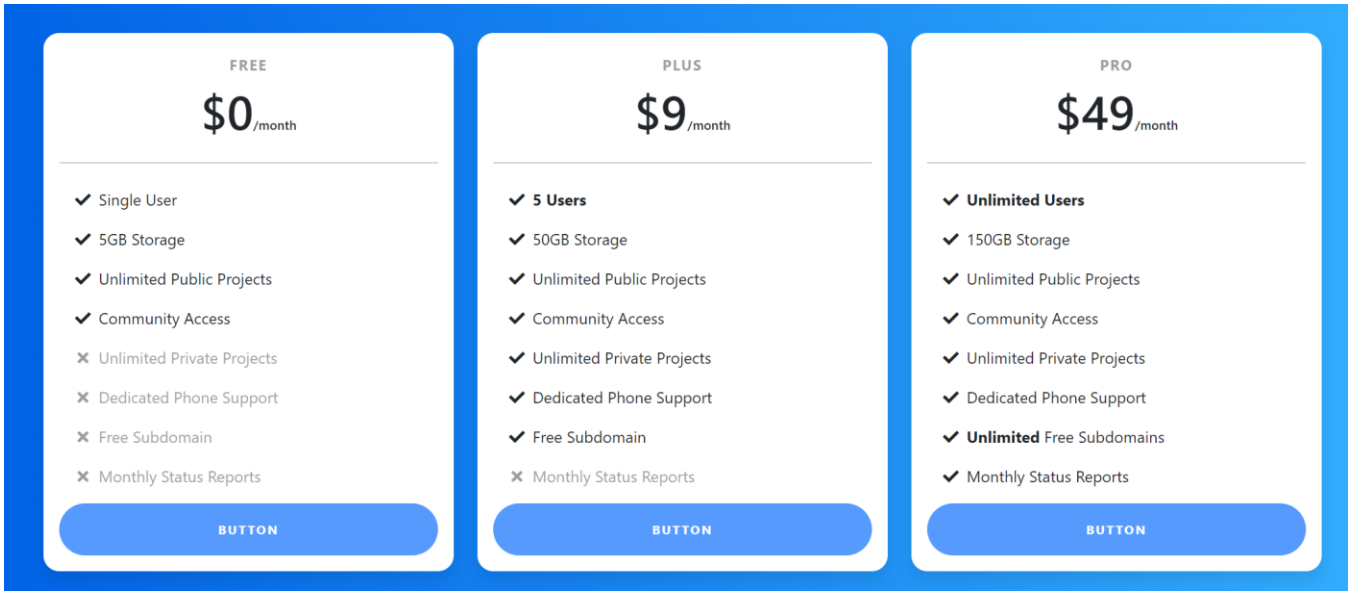
A	It has concise syntax.
B	It includes lots of boilerplate code.
C	It is easy to develop.
D	There is no need of in-depth knowledge.

Answers

1.	D
2.	B
3.	C
4.	D
5.	B

Try it Yourself

You are a developer in an ecommerce startup and will be developing a pricing page, which is part of the startup's app. Implement the design as shown in the following figure with ReactJS functional components.



Ensure the following:

- Make the design responsive so that it adapts to different screen sizes.
- Make the components reusable.

Session 3

JSX Elements

Session Overview

This session provides information about creating, nesting, and rendering JSX elements. It also provides information about JSX expressions and properties.

Objectives

In this session, students will learn to:

- Illustrate creating and rendering JSX elements
- Illustrate nesting JSX Elements
- Explain JSX expressions
- Explain the properties of JSX

3.1 Creating and Rendering JSX Elements

JavaScript XML (JSX) is a syntax extension for JavaScript that lets you write HTML-like markup within a JavaScript file. The Browser understands HTML, CSS, and JavaScript. The browser does not understand JSX. Hence, JSX is converted into JavaScript by Webpack and Babel.

During the initial stages of Web development, the developers kept content in HTML, design in CSS, and logic in JavaScript, usually as separate files. Content was marked up inside HTML as illustrated in Figure 3.1, while the logic of the page was present separately in JavaScript, as illustrated in Figure 3.2.

```
<div>
  <p></p>
  <form>
  </form>
</div>
```

Figure 3.1: HTML File

```
isLoggedIn() {...}
onClick() {...}
onSubmit() {...}
```

Figure 3.2: JavaScript File

Logic started to determine content as the Web started to become more interactive. JavaScript was in charge of the HTML. That's why in ReactJS, rendering logic and markup live together in the same place—components.

For example, keeping a rendering logic of the button and the markup together ensures that they stay in sync with each other on every edit. Conversely, unrelated details, such as the markup of the button, as shown in Figure 3.4 and the markup of the sidebar as shown in Figure 3.3 are isolated from one another. This isolation makes it safe to change the codes of independently, depending on the requirement.

```
Sidebar() {
  if (isLoggedIn()) {
    <p>Welcome</p>
  } else {
    <Form />
  }
}
```

Figure 3.3: Markup of the Sidebar

```
Form() {
  onClick() {...}
  onSubmit() {...}
  <form onSubmit>
    <input onClick />
    <input onClick />
  </form>
}
```

Figure 3.4: Markup of the Button

JSX looks much like HTML, but it is a bit **stricter** and can display **dynamic** information. The most optimal way to understand this is to convert any **HTML** markup to **JSX** markup.

3.2 Properties of JSX

Consider an example of converting an HTML code into JSX markup. Code Snippet 1 represents a valid HTML code.

Note: As HTML is a forgiving language, the closing `<li>` tags are optional.

Code Snippet 1:

```
<h1>Mark</h1>
<img
  src="https://images.pexels.com/photos/1704488/pexels-photo-
1704488.jpeg?auto=compress&cs=tinysrgb&dpr=1&w=500"
  alt="Mark"
>
<h2>Hobbies</h2>
<ul>
    <li>Badminton
    <li>Chess
    <li>Gaming
</ul>
```

The output of Code Snippet 1 is shown in Figure 3.5.

Figure 3.5: Output of the HTML Code Snippet 1

From Figure 3.5, students can see that the HTML is rendered in the browser without any hiccups.

The first attempt to convert the HTML to JSX markup is copying and pasting the Code Snippet 1 into the `return` part of Code Snippet 2 as shown in Code Snippet 4.

Code Snippet 2:

```
function UserProile() {
  return (
    // ???
 )
}
```

Code Snippet 3 is the CSS code that is used throughout examples. It must be written in `App.css`.

Code Snippet 3:

```
.App {
  text-align: center;
```

```
}
.user-name {
  font-family: Roboto;
  font-weight: normal;
}
.user-profile-pic {
  height: 250px;
  aspect-ratio: 1 / 1;
  border-radius: 50%;
  object-fit: cover;
}
.user-first-name {
  font-weight: bold;
  font-style: italic;
  color: orange;
}
```

Code Snippet 4 must be written in `App.js`.

Code Snippet 4:

```
function UserProile() {
  return (
      <h1>Mark</h1>
      <img
          src="https://images.pexels.com/photos/1704488/pexels-photo-
1704488.jpeg?auto=compress&cs=tinysrgb&dpr=1&w=500"
          alt="Mark"
          class="user-profile-pic"
      >
      <h2>Hobbies</h2>
      <ul>
          <li>Badminton
          <li>Chess
          <li>Gaming
      </ul>
 )
}
```

Code Snippet 4 does not work because JSX is stricter, and some rules must be followed for the code to work.

3.2.1 Rules and Properties

JSX follows three rules:

Returning a Single Root Element

Multiple elements cannot be returned at the same time. Hence, to work around this problem, multiple elements are wrapped with a single parent tag as in Code Snippet 5 and returned. The Code Snippet is written in `App.js`.

Code Snippet 5:

```
function UserProile() {
    return (
      <div>
        <h1>Mark</h1>
        <img
            src="https://images.pexels.com/photos/1704488/pexels-photo-
1704488.jpeg?auto=compress&cs=tinysrgb&dpr=1&w=500"
            alt="Mark"
            class="user-profile-pic"
        >
        <h2>Hobbies</h2>
        <ul>
            <li>Badminton
            <li>Chess
            <li>Gaming
        </ul>
      </div>
   )
  }
```

Following are some wrap options:

- In Code Snippet 5, `<div></div>` is used to wrap the elements in return.
- Another option wrap `<></>` as shown in Code Snippet 6.

<> and </> are called Fragments in ReactJS.

Code Snippet 6 is written in `App.js`.

Code Snippet 6:

```
function UserProile() {
    return (
      <>
        <h1>Mark</h1>
        <img
            src="https://images.pexels.com/photos/1704488/pexels-photo-
1704488.jpeg?auto=compress&cs=tinysrgb&dpr=1&w=500"
            alt="Mark"
            class="user-profile-pic"
        >
        <h2>Hobbies</h2>
        <ul>
            <li>Badminton
            <li>Chess
            <li>Gaming
        </ul>
      </>
   )
  }
```

Closing All Tags

JSX is stricter than HTML. JSX requires tags to be explicitly closed as shown in Code Snippet 7, such as:

- Self-closing tags like `<img>` must become `<img/>`.
- Wrapping tags like `<li>Badminton` must be written as `<li>Badminton</li>`.

Code Snippet 7 is written in `App.js`.

Code Snippet 7:

```
function UserProile() {
  return (
    <div>
      <h1>Mark</h1>
      <img
        src="https://images.pexels.com/photos/1704488/pexels-photo-
1704488.jpeg?auto=compress&cs=tinysrgb&dpr=1&w=500"
```

```
        alt="Mark"
        class="user-profile-pic"
      />
      <h2>Hobbies</h2>
      <ul>
        <li>Badminton </li>
        <li>Chess </li>
        <li>Gaming </li>
      </ul>
    </div>
  );
}
```

Using Camel Case (CamelCase)

JSX turns into JavaScript and attributes written in JSX become keys of JavaScript objects. Developers will often want to read those attributes into variables in the components used, but JavaScript has limitations on variable names. For example, the names cannot contain dashes or be reserved words like class.
Hence, in ReactJS, many HTML and Scalable Vector Graphics (SVG) attributes are written in CamelCase.

Example:

- Use `strokeWidth` instead of `stroke-width`.
- `class` is a reserved word. Hence, in ReactJS, you write `className` instead, named after the corresponding DOM property.

Hence in Code Snippet 8, the `class` attribute is converted into `className`.

Exceptional Cases:

`aria-*` and `data-*` attributes are written as in HTML with dashes due to historical reasons.

Code Snippet 8 is written in `App.js`.

Code Snippet 8:

```
function UserProile() {
    return (
      <div>
        <h1>Mark</h1>
        <img
          src="https://images.pexels.com/photos/1704488/pexels-photo-1704488.jpeg?auto=compress&cs=tinysrgb&dpr=1&w=500"
          alt="Mark"
          className="user-profile-pic"
```

```
        />
        <h2>Hobbies</h2>
        <ul>
          <li>Badminton </li>
          <li>Chess </li>
          <li>Gaming </li>
        </ul>
      </div>
    );
  }
```

3.3 JSX Expressions

JSX allows developers to write HTML-like markup inside a JavaScript file by keeping the rendering logic and content in the same place. Sometimes, developers might want to add some JavaScript logic or reference a dynamic property inside that markup. In such scenarios, the JSX code can include curly braces to open a window to JavaScript. For example, the name is defined and used with curly braces {} as shown in Code Snippet 9.

Curly braces {} is known as Template syntax in ReactJS and it supports JavaScript

Code Snippet 9 is written in `App.js`.

Code Snippet 9:

```
function Welcome() {
  const name = "Mark"
  return (
    <section>
      <h2 className="user-name">
        Hello, <span className="user-first-name">{name}</span> 🎉🔥
      </h2>
    </section>
  );
}
```

The output of Code Snippet 9 is shown in Figure 3.6.

Figure 3.6: Output of Code Snippet 9

To pass a string attribute to JSX, add it within single or double quotes.

In Code Snippet 10, `"user-profile-pic"` is passed as strings to `className`.

If `src` or `alt` needs text dynamically. Instead of specifying the value with "", it could be taken from JavaScript with curly braces {}.

Code Snippet 10 is written in `App.js`.

Code Snippet 10:

```
function User() {
  const name = "Mark";
  const pic =
    "https://images.pexels.com/photos/1704488/pexels-photo-
1704488.jpeg?auto=compress&cs=tinysrgb&dpr=1&w=500";


  return (
    <section>
      <img className="user-profile-pic" src={pic} alt={name} />
      <h2 className="user-name">
        Hello, <span className="user-first-name">{name}</span> 🎉🔥
      </h2>
    </section>
  );
}
```

The output of Code Snippet 10 is shown in Figure 3.7.

Figure 3.7: Output of Code Snippet 10

3.4 Nesting JSX Elements

Many times, multiple similar components need to be displayed from a collection of data. JavaScript array methods could be used to manipulate the array of data. In Code Snippet 12, the `map()` method is used to transform an array of strings into an array of components.

In Code Snippet 11, the `Welcome` component takes the `name` props. Code Snippet 11 is written in `App.js`.

Code Snippet 11:

```
function Welcome({ name }) {
  return (
    <section>
     <h2 className="user-name">
        Hello, <span className="user-first-name">{name}</span> 🎉🔥
      </h2>
    </section>
  );
}
```

Hence, to have a list of Welcome components, `map()` methods are used to iterate through.

Code Snippet 12 is written in App.js.

Code Snippet 12:

```
function App() {
  const names = ["Cuban", "Spencer", "Robert", "Einstein"];
  return (
    <div className="App">
      {names.map((nm) => (
```

```
        <Welcome name={nm} />
      ))}
    </div>
  );
}
```

In Code Snippet 12, `{}` is used to pass the value `nm` as `prop` to `Welcome`. `{}` must be included because `map()` uses an arrow function (=>), which returns JSX. This nesting of JSX is required to convert an array of strings to an array of JSX elements.

The output of Code Snippet 12 is shown in Figure 3.8.

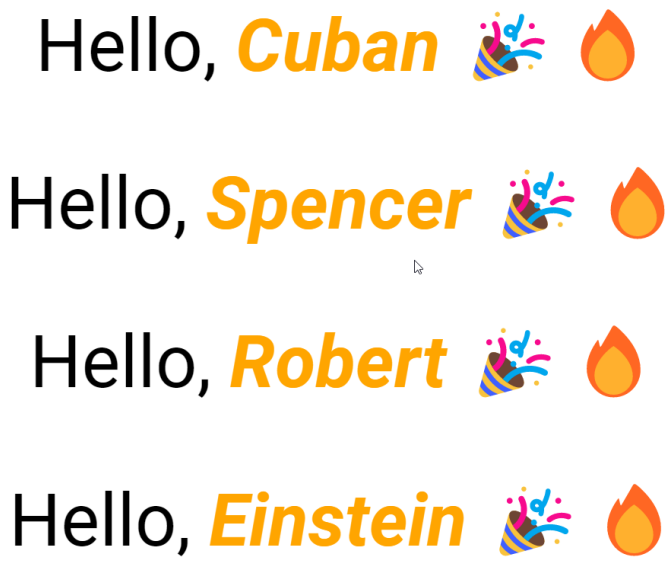


Figure 3.8: Output of Code Snippet 12

Code Snippet 13 takes name and pic as props so that both the values could be displayed on the screen. Code Snippet 13 is written in `App.js`.

Code Snippet 13:

```
function User({ name, pic }) {
  return (
    <section>
      <img className="user-profile-pic" src={pic} alt={name} />
      <h2 className="user-name">
        Hello, <span className="user-first-name">{name}</span> 🎉🔥
      </h2>
    </section>
  );
}
```

Code Snippet 14 is an additional example where an array of objects is converted to an array of components with the `map()` method. Code Snippet 14 is written in `App.js`.

Code Snippet 14:

```
function App() {
  const users = [
    {
      name: "Cuban",
      pic: "https://images.unsplash.com/photo-1618641986557-
1ecd230959aa?ixlib=rb-
4.0.3&ixid=MnwxMjA3fDB8MHxzZWFyY2h8NXx8cHJvZmlsZXxlbnwwfHwwfHw%3D&w=1000&q=80",
    },
    {
      name: "Spencer",
      pic: "https://images.unsplash.com/photo-1529665253569-
6d01c0eaf7b6?ixlib=rb-
4.0.3&ixid=MnwxMjA3fDB8MHxzZWFyY2h8NHx8cHJvZmlsZXxlbnwwfHwwfHw%3D&w=1000&q=80",
    },
    {
      name: "Robert",
      pic: "https://encrypted-
tbn0.gstatic.com/images?q=tbn:ANd9GcT2C_xadF4WT19MkU5PpYyU8njyMgMIuttwXQ&usqp=C
AU",
    },
    {
      name: "Einstein",
      pic: "https://media.istockphoto.com/id/1179420343/photo/smiling-man-
outdoors-in-the-city.jpg?s=612x612&w=0&k=20&c=8l-
gOboGEFSyCFXr09EguDmV0E0bFT5usAms1wyFBh8=",
    },
  ];
  return (
    <div className="App">
      {users.map((usr) => (
        <User name={usr.name} pic={usr.pic} />
      ))}
    </div>
  );
}
```

The output of Code Snippets 13 and 14 is shown in Figure 3.9.

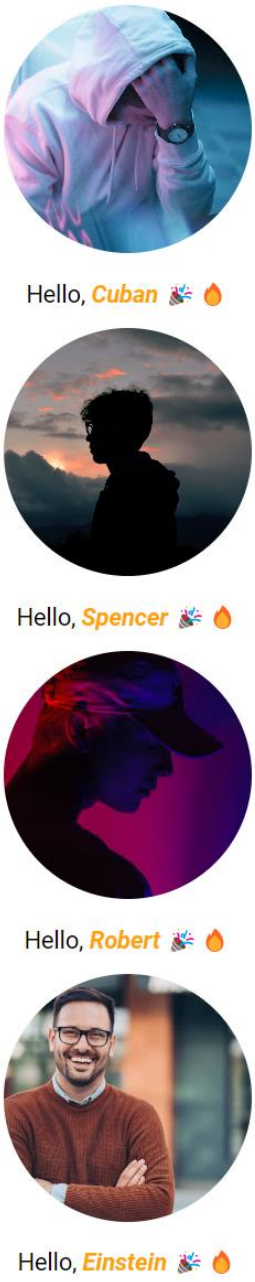


Figure 3.9: Output of Code Snippets 13 and 14

3.5 Summary

- ✓ JavaScript XML (JSX) is used to keep related logic and View in one place.
- ✓ `{}` is known as Template syntax.
- ✓ Template syntax is used to evaluate JavaScript expressions.
- ✓ JSX is stricter than HTML.
- ✓ JSX follows three rules:
    - Returning a single root element.
    - Closing all tags.
    - Using Camel case (CamelCase).
- ✓ The `map()` method is used to iterate through array of data in ReactJS.

3.6 Check Your Knowledge

1. JSX can return multiple elements.

A	True
B	False

2. Template syntax supports expressions.

A	True
B	False

3. ___ array method is used to convert data into a list of components?

A	`forEach`
B	`push`
C	`map`
D	`pop`

4. Which is the attribute if used, throws error in JSX?

A	`for`
B	`id`
C	`placeholder`
D	`type`

5. When does JSX not give an error?

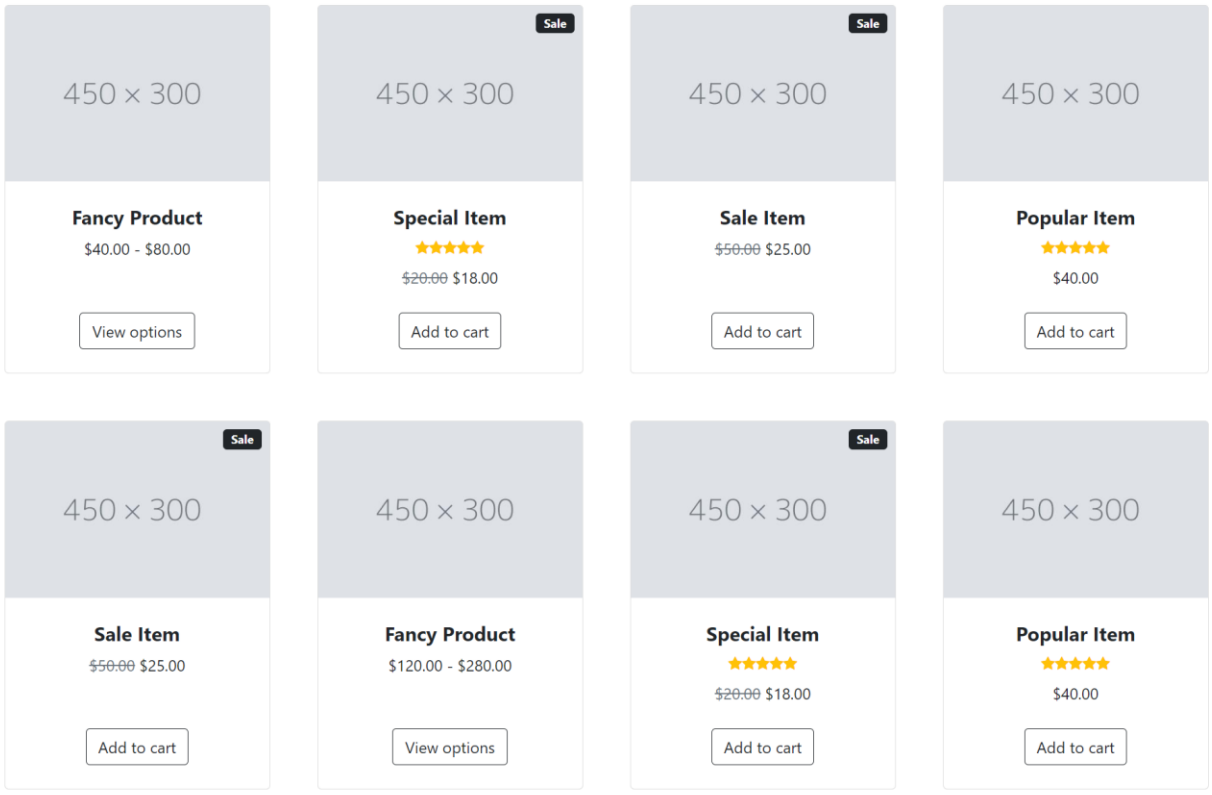
A	Unclose tags
B	Using self-closing tags
C	Return multiple elements
D	Using reserved keywords as attribute

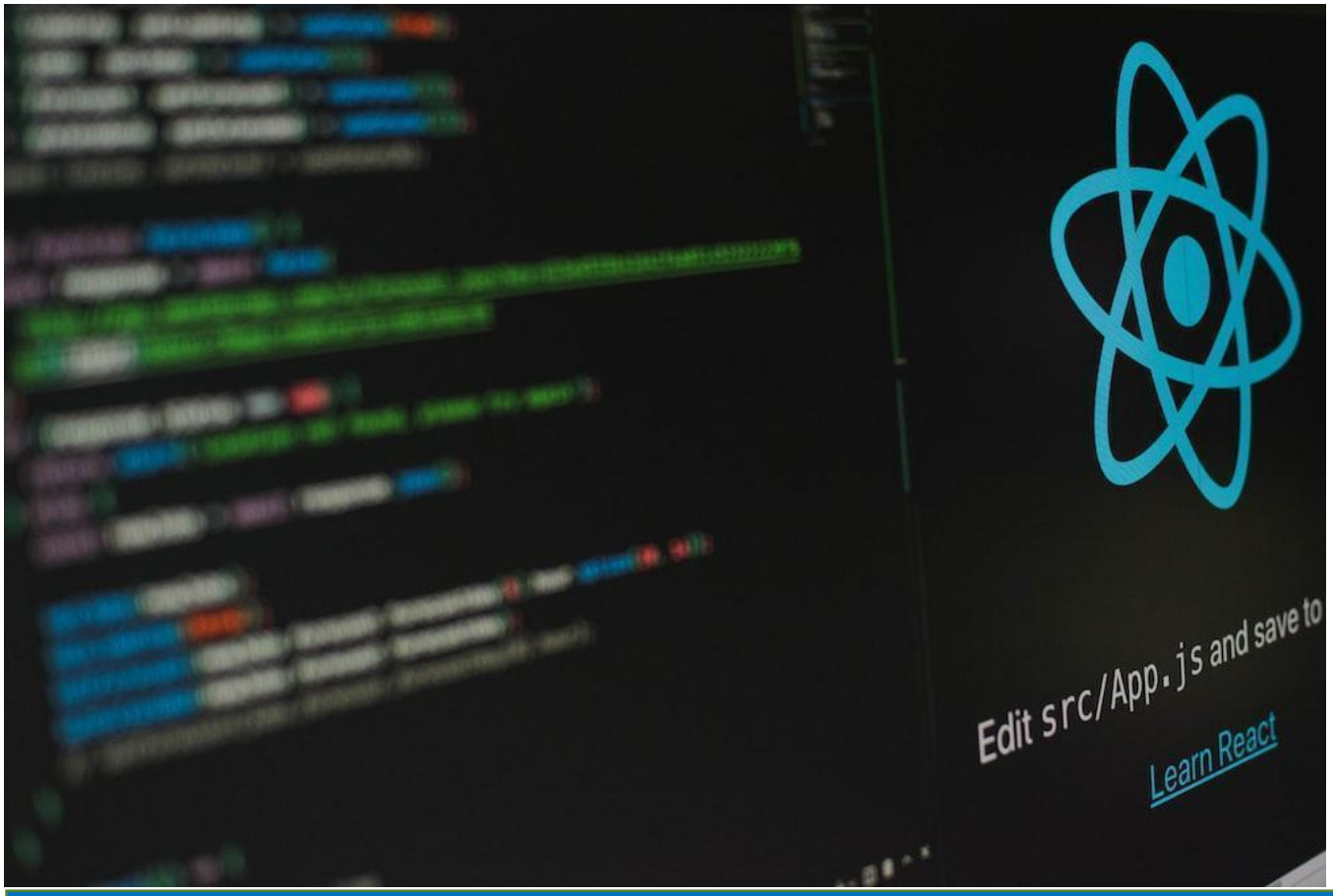
Answers

1	B
2	A
3	C
4	A
5	B

Try it Yourself

You are a developer, who is part of an e-commerce project. You will be developing a pricing page, wherein you will display a list of products. This design must allow you to choose the product list with prices, pictures, and ratings. Implement the design as shown in the following figure and ensure that the app layout is responsive, in other words, adapting to different screen sizes.





Session 4

Event Handling in ReactJS

Session Overview

This session provides information about handling different events in ReactJS. It also provides an overview about the types of events, event handling practices, and outlines the concept of Hooks.

Objectives

In this session, students will learn to:

- Explain event handling in ReactJS
- Explain the basic syntax of ReactJS event handling
- Explain the different types of events
- Explain the best practices of event handling
- Explain Hooks

4.1 What is Event Handling in ReactJS?

Interactivity is handled through event handlers on the Web. Consider a basic app that has been designed and developed. This is a static app. However, when users begin to use this app and interact with it by entering details or selecting options, and the app also reacts to this user interaction, then the app becomes an interactive app.

Example: The YouTube like and dislike buttons and the comment sections are places where users can interact with YouTube (www.youtube.com). This introduces dynamicity, thus creating a fun experience for the user.

There are many ways a user can interact. In ReactJS, there are a plethora of event handlers available. In this session, the common event handlers are discussed.

4.2 Basic ReactJS Event Handling Syntax

Users can interact through mouse events, keyboard events, touch events, and by filling forms. The basic ReactJS event handling syntax is shown in Figure 4.1.

Figure 4.1: Basic ReactJS Event Handling Syntax

When the user clicks the `Click me` button, the `Callback function` is executed. Thus, the app reacts to user interaction.

```
<button onClick={() => console.log("Clicked 👆")}>Click me</button>
```

Code Snippet 1 contains the `App.css` styles that are required for the app. Code Snippet 1 is written in `App.css`.

Code Snippet 1:

```
.App {
  text-align: left;
  margin: 0 auto;
  max-width: 400px;
}
.user-name {
  font-family: Roboto;
  font-weight: normal;
}
.user-profile-pic {
  height: 250px;
  aspect-ratio: 1 / 1;
  border-radius: 50%;
```

```
object-fit: cover;
}
.user-first-name {
font-weight: bold;
font-style: italic;
color: orange;
}
.login-form {
  display: flex;
  flex-direction: column;
  gap: 12px;
 max-width: 400px;
  margin: 20px auto;
}
.dot-container {
display: grid;
place-content: center;
  min-height: 100vh;
}
.counter-container {
  display: flex;
  gap: 0.5rem;
  justify-content: center;
 margin-bottom: 1rem;
}
.counter-container > button {
  font-size: 20px;
}
.user-container {
  text-align: center;
}
```

Best Practices of Event Handling

Some of the event handling best practices include:

- Using the appropriate naming conventions.
- Using arrow functions with event handlers.

4.2.1 Naming Conventions of Event Handlers

All event handers must be in CamelCase. ReactJS will not acknowledge the event handler, if it is not in CamelCase.

4.2.2 Using Arrow Functions with Event Handlers

The Callback function is recommended as an inline arrow function (=>) as shown in Code Snippet 2. The Code Snippet is written in `App.js`. If the Callback function is large, it must be separated by declaring the component with its name as shown in Code Snippet 3.

Code Snippet 2:

```
function ClickMe() {
  return <button onClick={() => console.log("Clicked 👆")}>Click me</button>;
}
```

Inline arrow functions are used in Code Snippet 2 as the callback is smaller in size.

Code Snippet 3:

```
function ClickMe() {
  const onUserClick = () => console.log("Clicked 👆");

  return <button onClick={onUserClick}>Click me</button>;
}
```

The Code Snippet is written in `App.js`. If the callback becomes larger, then as shown in Code Snippet 3 the callback could be extracted into a separate function (`onUserClick`).

The output of Code Snippets 2 and 3 is shown in Figure 4.2.

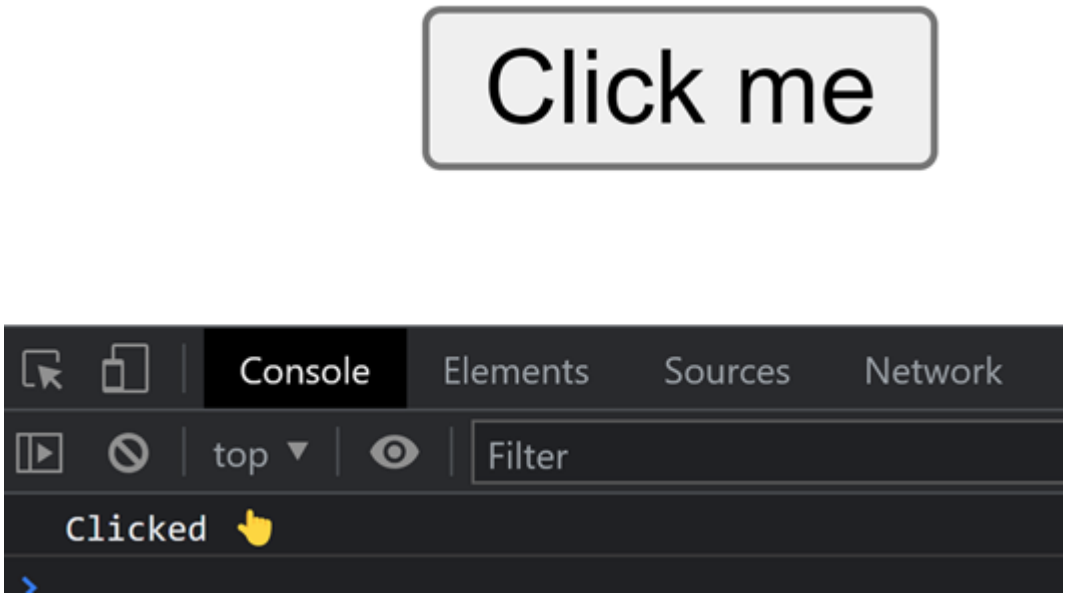


Figure 4.2: Output of Code Snippets 2 and 3

4.3 Introduction to Hooks

Hooks are functions that starts with the word, "`use`" and are usually used in combination with event handlers. ReactJS listens to changes made by hooks.

There are various hooks available in ReactJS. Following are some of the most commonly used hooks:

- `useState`
- `useEffect`
- `useNavigate`
- `useParams`

Consider the `useState` hook as shown in Code Snippet 5. State refers to the current data that a variable holds.

Code Snippet 4 includes the following:

`useState` - the hook that is imported from the ReactJS package.

`setState` - the function that updates the value of the state.

`InitialValue` - the value assigned to the state when the component is created.

Code Snippet 4:

```
import { useState } from "react";

const [state, setState] = useState(InitialValue)
```

4.4 Different Types of Events

Users can interact through mouse events, keyboard events, touch events, and by filling forms. Following are the event handlers that are frequently used in ReactJS:

`onClick`

`onChange`

`onSubmit`

`onTouchStar`

As the app becomes more interactive and complex, different types of events are required to handle them. In this section, the most commonly used events are discussed.

4.4.1 Mouse Events

The topmost interactions are done through the mouse because it is used for navigation throughout Web apps and the computer. Consider an example to discuss the usage of click events.

Example: Code Snippet 5 aims to build a like `Counter`, which on clicking will update the three `<h1>` tags.

Hence, the following occurs:

- The `like` variable is initialized with the value of 10.
- The `onClick` handler is used to update the value of `like` by incrementing it.

Code Snippet 5 is written in `App.js`.

Code Snippet 5:

```
function Counter() {
  let like = 10;
  return (
    <div>
      {/* camelcase */}
      <button
        onClick={() => {
          like++;
          console.log(like);
        }}
      >
        Like
      </button>
      <h1>{like}</h1>
      <h1>{like}</h1>
      <h1>{like}</h1>
    </div>
  );
}
```

Output of Code Snippet 5 is shown in Figure 4.3.

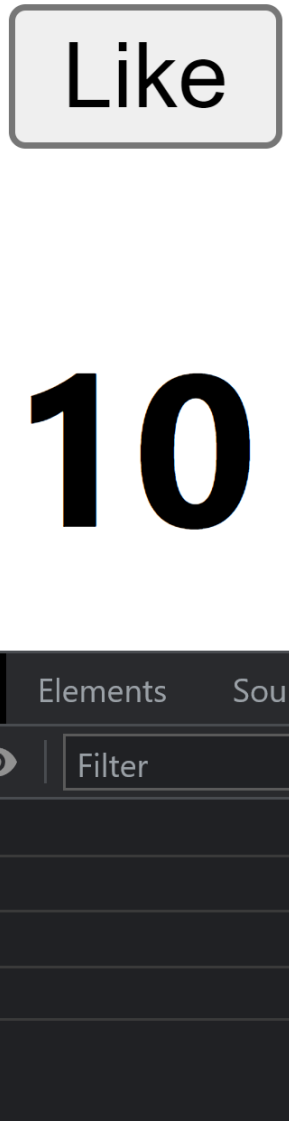


Figure 4.3: Output of Code Snippet 5

On clicking, the like button does not update the like value on the screen. However, the updated value is printed on the console.

Why is there a discrepancy?

There is a discrepancy because ReactJS does not listen to all the variables declared in the component. This slows the app greatly when `n` number of variables are declared. Hence, it only listens to the variables declared through hooks, as shown in Code Snippet 6. Code Snippet 6 is written in `App.js`.

Code Snippet 6:

```
function Counter() {
  let [like, setLike] = useState(0);


  return (
    <div>
      {/* camelcase */}
      <button onClick={() => setLike(like + 1)}>Like</button>
```

```
      <h1>{like}</h1>
    </div>
  );
}
```

The output of Code Snippet 6 is shown in Figure 4.4.

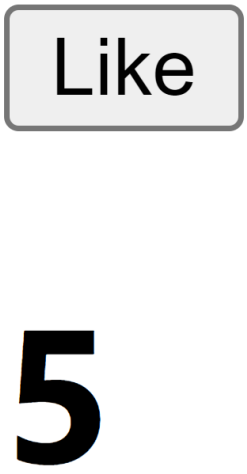


Figure 4.4: Output of Code Snippet 6

`like` is initialized with `0` and when the user clicks the button, the `onClick` event handler is triggered, which in turn calls the `setLike` function. Once the `setLike` is called, ReactJS is notified that the `like` value has to be updated and the view has to be re-rendered. Re-render refers to the re-creation of the elements on the screen. Thus, `like` is updated to `like + 1`, and the view is also re-rendered.

In Code Snippet 7, the button name is replaced with the 👍 emoji. Code Snippet 7 is written in `App.js`.

> **Pro Tip:**
> To type emojis hold **win** + (win is the Windows key)**.** The emoji panel opens and allows students to choose an emoji.

Code Snippet 7:

```
function Counter() {
  let [like, setLike] = useState(0);
  return (
    <div>
      {/* camelcase */}
      <button onClick={() => setLike(like + 1)}>👍 {like}</button>
    </div>
  );
}
```

The updated like is shown inside the button in the output as shown in Figure 4.5, creating a more pleasant experience for the user.

Figure 4.5: Output of Code Snippet 7 showing the Like Emoji

4.4.2 Keyboard Events

In addition to mouse events, the keyboard is the next go-to option to interact with the app. The keyboard is necessary for login forms, making tweets, commenting, and filling addresses. Consider an example to discuss the usage of typing events.

Example: Building a color box. The specialty of this box is, on typing, the color will change its background color to the same.

Code Snippet 8 is used to define the following:

- Initialize the styles object with a background color of orange.
- Provide the background color to the input field as an inline style.

Code Snippet 8 is written in `App.js`.

Code Snippet 8:

```
function ColorBox() {
  const styles = {
    background: "orange",
  };

  return (
    <div>
      <h1>Color Box</h1>
      <input type="text" style={styles} placeholder="Type a color" />
    </div>
  );
}
```

The style is provided as inline because it has to react to the user interaction. Code Snippet 9 has the following definitions:

- `color` state variable is declared and initialized with orange color.
- It is provided to the `background` key.

The Code Snippet 9 is written in `App.js`.

Code Snippet 9:

```
function ColorBox() {
  const [color, setColor] = useState("orange");
  const styles = {
    background: color,
  };
  return (
    <div>
      <h1>Color Box</h1>
      <input type="text" style={styles} placeholder="Type a color" />
    </div>
  );
}
```

Figure 4.6: Output of Code Snippets 8 and 9

`onChange` is the event handler that is used to listen to the typing event from the user. As shown in Code Snippet 11:

- The `event` object in the callback function contains the typed event.
- The typed value is present in `event.target.value`.

Code Snippet 10 is written in `App.js`.

Code Snippet 10:

```
function ColorBox() {
```

```
  const [color, setColor] = useState("");
  const styles = {
    background: color,
  };
  return (
    <div>
      <h1>Color Box</h1>
      <input
        type="text"
        style={styles}
        placeholder="Type a color"
        onChange={(event) => setColor(event.target.value)}
      />
    </div>
  );
}
```

The code works in the following manner:

- When the user types in the input field, the `onChange` event handler is triggered, which in turn calls the `setColor` function.
- After the `setColor` is called, ReactJS is notified that the color value has to be updated, and the view has to be re-rendered with the new inline style.
- The background color then changes to pink.

Output of Code Snippet 10 is shown in Figure 4.7.

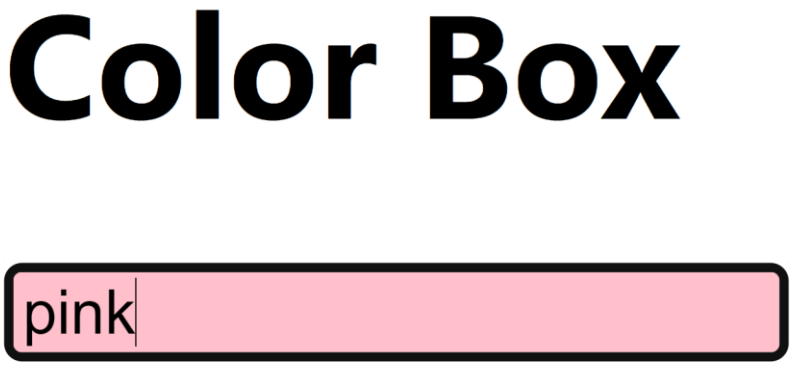


Figure 4.7: Output of Code Snippet 10

4.4.3 Form Events

Today, apps are filled with forms such as login forms, comment sections, address forms, and so on.

Consider an example to discuss the usage of `onSubmit` events.

Example: To build a login form and show the form values on the Console when the form is submitted.

In Code Snippet 11, the following input fields are added inside the form element:

- Field for email address (`email`).
- Field for password (`password`).

Code Snippet 11 is written in `App.js`.

Code Snippet 11:

```
function LoginForm() {
  return (
    <form className="login-form">
      <h1>Login Form</h1>
      <input name="email" type="email" placeholder="username" />
      <input name="password" type="password" placeholder="password" />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Output of Code Snippet 11 is shown in Figure 4.8.

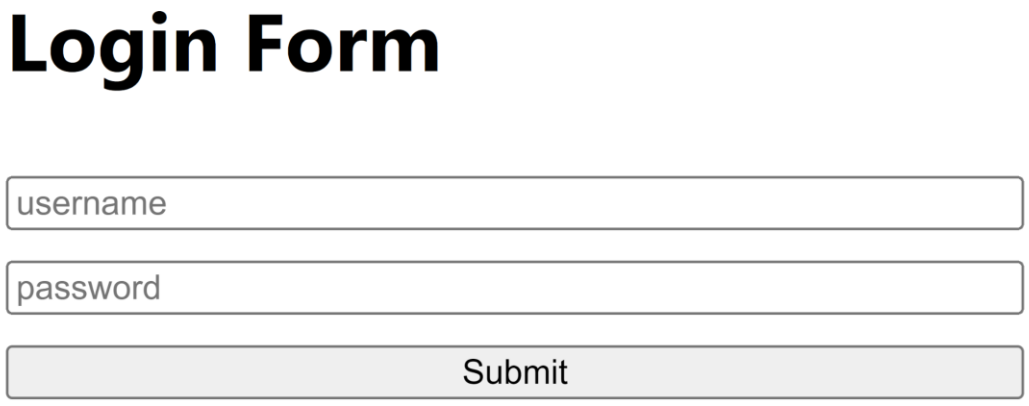


Figure 4.8: Output of Code Snippet 11

Code Snippet 12 is used to define the following:

- Two `useState` variables (`email` and `password`), which are added to track the typed values.
- The email state is initialized with the value of mark@gmail.com and the password with secret123.
- The `value` attribute in the input is set to the state variable to display the value on the screen.
- `onChange` is used to track the typed value.

Finally, when the submit button is clicked, the browser refreshes with the value in the form. However, this refresh must be avoided.

Code Snippet 12 is written in `App.js`.

Code Snippet 12:

```
function LoginForm() {
  const [email, setEmail] = useState("mark@gmail.com");
  const [password, setPassword] = useState("secret123");
  return (
    <form className="login-form">
      <h1>Login Form</h1>
      <input
        value={email}
        onChange={(event) => setEmail(event.target.value)}
        name="email"
        type="email"
        placeholder="username"
      />
      <input
        value={password}
        onChange={(event) => setPassword(event.target.value)}
        name="password"
        type="password"
        placeholder="password"
      />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Output of Code Snippet 12 is shown in Figure 4.9.

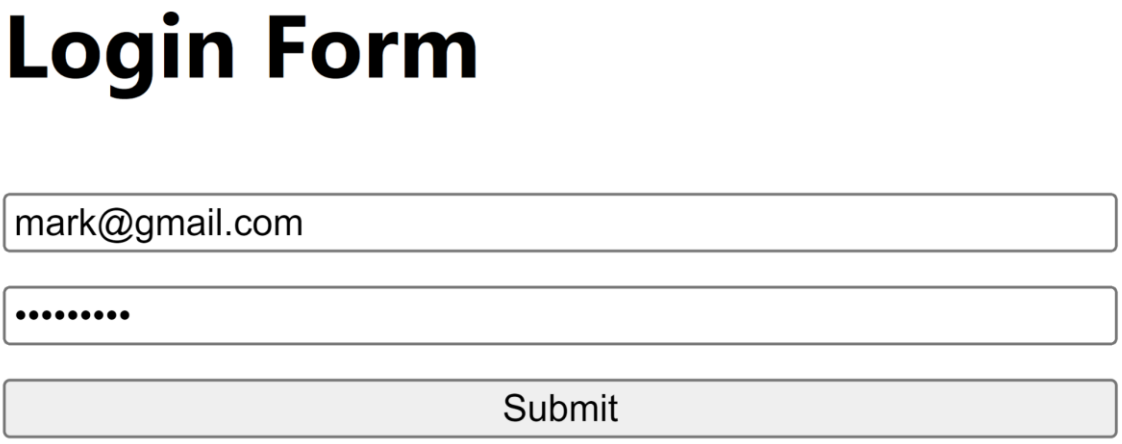


Figure 4.9: Output of Code Snippet 12

Code Snippet 13 is written in `App.js`. In Code Snippet 13, avoiding the refresh is taken care of. Ensure the following in Code Snippet 13:

- Clicking the Submit button triggers the `onSubmit` event handler. Hence, attach the `onSubmit` event handler to the form.
- In the callback function of `onSubmit`, to avoid the refresh, use `event.preventDefault()`.
- To display the form values on the Console, use the state variables `email` and `password`.

Code Snippet 13:

```
function LoginForm() {
  const [email, setEmail] = useState("mark@gmail.com");
  const [password, setPassword] = useState("secret123");
  return (
    <form
      className="login-form"
      onSubmit={(event) => {
        event.preventDefault();
        console.log(email, password);
      }}
    >
      <h1>Login Form</h1>
      <input
        name="email"
        value={email}
        onChange={(event) => setEmail(event.target.value)}
        type="email"
        placeholder="username"
      />
```

```
      <input
        name="password"
        value={password}
        onChange={(event) => setPassword(event.target.value)}
        type="password"
        placeholder="password"
      />
      <button type="submit">Submit</button>
    </form>
  );
}
```

The output of Code Snippet 13 is shown in Figure 4.10.

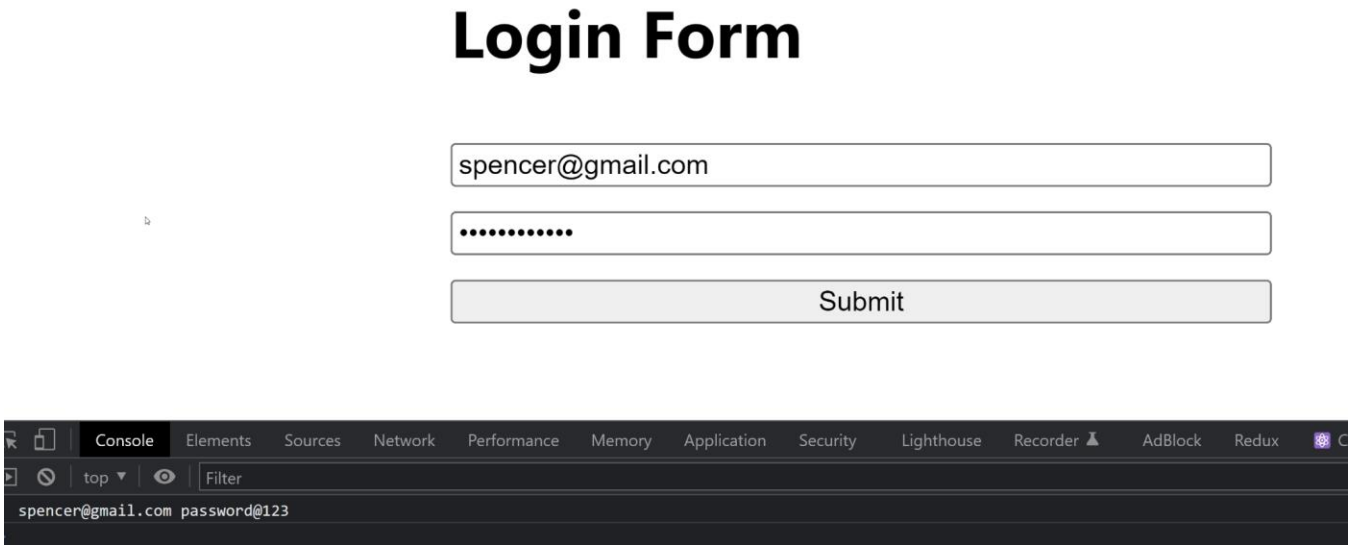


Figure 4.10: Output of Code Snippet 13

4.4.4 Touch Events

Nowadays, touch events are very popular and trending due to the rise of mobile devices. Consider an example to discuss the usage of touch events.

Example: Build a component to paint on the touched area.

Code Snippet 14 aims to position the inner division as absolute and change it into a circle with pink color. Code Snippet 14 is written in `App.js`.

Code Snippet 14:

```
function PaintDot() {
  const styles = {
    position: "absolute",
    top: "0px",
```

```
    left: "0px",
    height: "100px",
    aspectRatio: "1/1",
    borderRadius: "50%",
    background: "pink",
    transform: "translateX(-50%) translateY(-50%)",
  };
  return (
    <div className="dot-container">
      <div style={styles}></div>
    </div>
  );
}
```

Output of Code Snippet 14 is shown in Figure 4.11.

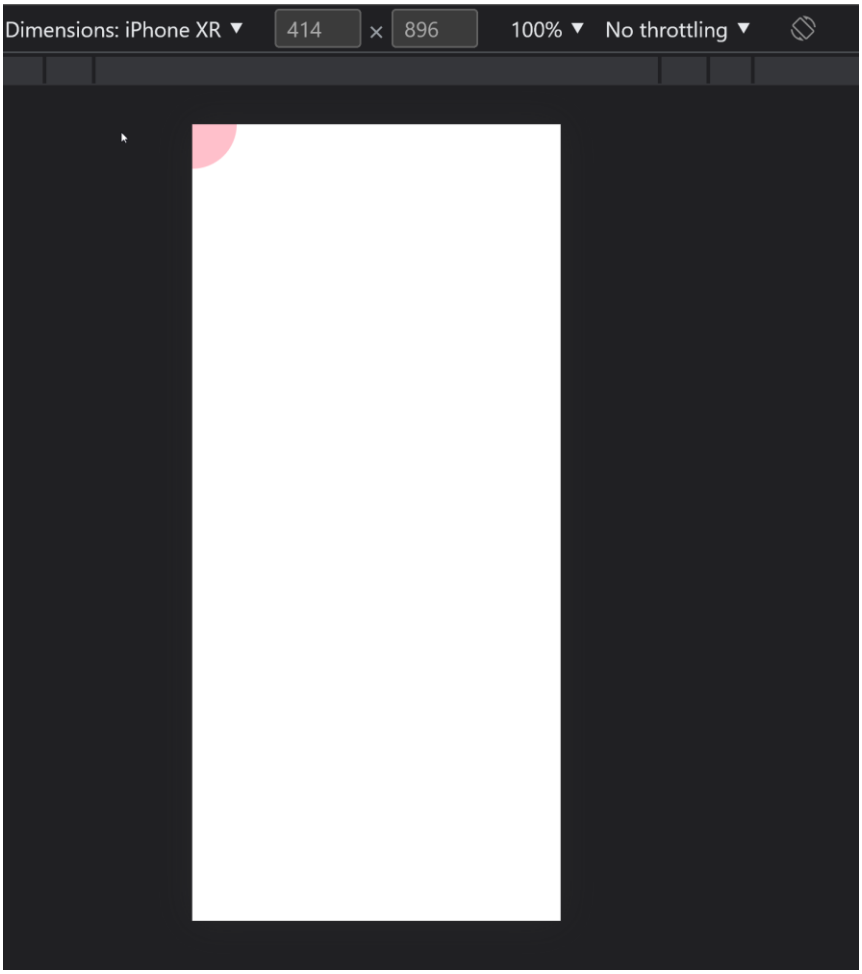


Figure 4.11: Output of Code Snippet 14

In Code Snippet 15, `onTouchStart` is used to track the event when the user touches the screen. The Code Snippet 15 is written in `App.js`.

Code Snippet 15:

```
function PaintDot() {
  const [pos, setPos] = useState({});
  const styles = {
    position: "absolute",
    top: "0px",
    left: "0px",
    height: "100px",
    aspectRatio: "1/1",
    borderRadius: "50%",
    background: "pink",
    transform: "translateX(-50%) translateY(-50%)",
  };
  return (
    <div
      onTouchStart={(event) => setPos(event.changedTouches[0])}
      className="dot-container"
    >
      <h1>X: {pos.pageX}</h1>
      <h1>Y: {pos.pageY}</h1>
      <div style={styles}></div>
    </div>
  );
}
```

Output of Code Snippet 15 is shown in Figure 4.12.

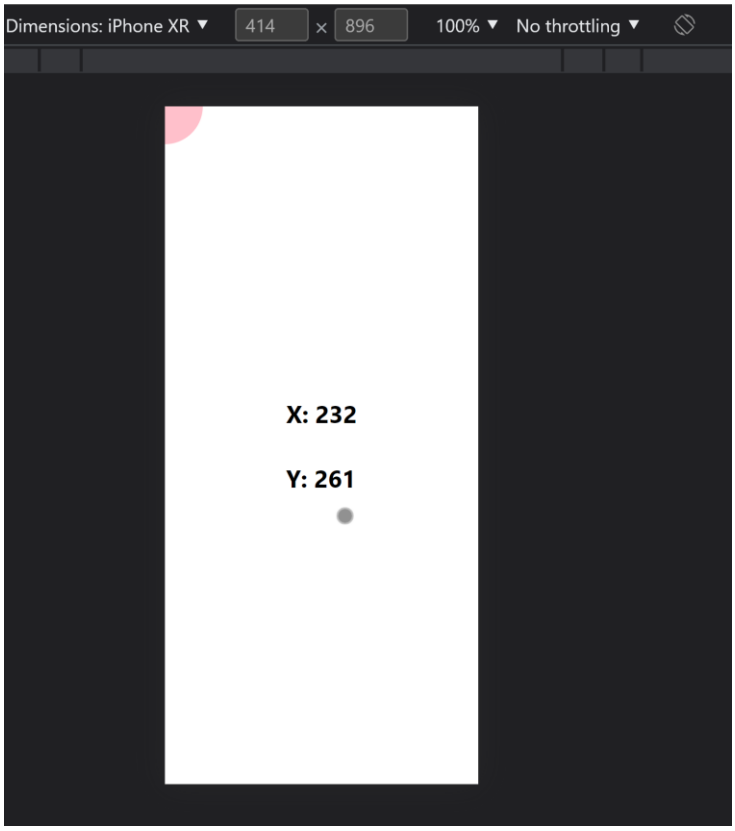


Figure 4.12: Output of Code Snippet 15

The code works in the following manner:

- When the user touches the screen, the `onTouchStart` event handler is triggered, which in turn calls the `setPos` function.
- Once `setPos` is called, ReactJS is notified that the `pos` value has to be updated and the view has to be re-rendered.
- `pos.pageX and pos.pageY are` updated and the view is also re-rendered, showing the X and Y co-ordinates of the touched position.
- The `event.changedTouches[0]` is used because multiple fingers can also be used, but tracking here is only for one finger touch.

However, the `paintDot` does not react to touch. Hence, to move it, as shown in Code Snippet 16**,** the top and left positions are updated with `pos.pageX` & `pos.pageY`. Code Snippet 17 is written in `App.js`.

Code Snippet 16:

```
function PaintDot() {
  const [pos, setPos] = useState({});
  const styles = {
    position: "absolute",
    top: `${pos.pageY}px`,
    left: `${pos.pageX}px`,
    height: "100px",
    aspectRatio: "1/1",
```

```
    borderRadius: "50%",
    background: "pink",
    transform: "translateX(-50%) translateY(-50%)",
  };
  return (
    <div
      onTouchStart={(event) => setPos(event.changedTouches[0])}
      className="dot-container"
    >
      <h1>X: {pos.pageX}</h1>
      <h1>Y: {pos.pageY}</h1>
      <div style={styles}></div>
    </div>
  );
}
```

The output of Code Snippet 16 is shown in Figure 4.13.

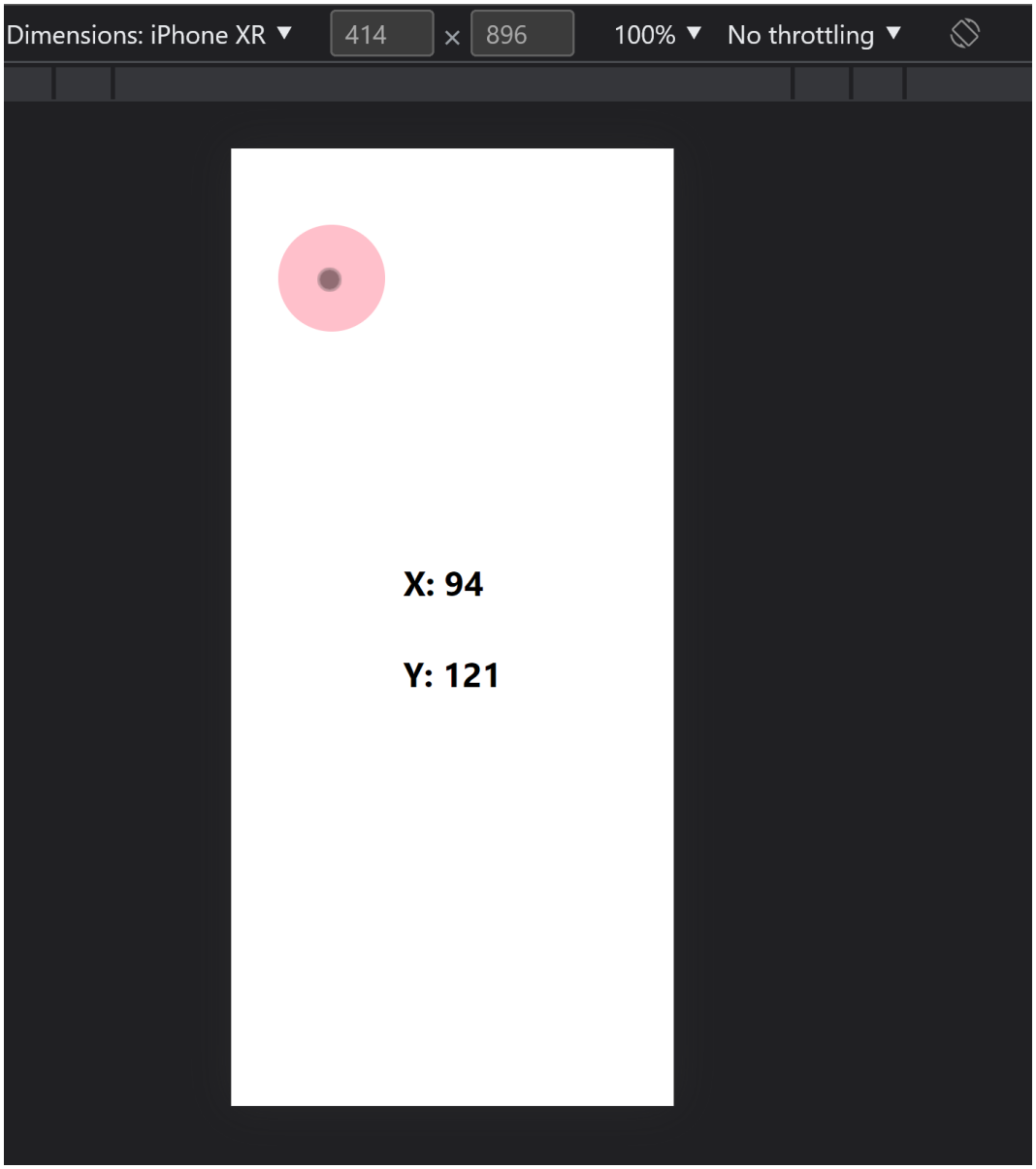


Figure 4.13: Output of Code Snippet 16

4.5 Handling Different Events

As discussed in Session 2, there is no direct way to pass data from child to parent. Hooks are a workaround to pass data from the child to parent. This is especially useful when one has to inform the parent component about changes in the child component. Consider an example to discuss this.

4.5.1 User Events

In Code Snippet 17, the `startGame` function requires to be called `onClick` of the `StartGame`.

The code works in the following manner:

- As props flow is unidirectional, which is from parent component to child component, the `startGame` function is passed as props to `onPlayerClick`, and called when the `onClick` of the button is triggered.
- The count on the view is updated and “Game is started” is printed on the console.

Code Snippet 17 is written in `App.js`.

Code Snippet 17:

```
function Game() {
  let [count, setCount] = useState(0);
  const startGame = () => {
    console.log("Game is started");
    setCount(count + 1);
  };
  return (
    <div>
      <h1>Game started {count} time</h1>
      <StartGame onPlayerClick={startGame} />
    </div>
  );
}
function StartGame({ onPlayerClick }) {
  return (
    <div>
      <button onClick={() => onPlayerClick()}>Start</button>
    </div>
  );
```

Output of Code Snippet 17 is shown in Figure 4.14.

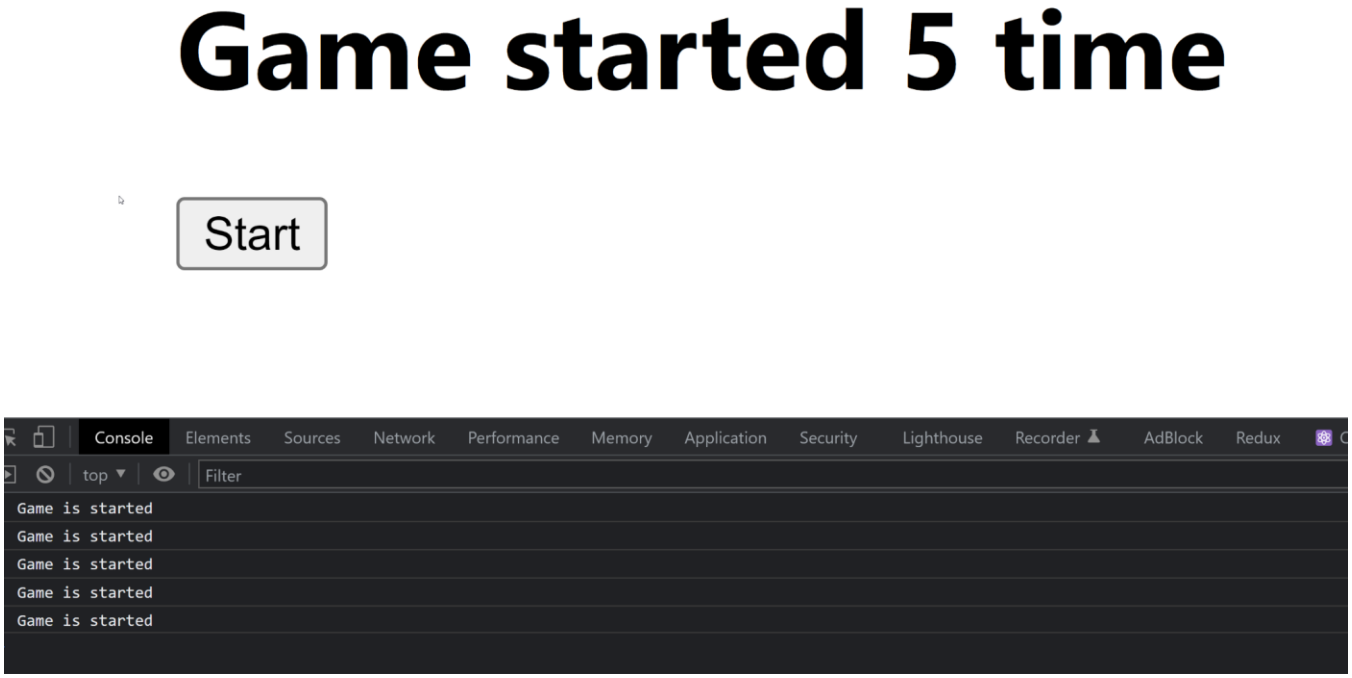


Figure 4.14: Output of Code Snippet 17

4.6 Summary

- ✓ Event handlers are attached to elements to add interactivity to the app.
- ✓ ReactJS listens to changes made by hooks.
- ✓ Hooks are functions that starts with the word, “use”.
- ✓ Event handlers must always be camelCased.
- ✓ `useState` variables are declared to update the view.
- ✓ `onClick`, `onSubmit`, `onChange`, and `onTouchStart` are event handlers.
- ✓ Function can be passed as props to call the parent function.

4.7 Check Your Knowledge

1. Is `onclick` a valid ReactJS event handler?

A	True
B	False

2. Functions can be passed as props.

A	True
B	False

3. _____ is a recommended callback function.

A	Normal function
B	IIFE
C	Arrow function
D	`ClickMe` function

4. _____ hook is required to update value on screen.

A	`useEffect`
B	`useState`
C	`useNavigate`
D	`useParams`

5. _____ event handler is used to listen to typing events.

A	`onClick`
B	`onTouchStart`
C	`onSubmit`
D	`onChange`

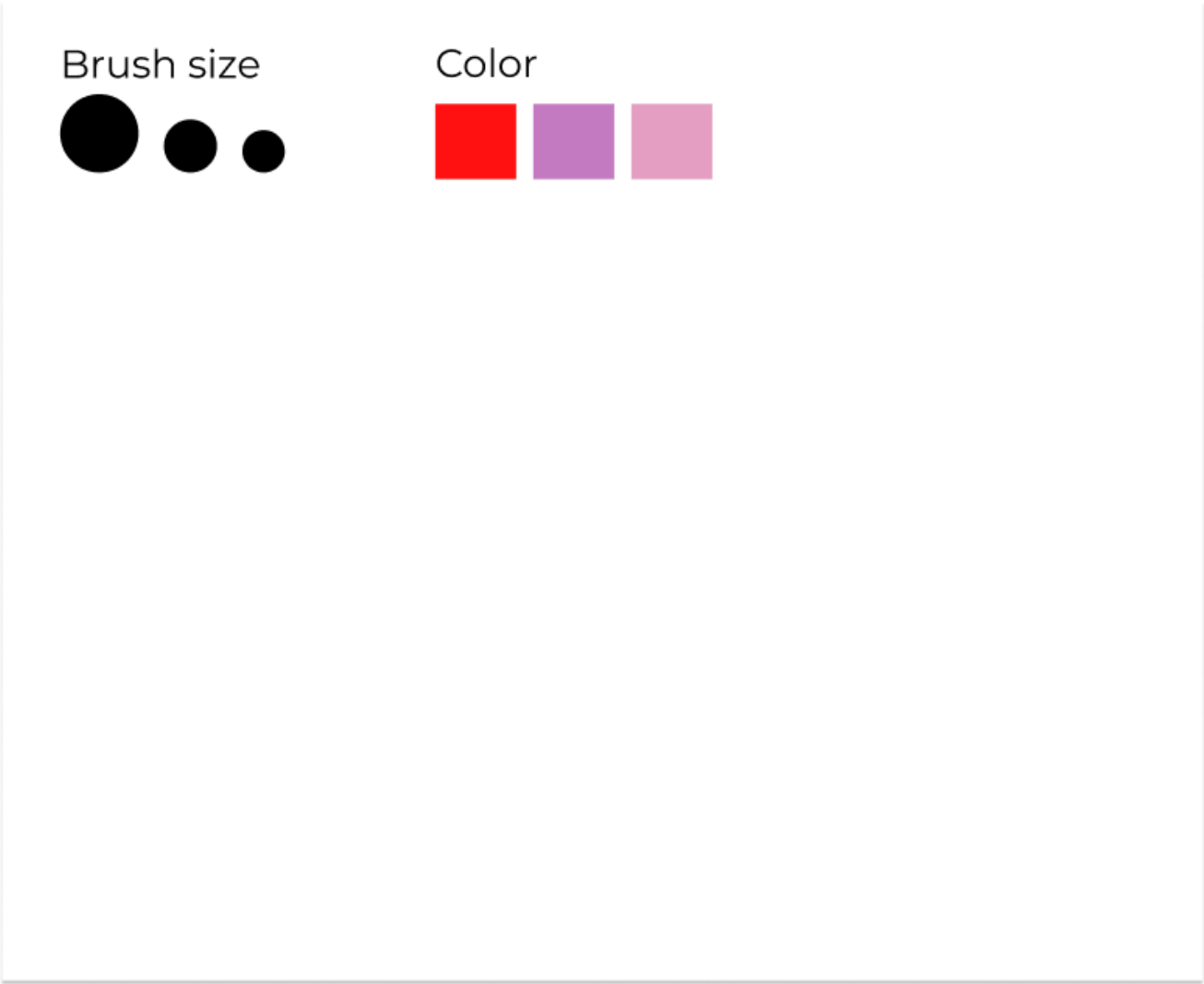
Answers

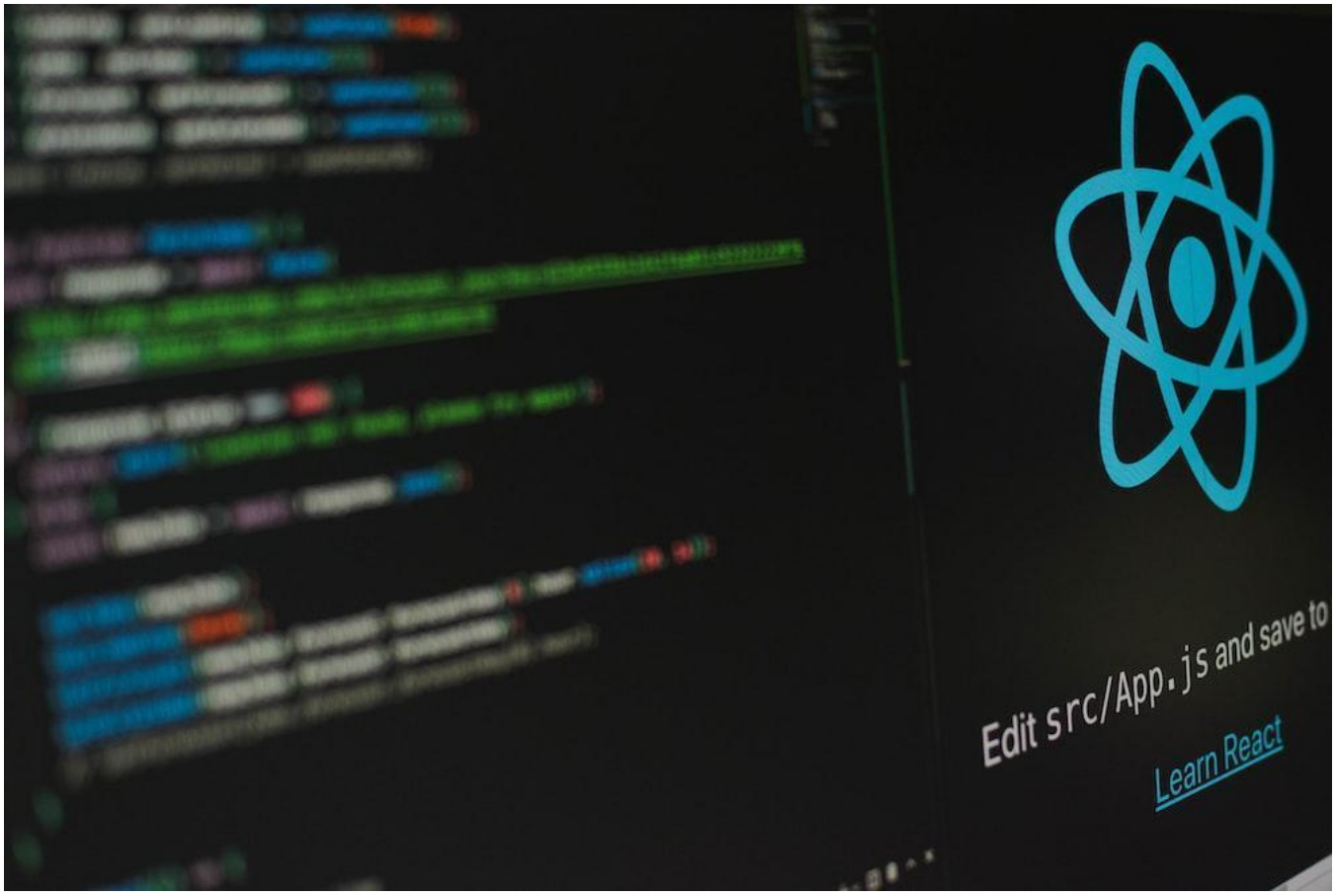
1	B
2	A
3	C
4	B
5	D

Try It Yourself

You are a game designer and developer. Design and develop a game where users can draw with fingers. Also, provide three different color choices and brush sizes for the user to choose from, as shown in the following Figure.

Hint: Use **onTouchMove** to implement the design.





Session 5
ReactJS Router

Session Overview

This session provides information about the ReactJS router, its basic components, and using routing modules. It also discusses the difference between single and multi-page applications.

Objectives

In this session, students will learn to:

- Explain about the ReactJS router and its basic components
- Explain the difference between single and multi-page applications
- Explain the advantages and drawbacks of single and multi-page applications

5.1 Single-Page Applications vs. Multi-Page Applications?

To understand the difference between Single-Page Applications (SPA) and Multi-Page Applications (MPA), observe the illustrations as shown in Figure 5.1.

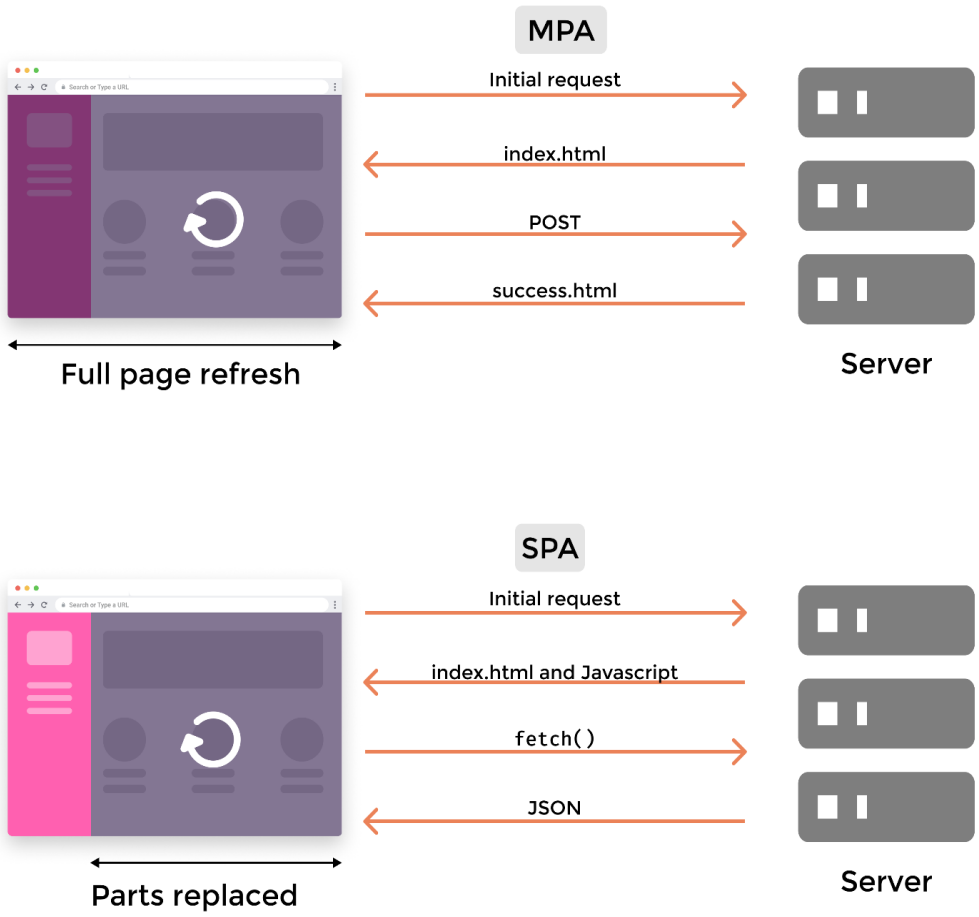


Figure 5.1: Working of SPA and MPA

MPA

The traditional Web app is known as MPA. This style of application works as shown in Figure 5.1. When the app has to load the Web page, the browser requests for the `index.html` file. When the user submits a form, a POST request is sent to the server. The server responds back with a new success.html file, which in turn wipes off all elements on the screen, and the browser starts rendering the page from scratch. In other words, the Web page loads every time the user submits a request.

Drawbacks of MPA

Some of the drawbacks of MPA are:

- Creates a blank screen for a few seconds depending on the Internet speed, which is undesirable.
- Even the common elements are recreated.
- Extra bandwidth of the user is spent because the common elements are loaded again.
- Loss in performance because the browser re-creates all the elements.

SPA

An SPA works as shown in Figure 5.1. To load the Web page, an initial request is made to the server. The server responds with an `index.html` and a JavaScript file. The JavaScript file is the core of the app because all the elements on the screen are created through JavaScript. When a form is submitted, the response comes in the JavaScript Object Notation (JSON) format. The entire page is not refreshed and only parts of the page that require to be changed are replaced.

> **Note**: AngularJS plays an important role in SPA. It enables pre-loading and caching of all pages. Hence, additional requests to download the pages are not required. AngularJS allows pre-loading of all relevant pages when the Internet connection is lost.

Advantages of SPA

Advantages of an SPA are:

No Full-page Refresh

A user navigates between pages without refreshing the whole page because only the required section of the page that requires a change is re-loaded.

Offline Functioning

An SPA allows navigating through its sections even if the Internet connection is suddenly lost. In SPA apps, the pre-loaded (previously visited) page could be re-used in the next visit. This will greatly improve the user experience when the Internet connection is unavailable.

Improved User Experience with Bandwidth Saving

An SPA gives a feel of a desktop application, which is fast and responsive.
Only the parts that require to be changed are requested and hence additional bandwidth is not used.

Drawbacks of SPA

Some of the drawbacks of SPA are:

Complex Development

A developer must write complex JavaScript coding to manage different entities and functionalities such as permissions and shared states across pages.

Additional Search Engine Optimization (SEO) Care

It is essential for a search engine crawler to execute JavaScript for indexing the built SPA application.
Recently, Bing and Google began executing JavaScript for indexing Asynchronous JavaScript and XML (AJAX) pages. This indicates that one would require static HTML views for search engines that have not started executing JavaScript.

Client-side JavaScript Disabled Possibility

It is necessary for a client to enable JavaScript on its browser because SPAs require it. If the client-side browser has JavaScript disabled, the SPA will not work. Fortunately, all modern browsers have JavaScript enabled.

Slow Initial Load

When loading for the first time, an SPA downloads more resources.
This downloading of resources makes the initial loading slow.

5.2 What is ReactJS Router?

Although the SPA style of application development was present before the advent of ReactJS, it was popularized by ReactJS. To bring out SPA in ReactJS, the `react-router-dom package` is used. The `react-router-dom` package ensures routing in ReactJS. Routing refers to the change of page in the SPA.

Why Routing?

Before deep dividing into the ReactJS Router, it is important to understand why routing is required in a ReactJS application.

Advantages of Routing

Advantages of routing are:

- The app looks organized.
- User can easily access different parts of the app swiftly.
- Users can bookmark the page for future reference.
- Users can spread the word by sharing the Uniform Resource Locator (URL) with others in social networks.
- The pages can be protected behind a login form.

5.2.1 Adding ReactJS Router to an Existing Project

To add a ReactJS Router to an existing project, install the necessary dependencies using `npm`. Hence, type the command in the command line of the project root path.

```
npm install react-router-dom@6
```

5.3 Basic Components of ReactJS Router

In this section, students will create a separate route for the user's page and home page. The pre-requisite for this is to have `UserList` and `User component` as shown in Code Snippet 1.

The `users` variable is passed as props to the `UserList` component. The User component is looped through and rendered on the screen.

Note: These techniques have already been discussed in Session 4.

Code Snippet 1 is written in `App.js`.

Code Snippet 1:

```
const USER_LIST = [

  {

    name: "Cuban",

    pic: "https://images.unsplash.com/photo-1618641986557-
1ecd230959aa?ixlib=rb-
4.0.3&ixid=MnwxMjA3fDB8MHxzZWFyY2h8NXx8cHJvZmlsZXxlbnwwfHwwfHw%3D&w=1000&q=80",

    bio: "Travel fan. Hipster-friendly tv scholar. Friendly communicator.
Coffee enthusiast.",

  },

  {

    name: "Spencer",

    pic: "https://images.unsplash.com/photo-1529665253569-
6d01c0eaf7b6?ixlib=rb-
4.0.3&ixid=MnwxMjA3fDB8MHxzZWFyY2h8NHx8cHJvZmlsZXxlbnwwfHwwfHw%3D&w=1000&q=80",

    bio: "Award-winning web lover. Thinker. Social media advocate. Creator.
Bacon scholar. Zombie geek",

  },

  {

    name: "Robert",

    pic: "https://encrypted-
tbn0.gstatic.com/images?q=tbn:ANd9GcT2C_xadF4WT19MkU5PpYyU8njyMgMIuttwXQ&usqp=C
AU",

    bio: "Professional communicator. Travel scholar. Friendly music junkie.
Hardcore zombie aficionado",

  },
```

```
  {
    name: "Einstein",
    pic: "https://media.istockphoto.com/id/1179420343/photo/smiling-man-outdoors-in-the-city.jpg?s=612x612&w=0&k=20&c=8l-gOboGEFSyCFXr09EguDmV0E0bFT5usAms1wyFBh8=",
    bio: "Typical travel guru. Friendly entrepreneur. Zombie expert. Thinker. Pop culture evangelist",
  },
];
function App() {
  const users = USER_LIST;
  return (
    <div className="App">
      <UserList users={users} />
    </div>
  );
}
function UserList({ users }) {
  return (
    <div className="user-list-container">
      {users.map((usr) => (
        <User name={usr.name} pic={usr.pic} />
      ))}
    </div>
  );
}
function User({ name, pic }) {
  return (
    <section className="user-container">
      <img className="user-profile-pic" src={pic} alt={name} />
      <h2 className="user-name">
        Hello, <span className="user-first-name">{name}</span> 🥦💪
      </h2>
    </section>
  );
```

```
}
```

Output of Code Snippet 1 is shown in Figure 5.2.

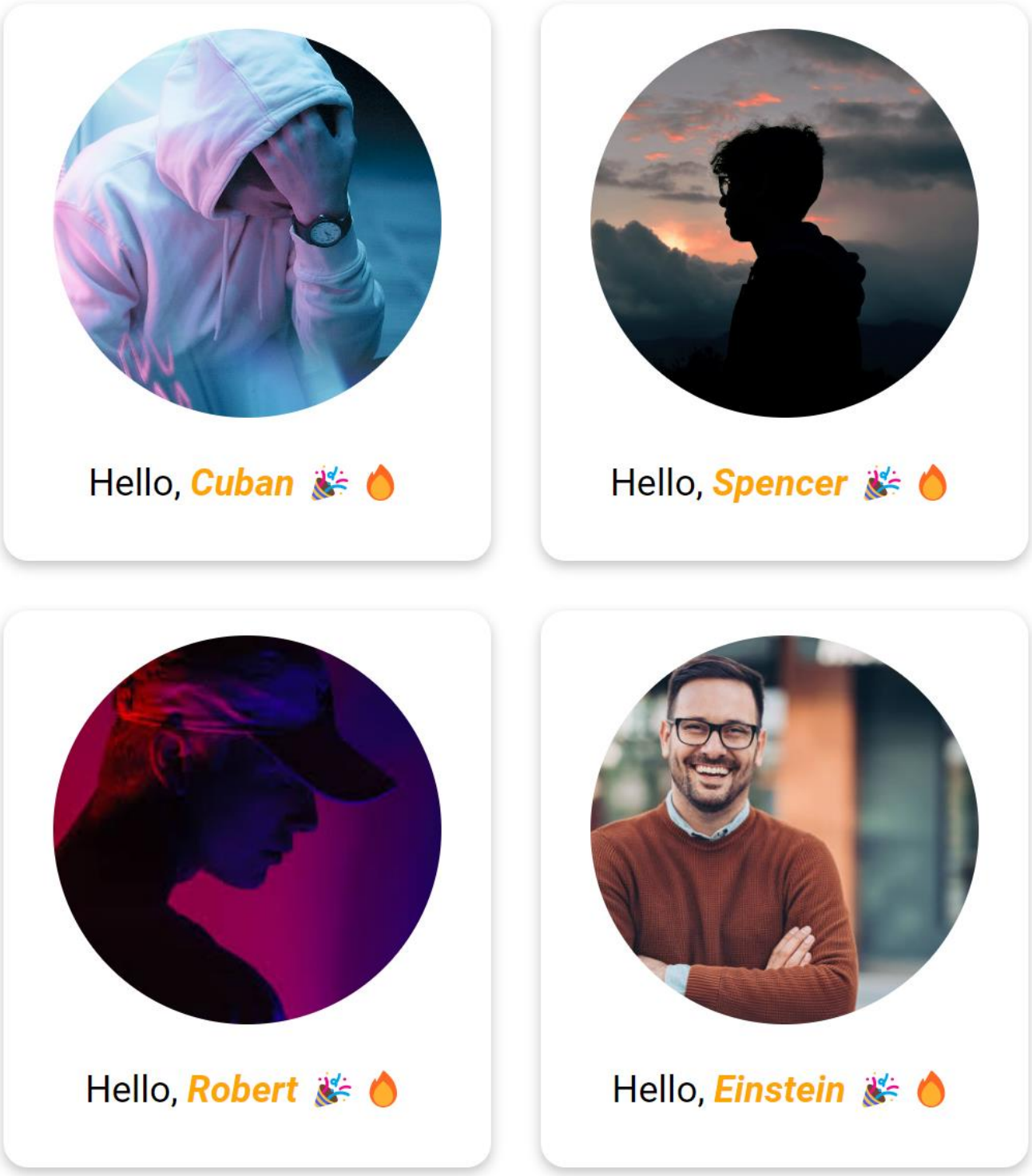


Figure 5.2: Output of Code Snippet 1

Styles to be used for the app are given in Code Snippet 2. The Code Snippet is written in `App.css`.

Code Snippet 2:

```
.App {
```

```
  text-align: left;
  margin: 0 auto;
  /* max-width: 400px; */
}

.nav-list {
  display: flex;
  gap: 16px;
  margin: 16px;
}
.home-container {
  text-align: center;
}
.user-name {
  font-family: Roboto;
  font-weight: normal;
}
.user-profile-pic {
  height: 250px;
  aspect-ratio: 1 / 1;
  border-radius: 50%;
  object-fit: cover;
}
.user-first-name {
  font-weight: bold;
  font-style: italic;
  color: orange;
}
.login-form {
  display: flex;
  flex-direction: column;
  gap: 12px;
  max-width: 400px;
  margin: 20px auto;
}
.dot-container {
  display: grid;
  place-content: center;
  min-height: 100vh;
}
.counter-container {
  display: flex;
  gap: 0.5rem;
  justify-content: center;
  margin-bottom: 1rem;
}
.counter-container > button {
  font-size: 20px;
}
.user-container {
  text-align: center;
  padding: 16px 32px;
  border-radius: 16px;
  box-shadow: rgba(0, 0, 0, 0.24) 0px 3px 8px;
  transition: 0.2s all;
```

```
  cursor: pointer;
}
.user-detail-container {
  padding: 16px 32px;
  border-radius: 16px;
  box-shadow: rgba(0, 0, 0, 0.24) 0px 3px 8px;
  display: flex;
  gap: 16px;
  max-width: 600px;
  margin: 0 auto;
}
.user-list-container {
  display: flex;
  flex-wrap: wrap;
  gap: 32px;
  align-items: center;
  justify-content: center;
  padding: 32px;
}
.user-container:hover {
  transform: scale(1.1);
}
```

Basic components of the ReactJS Router are:

- Router/ Browser Router
- Route
- Nav
- Link

5.3.1 Router/Browser Router

There are two types of routers in `react-router-dom`:

BrowserRouter

It is used with newer browsers because only in the newer browsers, the new features used by `BrowserRouter` are supported. Older browsers do not have these features.

HashRouter

`HashRouter` is used in older browsers like Internet explorer.

`HashRouter` router will work in both new and old browsers, but new features are unavailable.

The application being discussed in this session is only going to support newer browsers, `BrowserRouter` is imported and wrapped around the `App` component as shown in Code Snippet 3. Routing will start functioning in the application only after this setup is completed.

Code Snippet 3 is written in `index.js`.

Code Snippet 3:

```
import React from "react";
import ReactDOM from "react-dom/client";
import { BrowserRouter } from "react-router-dom";
import "./index.css";
import App from "./App";
import reportWebVitals from "./reportWebVitals";
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
  <React.StrictMode>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </React.StrictMode>
);
// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

5.3.2 Route

Route is the component that will the match the URL with the given component. In Code Snippet 5:

- `path="/"` indicates that if the URL provided is home page, then it will match the `<Home />` component.
- If the URL is `"/users"`, then it will match the `UserList` component.

Code Snippet 4 is written in `App.js`.

Code Snippet 4:

```
import "./App.css";
import { Routes, Route, Link } from "react-router-dom";
function App() {
  const users = USER_LIST;
  return (
```

```
    <div className="App">
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/users" element={<UserList users={users} />} />
      </Routes>
    </div>
  );
}
export default App;
```

Steps to test out the Code Snippet 4 in the browser are:

1. Type **`http://localhost:3000/`** ("/" at the end refers to the homepage).
2. Press the **Enter** key.
   The screen displays only the `Welcome` component and not the `UserList` component as shown in Figure 5.3.

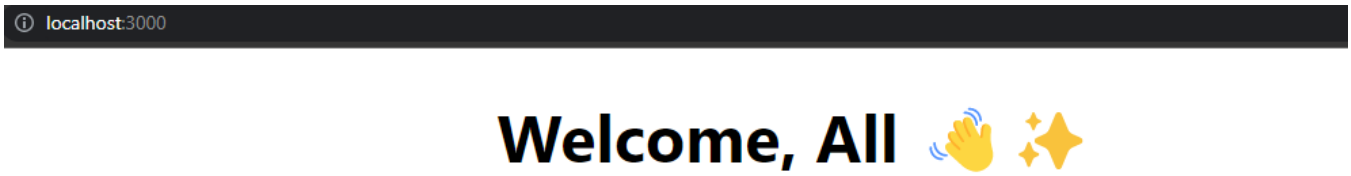


Figure 5.3: Output of Code Snippet 4 for URL: http://localhost:3000/

When the URL is changed to **`http://localhost:3000/users`** the screen displays only the `UserList` component and not the `Welcome` component as shown in Figure 5.4.

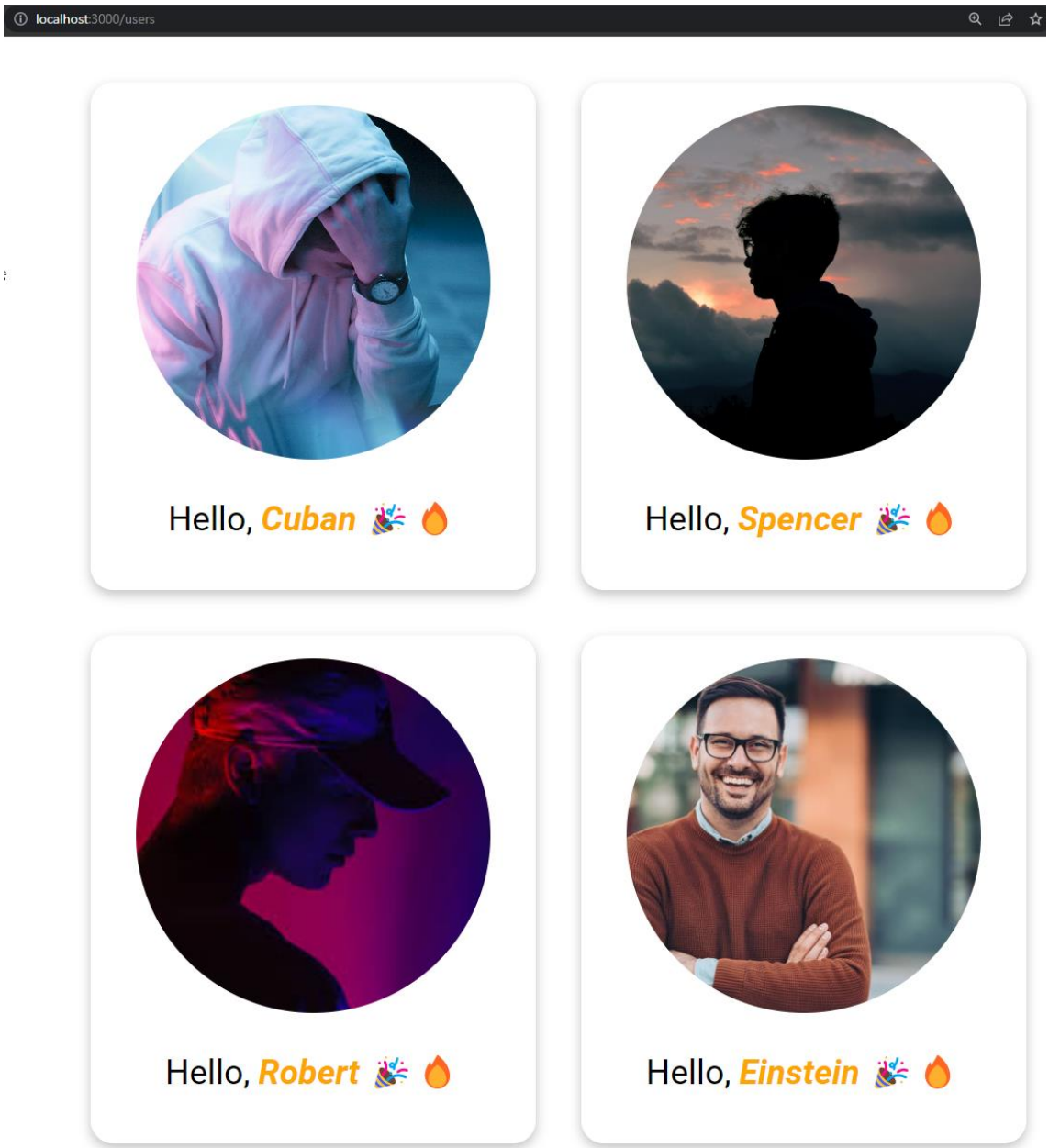


Figure 5.4: Output of Code Snippet 4 for URL: `http://localhost:3000/users`

5.3.3 Link

Navigating pages by typing the URL is inconvenient because users might not be able to remember all the links in the app. New users might find it difficult to discover the URLs or pages present in the app. Hence, the Link component was introduced to solve this problem as shown in Code Snippet 5. The Link component allows the user to navigate the application through the mouse click as shown in Figures 5.5 and 5.6.

Why the anchor tag (<a>) cannot be used instead of the Link component?

The anchor tag cannot be used because it will trigger a page refresh whereas the Link component will not refresh the page. Hence, the Link component improves the user experience.

The Code Snippet 5 is written in `App.js`.

Code Snippet 5:

```
function App() {
  const users = USER_LIST;
  return (
    <div className="App">
      <nav className="nav-list">
        <Link to="/">Home</Link>
        <Link to="/users">Users</Link>
        {/* Not recommended */}
        <a href="/users">Users with anchor</a>
      </nav>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/users" element={<UserList users={users} />} />
      </Routes>
    </div>
  );
}
```

The output of Code Snippet 5 is shown in Figure 5.5 and Figure 5.6.

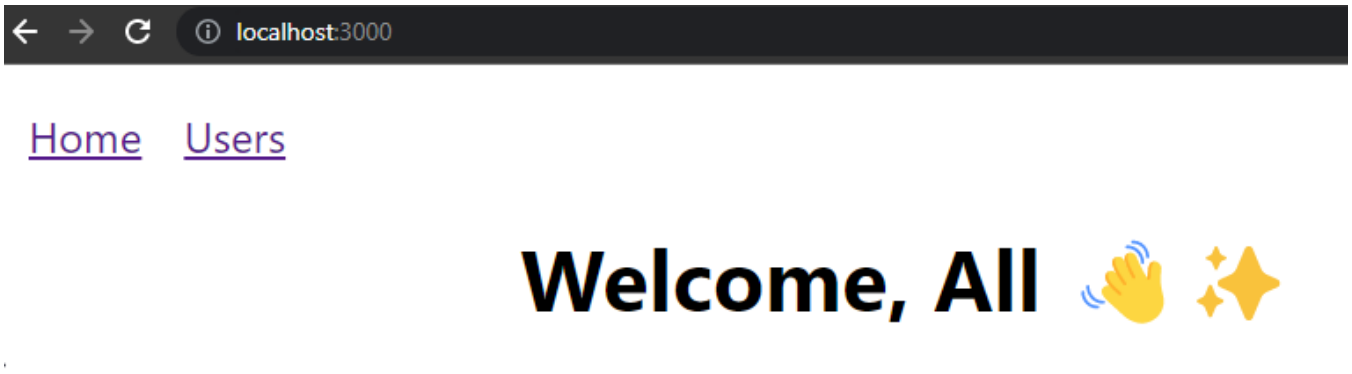


Figure 5.5: Output of Code Snippet 5 – Home Page

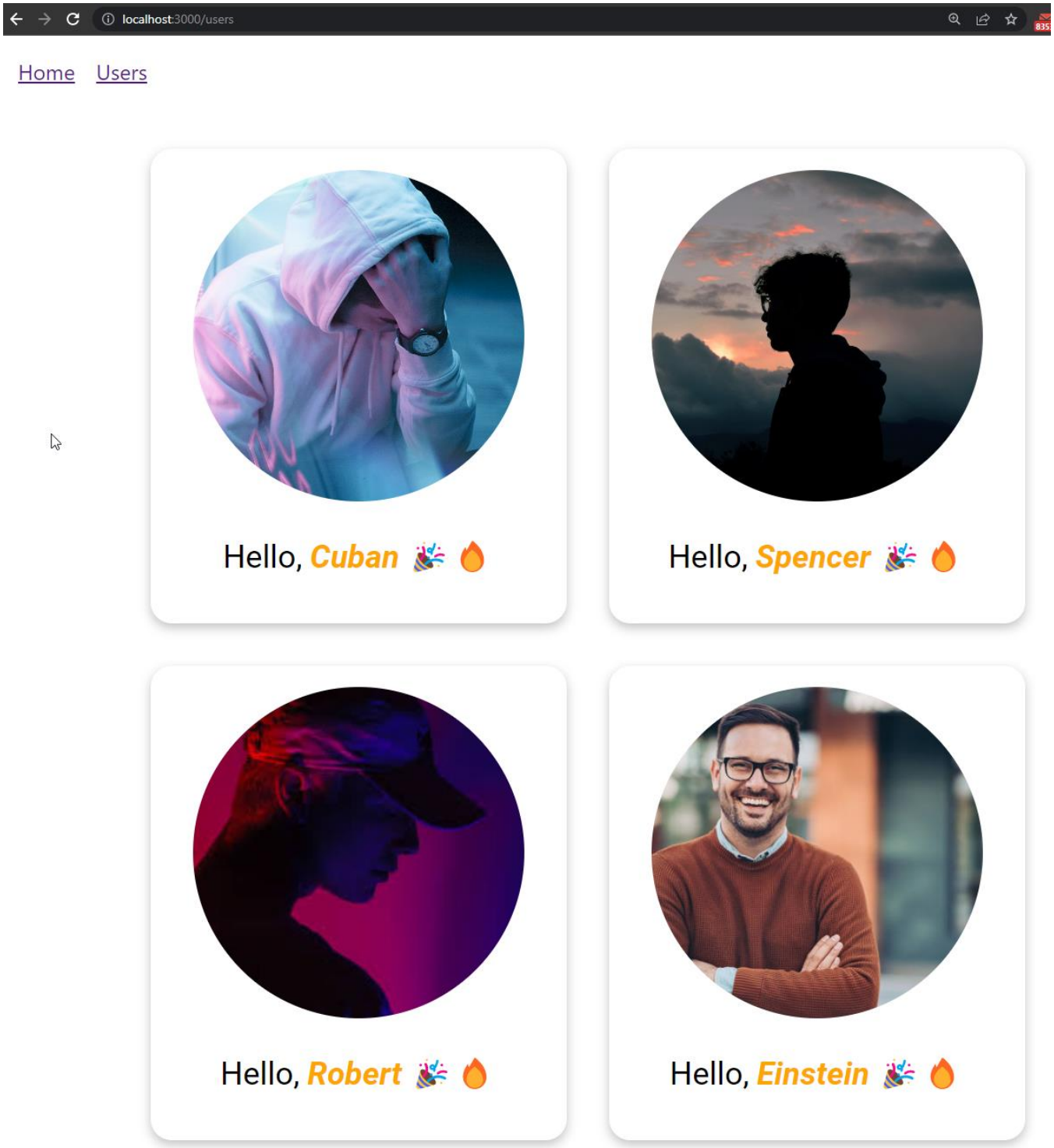


Figure 5.6: Output of Code Snippet 5 – Users Page

5.3.4 Nav

Many apps have a know more functionality integrated. For example: Consider an e-commerce app, wherein you search for a product and then navigate to the product details from the search results to know more details such as reviews, ratings, variants, colors available, similar products for the user, and so on. Consider implementing the same in the app, wherein when the user clicks on the user card, the user details page is displayed as shown in Figure 5.7. To address this scenario, add two new hooks, `useNavigate` and `useParams`.

`useNavigate`

It is used to change the URL programmatically. For example, when a payment is made on an e-commerce Website, the page automatically navigates to the success page or retry page depending on whether the payment was successful or not. Hence, the URL requires to be changed based on the logic without any user intervention. This is one of the best places to use, `useNavigate`. Another scenario would be when the styling must look like a button or a card and not like a Link. Here, instead of the Link component, `useNavigate` is used.

`useParams`

It is used to extract the parameter from the URL and pass it into the component. The extracted parameter would usually be an `id` of the product (in e-commerce apps). This `id` provides the gateway to extract more information of the product from a product list data. Hence, `useParams` is used in product details component to get the `id` from the URL, display the relevant information, and eventually helps in building the product details page.

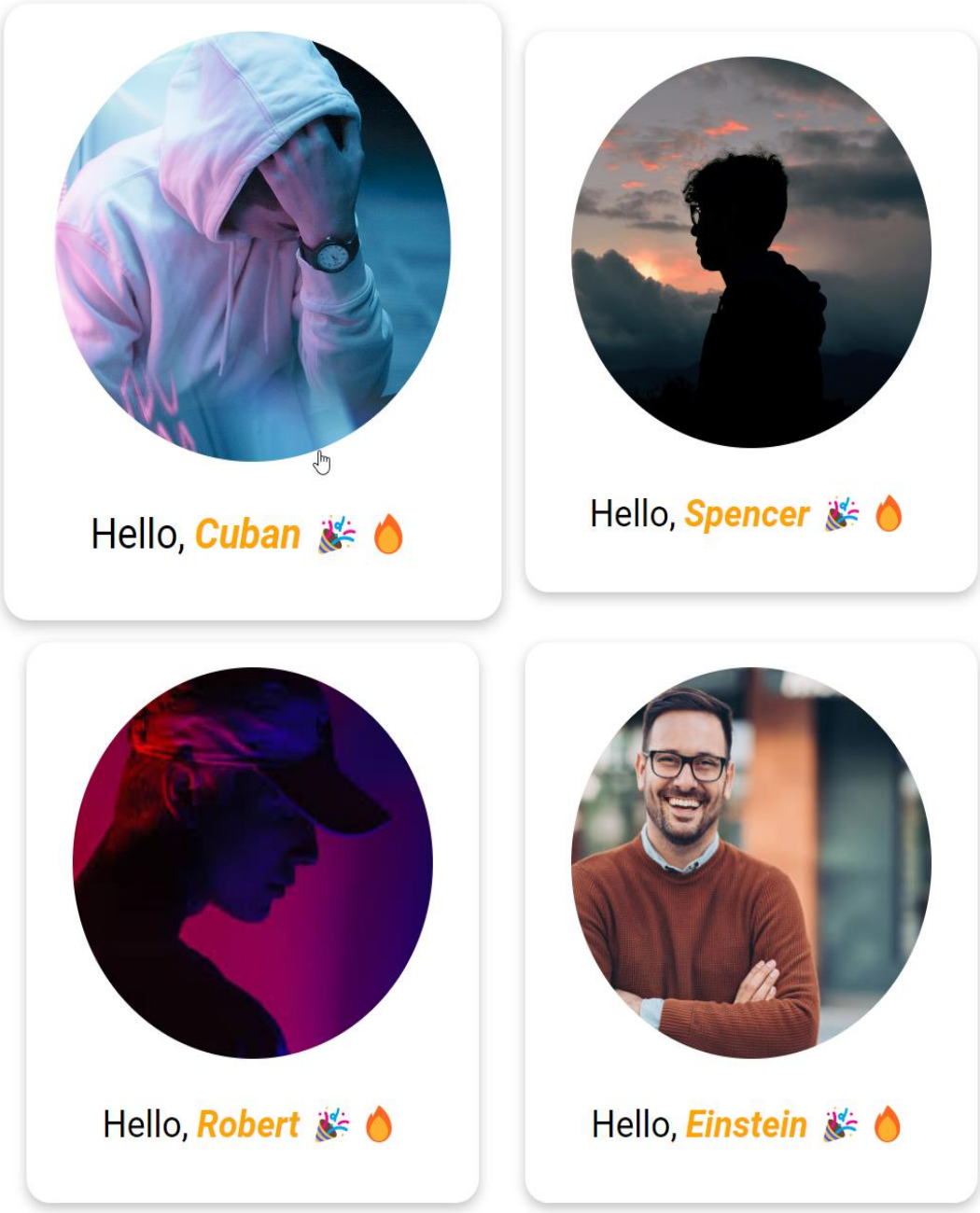


Figure 5.7: Navigating to User Cards

5.4 Using Routing Modules

To display the User details as a separate route, the first step is to start providing bio information to each user in `USER_LIST` as shown in Code Snippet 6. Code Snippet 6 is written in `App.js`.

Code Snippet 6:

```
const USER_LIST = [
  {
    name: "Cuban",
    pic: "https://images.unsplash.com/photo-1618641986557-1ecd230959aa?ixlib=rb-4.0.3&ixid=MnwxMjA3fDB8MHxzZWFyY2h8NXx8cHJvZmlsZXxlbnwwfHwwfHw%3D&w=1000&q=80",
    bio: "Travel fan. Hipster-friendly tv scholar. Friendly communicator. Coffee enthusiast.",
  },
  {
    name: "Spencer",
    pic: "https://images.unsplash.com/photo-1529665253569-6d01c0eaf7b6?ixlib=rb-4.0.3&ixid=MnwxMjA3fDB8MHxzZWFyY2h8NHx8cHJvZmlsZXxlbnwwfHwwfHw%3D&w=1000&q=80",
    bio: "Award-winning web lover. Thinker. Social media advocate. Creator. Bacon scholar. Zombie geek",
  },
  {
    name: "Robert",
    pic: "https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcT2C_xadF4WT19MkU5PpYyU8njyMgMIuttwXQ&usqp=CAU",
    bio: "Professional communicator. Travel scholar. Friendly music junkie. Hardcore zombie aficionado",
  },
  {
    name: "Einstein",
```

```
    pic: "https://media.istockphoto.com/id/1179420343/photo/smiling-man-outdoors-in-the-city.jpg?s=612x612&w=0&k=20&c=8l-gOboGEFSyCFXr09EguDmV0E0bFT5usAms1wyFBh8=",

    bio: "Typical travel guru. Friendly entrepreneur. Zombie expert. Thinker. Pop culture evangelist",

  },

];
```

5.4.1 Creating Routes

For the route setup, the requirement is:

- When the user visits `/users/0`, URL details of the first user must be displayed.
- When the user visits `/users/1`, the details of the second user must be displayed.
- When the user visits `/users/2`, the details of the third user must be displayed, and so on.

Hence, the Route path has to match a dynamic path: `/users/x`, where `/x` is dynamic. Thus, to make the route path a variable, ":" is added. Hence, path is given as path="`/users/:id`" as shown in Code Snippet 7. The "`:id`" specifies that the `id` is a variable and where the number given in the URL will be stored in it.

Code Snippet 7 is written in `App.js`.

Code Snippet 7:

```
function App() {

  const users = USER_LIST;

  return (

    <div className="App">

      <nav className="nav-list">

        <Link to="/">Home</Link>

        <Link to="/users">Users</Link>

      </nav>

      <Routes>

        <Route path="/" element={<Home />} />

        <Route path="/users" element={<UserList users={users} />} />

        <Route path="/users/:id" element={<UserDetail users={users} />} />

      </Routes>

    </div>

  );
```

```
}
```

Pass the index of the user in the URL and utilize the same in the `UserDetail` component. Hence, when a user provides the URL as http://localhost:3000/users/0 as shown in Figure 5.8:

- The index 0 is stored in "`:id`" and is extracted through `useParams()` hook in the `UserDetail` component as shown in Code Snippet 8.
- The `useParams()` hook returns an object, which is destructed as the id variable used in `UserDetail`.
- The matching user is extracted passing the `id` as index to users.

After the user details are extracted, the user details like picture, name, and bio are shown on the screen.

Code Snippet 8 is written in `App.js`.

Code Snippet 8:

```
import {
  Routes,
  Route,
  Link,
  useParams,
} from "react-router-dom";
function UserDetail({ users }) {

  const { id } = useParams();

  console.log(id);

  const user = users[id];

  return (

    <section className="user-detail-container">

      <img className="user-profile-pic" src={user.pic} alt={user.name} />

      <div>

        <h2 className="user-name">{user.name}</h2>

        <p>{user.bio}</p>

      </div>

    </section>

  );

}
```

Output of Code Snippet 8 is shown in Figure 5.9.

Figure 5.8: URL Provided By the User

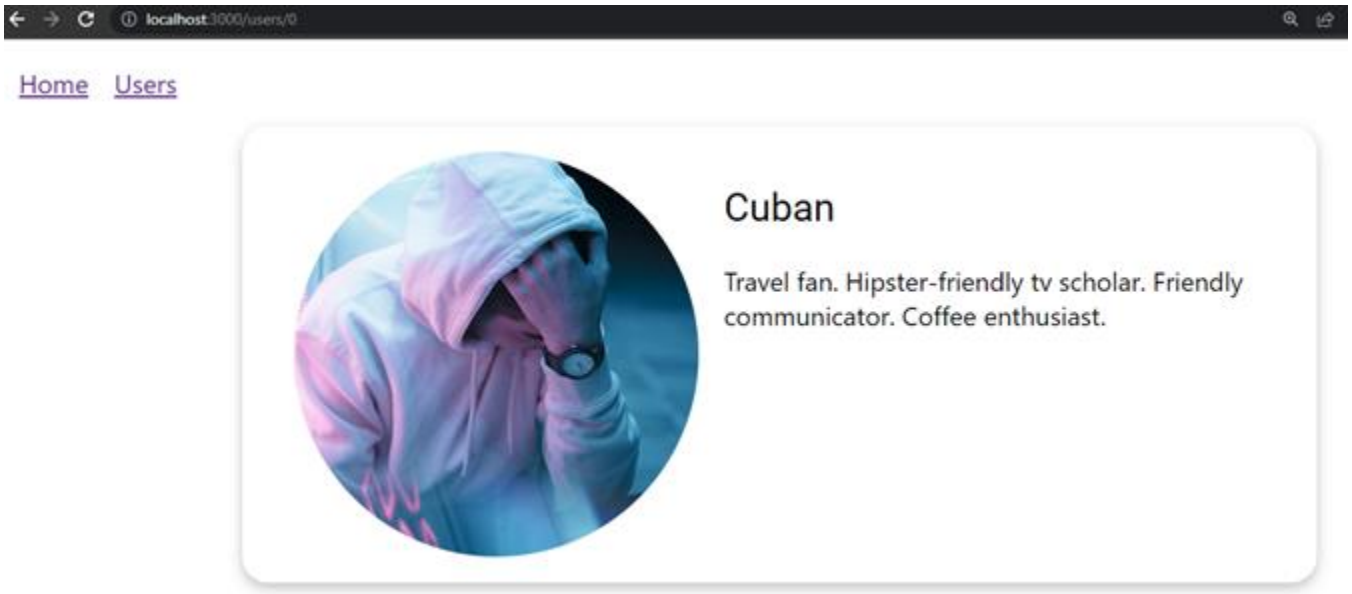


Figure 5.9: Output of Code Snippet 8

5.4.2 Navigating between Different Routes

To improve the user experience, consider showing the user details as shown in Figure 5.10, when the user card is clicked.

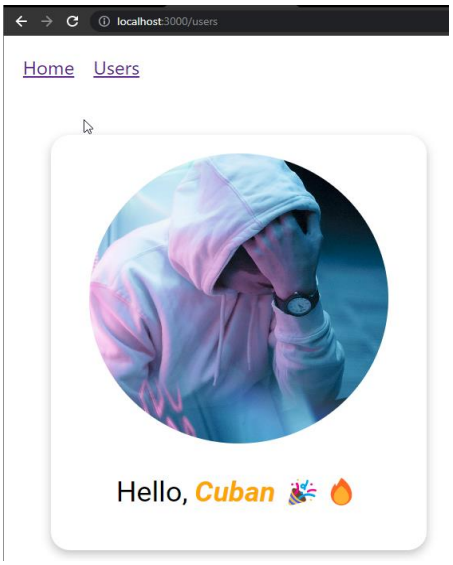


Figure 5.10: User Details

Hence, to achieve the same, user `index` is passed as `id prop` to the User component as shown in Code Snippet 9.

Code Snippet 9 is written in `App.js`.

Code Snippet 9:

```
function UserList({ users }) {

  return (

    <div className="user-list-container">

      {users.map((usr, index) => (

        <User name={usr.name} pic={usr.pic} id={index} />

      ))}

    </div>

  );

}
```

The `onClick` event handler is attached to the `section`. Hence, when the user clicks the first user card, the URL will be updated to "`/users/0`" because `navigate` (from `useNavigate()` hook) is used as shown in Code Snippet 10. Code Snippet is written in `App.js`.

Code Snippet 10:

```
import {
  Routes,
  Route,
  Link,
  useNavigate,
  useParams,
} from "react-router-dom";
function User({ name, pic, id }) {

  const navigate = useNavigate();

  return (

    <section

      onClick={() => navigate(`/users/${id}`)}

      className="user-container"

    >

      <img className="user-profile-pic" src={pic} alt={name} />

      <h2 className="user-name">

        Hello, <span className="user-first-name">{name}</span> 🎉🔥

      </h2>

    </section>
```

```
  );
}
```

Output of Code Snippet 10 is shown in Figure 5.11.

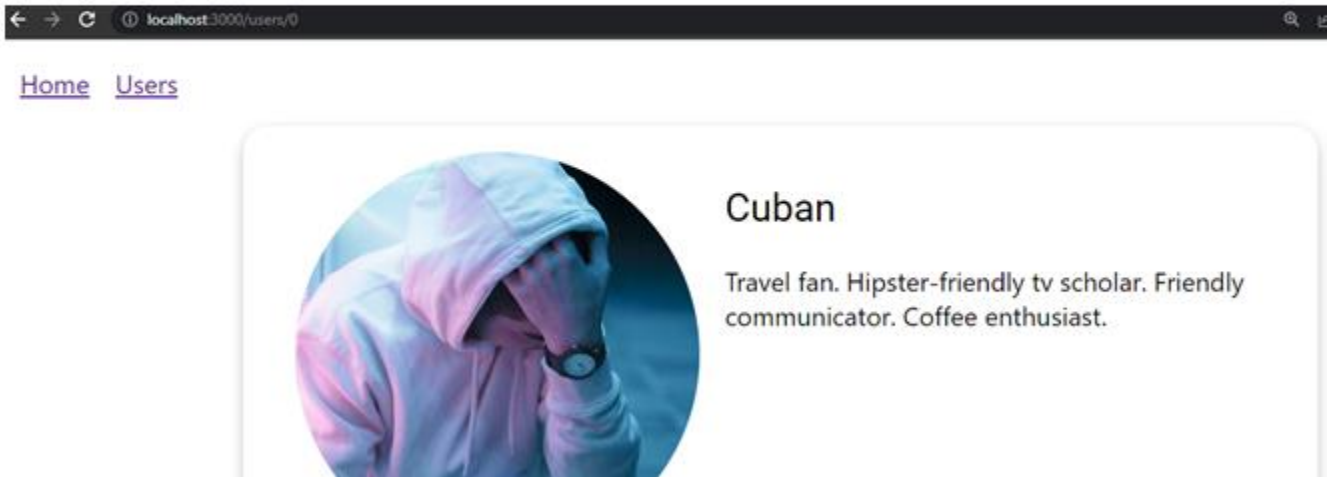


Figure 5.11: Output of Code Snippet 10

Not Found Page

When a user types a random URL or tries navigating to a page, a “404 Not found” message must appear.

To achieve this result, in the Routing setup, give `path="*"` to match all paths, if all other paths fail to match.

```
<Route path="*" element={<NotFound />} />
```

The Code Snippet 11 is written in `App.js`.

Code Snippet 11:

```
function NotFound() {
  return (
    <div>
      <h1>404 Not found</h1>
    </div>
  );
}
```

Hence, the `NotFound` component is displayed when the user types `http://localhost:3000/random`.

The output of Code Snippet 11 is shown in Figure 5.12.

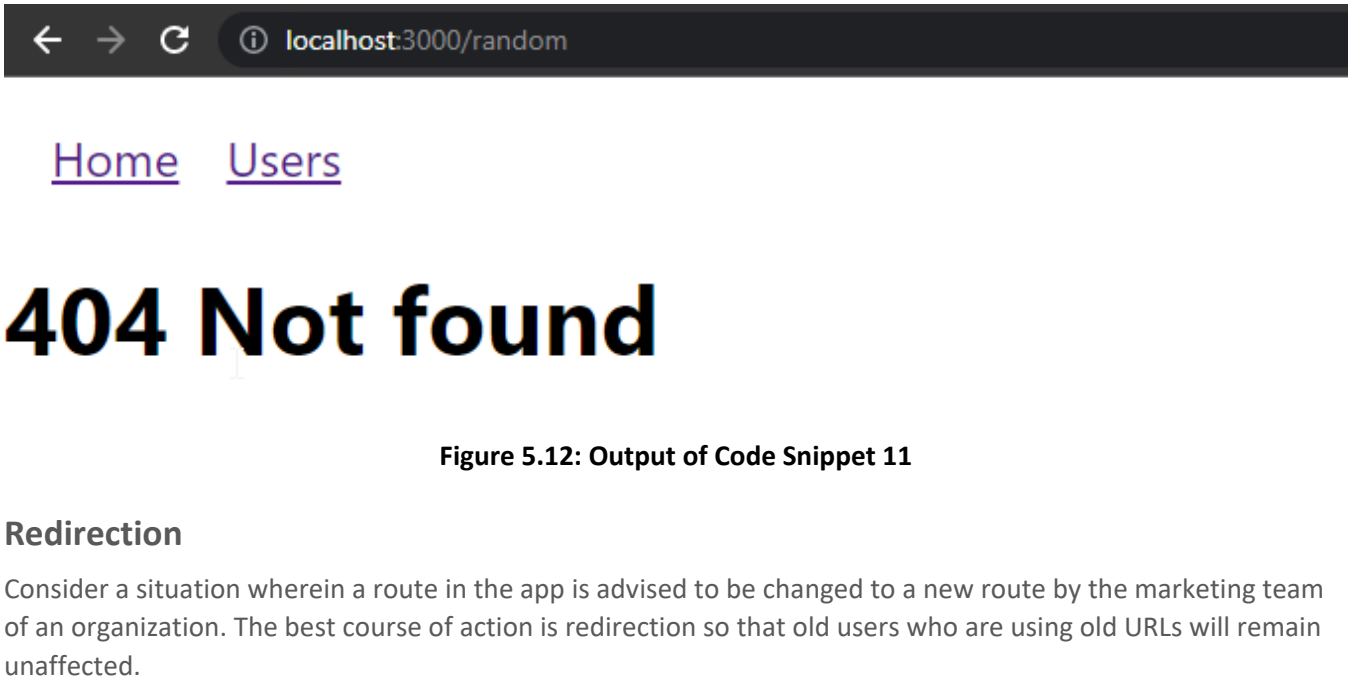


Figure 5.12: Output of Code Snippet 11

Redirection

Consider a situation wherein a route in the app is advised to be changed to a new route by the marketing team of an organization. The best course of action is redirection so that old users who are using old URLs will remain unaffected.

Hence, when "`/people`" as shown in Figure 5.13 has to be changed to "`/users`", use `Navigate` and `replace` as shown in Code Snippet 12 (written in `App.js`) to redirect users to "`/users`". Old users will be redirected accordingly, will also come to know the new route, and use the same route going forward.

Code Snippet 12:

```
 import {
  Routes,
  Route,
  Link,
  useNavigate,
  useParams,
} from "react-router-dom";
 <Route path="/people" element={<Navigate replace to="/users" />} />
```

Output of Code Snippet 12 is shown in Figure 5.14.

Figure 5.13: URL to be Changed

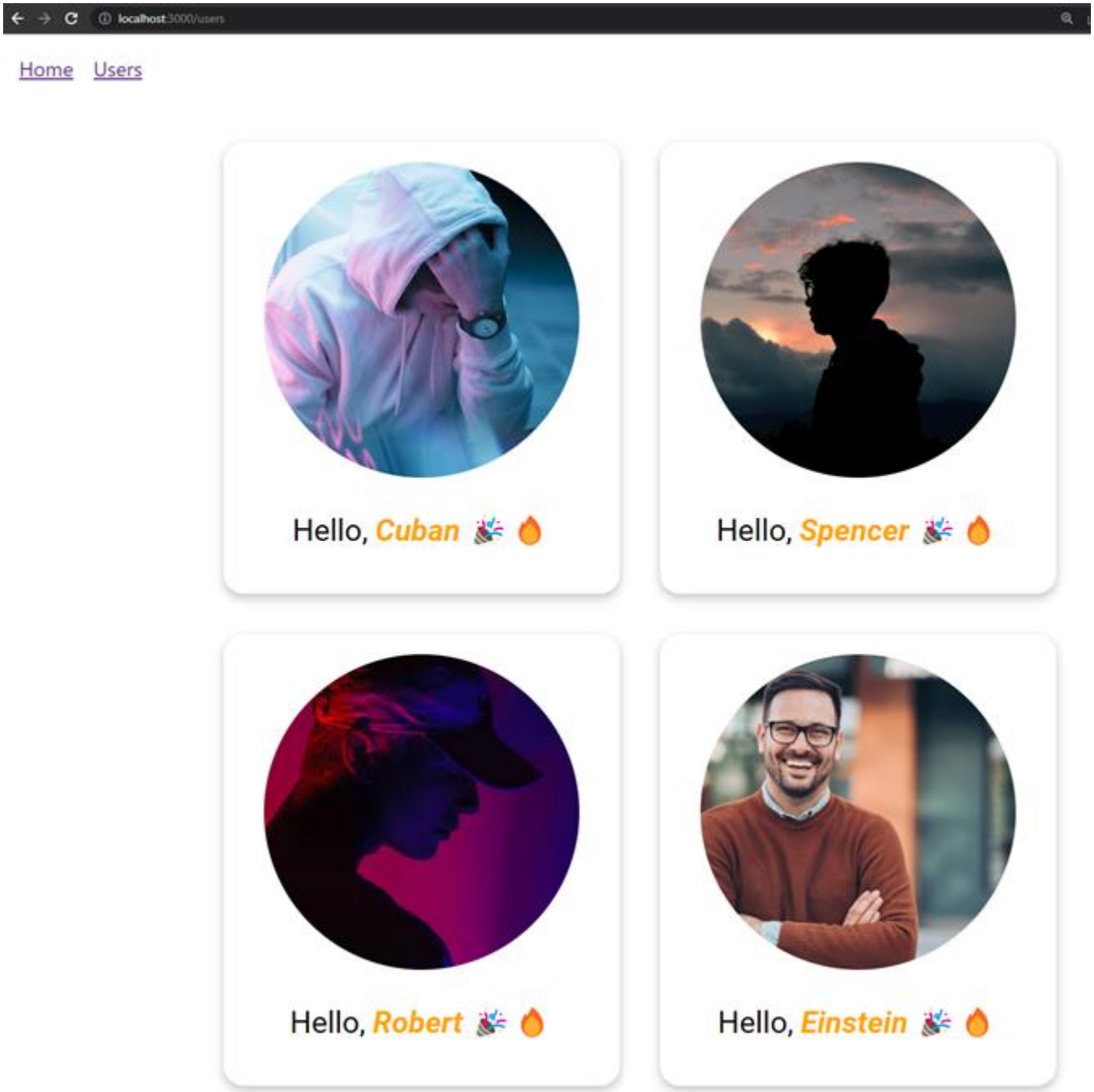


Figure 5.14: Output of Code Snippet 12

5.5 Summary

- ✓ SPA – Single Page Application, which is popularized by ReactJS.
- ✓ SPA is fast and does not require refresh.
- ✓ Two types of routers are Browser and Hash.
- ✓ Link component causes a no refresh.
- ✓ `useNavigate` – Programmatically changes URL.
- ✓ `useParams` – Extracts the value from the URL.
- ✓ `path=*` for matching 404 page.
- ✓ Redirection is achieved by the Navigate component.

5.6 Check Your Knowledge

1. Does `HashRouter` work in old browsers?

A	True
B	False

2. _____ is used to extract parameters from the URL.

A	`useNavigate()`
B	`useState()`
C	`useEffect()`
D	`useParams()`

3. _____ helps in redirection of a page.

A	Link
B	Route
C	Routes
D	Navigate

4. Anchor tag will not cause refresh.

A	True
B	False

5. _____ matches the URL with the component.

A	Link
B	Route
C	Routes
D	Navigate

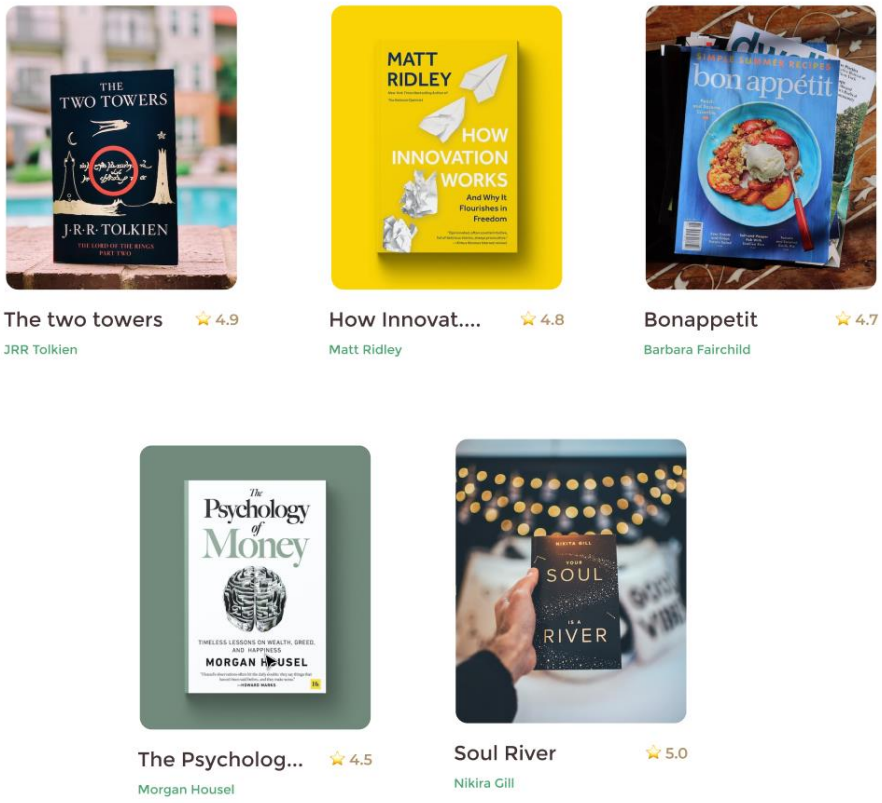
Answers

1	B
2	D
3	D
4	B
5	B

Try It Yourself

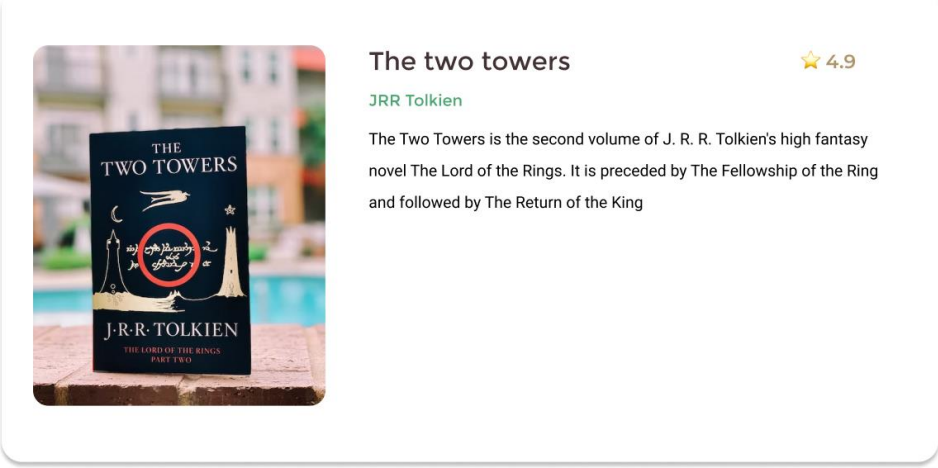
You are a developer and must build an e-book library management app. In this app, the Books Page must display all the available books as shown in the following Figure.

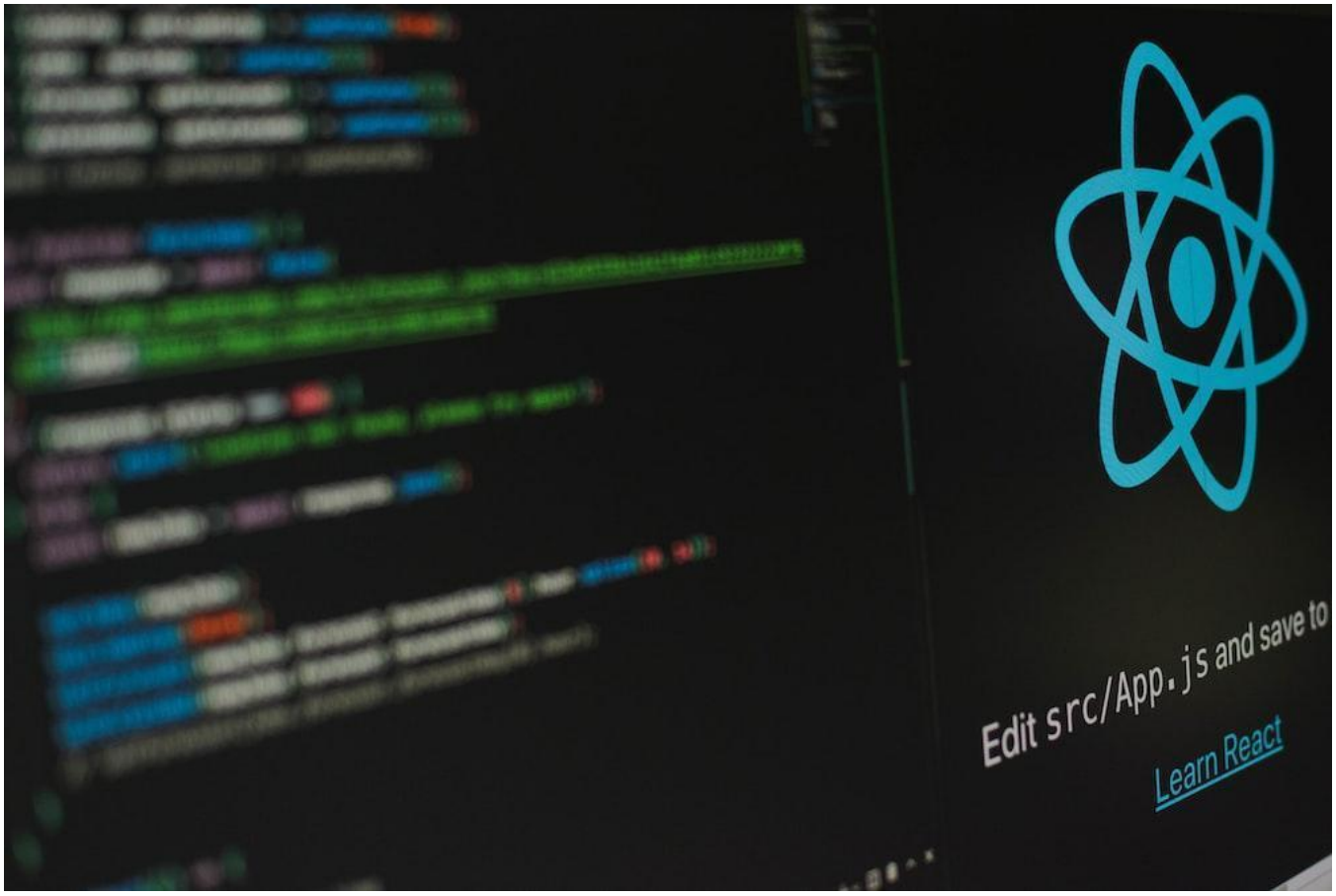
Books Page



When users click any of the books, it must navigate to the Book Detail page as shown in the following Figure.

Book Detail page





Session 6

Styling React Elements

Session Overview

This session provides information about the ways of applying different styles to HTML elements. It also provides information about styling methods in ReactJS.

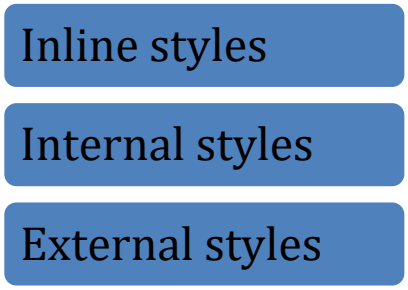
Objectives

In this session, students will learn to:

- Explain inline, internal, and external styles
- Explain the different styling methods in ReactJS
- Illustrate the usage of different styling methods

6.1 Applying Basic Style to HTML Elements

There are three main styles that are applied in HTML by using CSS:



6.1.1 Inline Styles

In inline styles, the styles are given directly to the style attribute as shown in Code Snippet 1. Following are specified in the code snippet:

- The image is given an aspect-ratio: 1/1 to keep the height and width the same.
- The image is surrounded by a border using a border-radius: 50%.
- To get a perfect circle, the height and width must be equal.
- The object-fit: cover is given to zoom in on the image so that it does not look like it is stretched.

Code Snippet 1 is written in `thereways.html`.

Code Snippet 1:

```
<!DOCTYPE html>
<html>
  <head>
    <title>There ways of styling</title>
    <meta charset="UTF-8" />
  </head>
  <body>
    <img
      style="
        height: 200px;
        aspect-ratio: 1/1;
        border-radius: 50%;
        object-fit: cover;
      "
      src="https://images.pexels.com/photos/1704488/pexels-photo-
1704488.jpeg?auto=compress&cs=tinysrgb&dpr=1&w=500"
      alt="Mark"
```

```
    />
  </body>
</html>
```

Output of Code Snippet 1 is shown in Figure 6.1.

Figure 6.1: Output of Code Snippet 1

6.1.2 Internal Styles

In internal styles, the style tag is declared in the `<head>` and styles are provided as a `class` or as an `id`. As shown in Code Snippet 2, the `.profile-pic` class is used for styling the image. Code Snippet 2 is written in `thereways.html`.

Code Snippet 2:

```
<!DOCTYPE html>
<html>
  <head>
    <title>There ways of styling</title>
    <meta charset="UTF-8" />
    <style>
      .profile-pic {
        height: 200px;
        aspect-ratio: 1/1;
```

```
        border-radius: 50%;
        object-fit: cover;
      }
    </style>
  </head>
  <body>
    <img
      class="profile-pic"
      src="https://images.pexels.com/photos/1704488/pexels-photo-
1704488.jpeg?auto=compress&cs=tinysrgb&dpr=1&w=500"
      alt="Mark"
    />
  </body>
</html>
```

6.1.3 External Styles

In external styles, the styles are provided as a separate file, and styles are provided as a class or as an id. In external files, students must use the following process:

- Use the `.profile-pic` class for styling the image and declare it in a separate file named `style.css` as shown in Code Snippet 4.
- Link this separate file by the `<link>` tag in the `<head>` of the HTML file as shown in Code Snippet 3.

Code Snippet 3 is written in `thereways.html`.

Code Snippet 3:

```
<!DOCTYPE html>
<html>
  <head>
    <title>There ways of styling</title>
    <meta charset="UTF-8" />
    <link rel="stylesheet" href="./style.css" />
  </head>
  <body>
    <img
      class="profile-pic"
      src="https://images.pexels.com/photos/1704488/pexels-photo-
1704488.jpeg?auto=compress&cs=tinysrgb&dpr=1&w=500"
      alt="Mark"
    />
  </body>
```

```
</html>
```

Code Snippet 4:

```
.profile-pic {
  height: 200px;
  aspect-ratio: 1/1;
  border-radius: 50%;
  object-fit: cover;
}
```

Recommended Style

Among all the three styles, the External style is recommended due to the following reasons:

- Inline styles are hard to override.
- Internal styles are used in the scenario only when critical style has to be sent, whenever the HTML file is loaded.
- External styles follow the separation of concern, which states that the HTML and the CSS code must be separated.

6.2 Different Styling Methods in ReactJS

In this section, students will learn how to use the styles in ReactJS.

6.2.1 Plain Inline Styling/Using Style Objects

In ReactJS, the style attribute cannot be defined as shown in Code Snippet 5 because it will result in an error, as shown in Figure 6.2. This error occurs because ReactJS expects styles in the form of JavaScript objects. Hence, the code must be converted as shown in Code Snippet 6.

Code Snippet 5 is written in `App.js`.

Code Snippet 5:

```
function UserPic() {
  return (
    <div>
      <img
        style="
        height: 200px;
        aspect-ratio: 1/1;
        border-radius: 50%;
        object-fit: cover;
      "
        src="https://images.pexels.com/photos/1704488/pexels-photo-
1704488.jpeg?auto=compress&cs=tinysrgb&dpr=1&w=500"
        alt="Mark"
```

```
      />
    </div>
  );
}
function App() {
  return (
    <div className="App">
      <UserPic />
    </div>
  );
}
```

The error displayed on running Code Snippet 5 is shown in Figure 6.2.

Figure 6.2: Error on Running Code Snippet 5

In Code Snippet 6, the styles object is provided as value to the style attribute using the Template syntax.

> **Important:**
> An important point to keep in mind is that the styles object keys must all be in CamelCase. Otherwise, ReactJS will give an error.

Code Snippet 6 is written in `App.js`.

Code Snippet 6:

```
function UserPic() {
  const styles = {
    height: "200px",
    aspectRatio: "1/1",
    borderRadius: "50%",
    objectFit: "cover",
  };
  return (
```

```
    <div>
      <img
        style={styles}
        src="https://images.pexels.com/photos/1704488/pexels-photo-
1704488.jpeg?auto=compress&cs=tinysrgb&dpr=1&w=500"
        alt="Mark"
      />
    </div>
  );
}
function App() {
  return (
    <div className="App">
      <UserPic />
    </div>
  );
}
```

The output of Code Snippet 6 as shown in Figure 6.3 clearly indicates that the styles are applied as inline styles in the DOM.

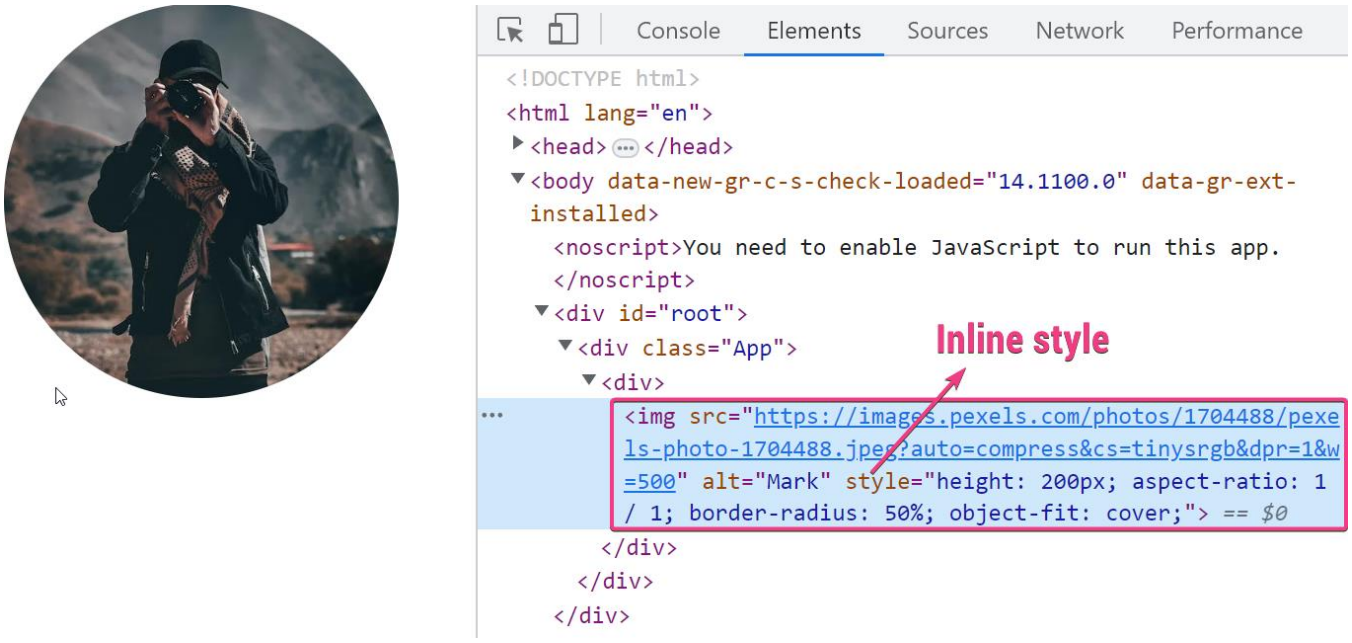


Figure 6.3: Output of Code Snippet 6 and Code showing the Inline Style

Note: Inline styles are generally used in ReactJS when the style has to be changed dynamically.

6.2.2 External CSS Stylesheet/Using Plain CSS

To use an external CSS with ReactJS, the student must use the following process:

Declare the styles separately in a file such as `App.css`, as shown in Code Snippet 8.

Import the `App.css` file into the component file as shown in Code Snippet 7.

The classes given in the CSS file can be used in the component by assigning the class to the `className` attribute of the element as shown in Code Snippet 7.

Code Snippet 7 is written in `App.js`.

Code Snippet 7:

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

function UserPic() {

  return (

    <div>

      <img

        className="profile-pic"

        src="https://images.pexels.com/photos/1704488/pexels-photo-
1704488.jpeg?auto=compress&cs=tinysrgb&dpr=1&w=500"

        alt="Mark"

      />

    </div>

  );

}
```

Code Snippet 8 is written in `App.css`.

Code Snippet 8:

```
.profile-pic {

  height: 200px;

  aspect-ratio: 1/1;

  border-radius: 50%;

  object-fit: cover;

}
```

When the app is running in development, the styles are added as Internal CSS by ReactJS as shown in Figure 6.4.

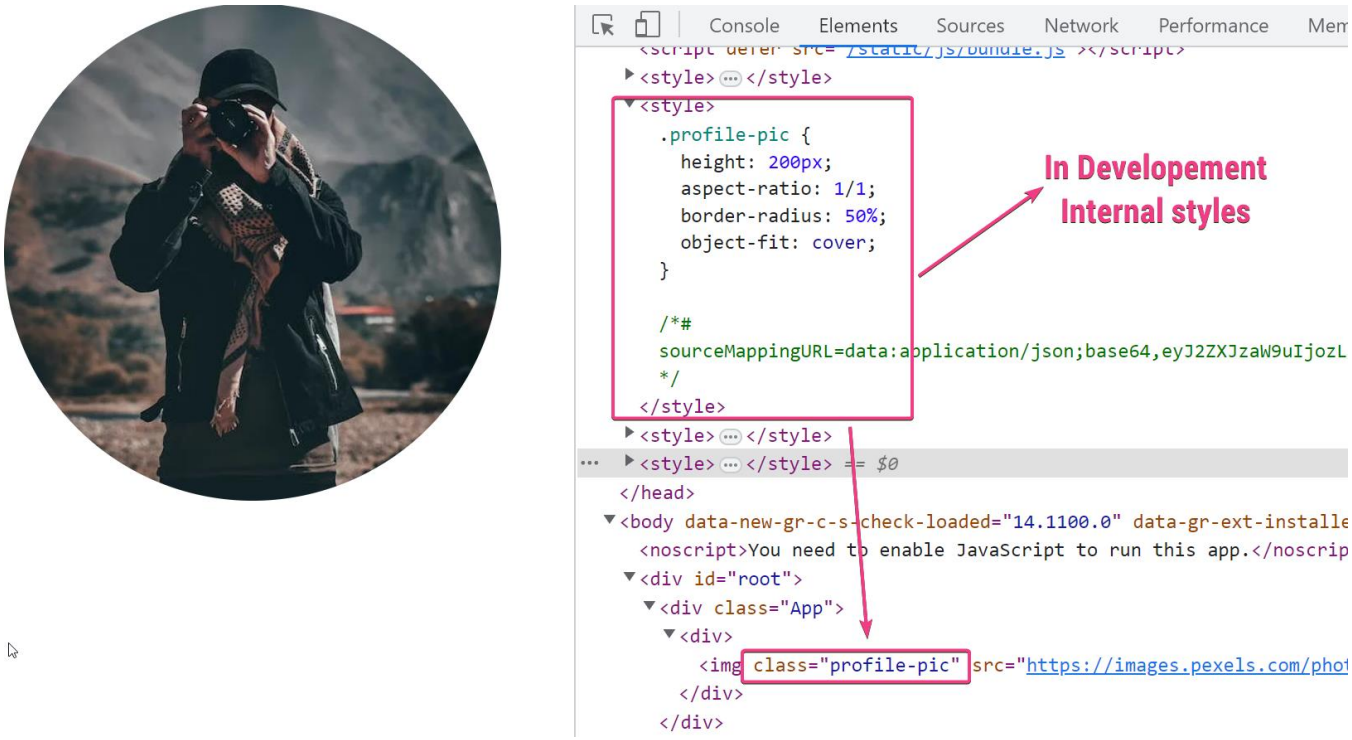


Figure 6.4: Output of Code Snippets 7 and 8 When in Development

When the app is in production, that is when it reaches the user, it is loaded as an External CSS file as shown in Figure 6.5.

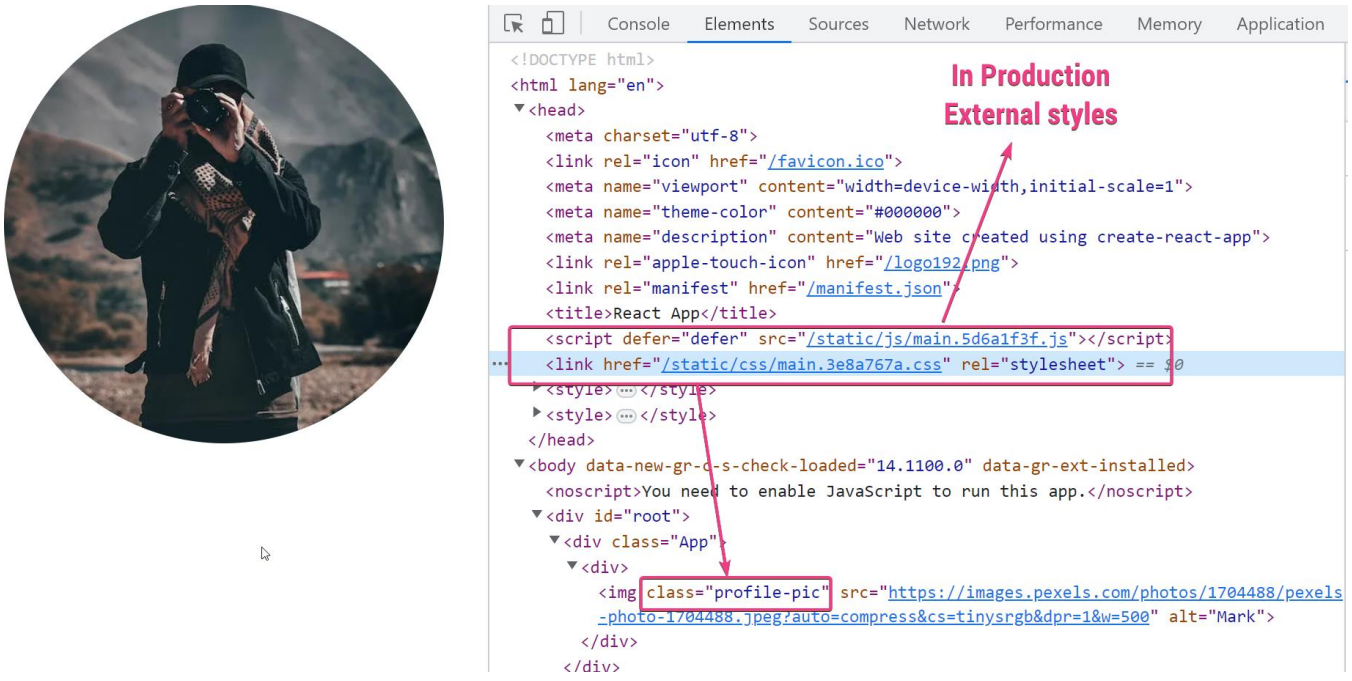


Figure 6.5: Output of Code Snippets 7 and 8 When in Production

6.2.3 CSS Module

To use CSS modules with ReactJS, the student must use the following process:

Declare the styles separately in a file with a filename ending with `module.css` such as `App.module.css`, as shown in Code Snippet 10.

Import the file into the component file as shown in Code Snippet 9.

The classes given in the CSS file can be used in the component by assigning the class to the `className` attribute of the element as shown in Code Snippet 9.

`styles["profile-pic"]` is the value that contains styles declared in Code Snippet 10 because the styles contain the class names as keys.

Code Snippet 9 is written in `App.js`.

Code Snippet 9:

```
import styles from "./App.module.css";

function UserPic() {

  return (

    <div>

      <img

        className={styles["profile-pic"]}

        src="https://images.pexels.com/photos/1704488/pexels-photo-
1704488.jpeg?auto=compress&cs=tinysrgb&dpr=1&w=500"

        alt="Mark"

      />

    </div>

  );

}
```

Code Snippet 10 is written in `App.css`.

Code Snippet 10:

```
.profile-pic {
  height: 200px;
  aspect-ratio: 1/1;
  border-radius: 50%;
  object-fit: cover;
}
```

When the app is running in development, the styles are added as Internal CSS by ReactJS as shown in Figure 6.6.

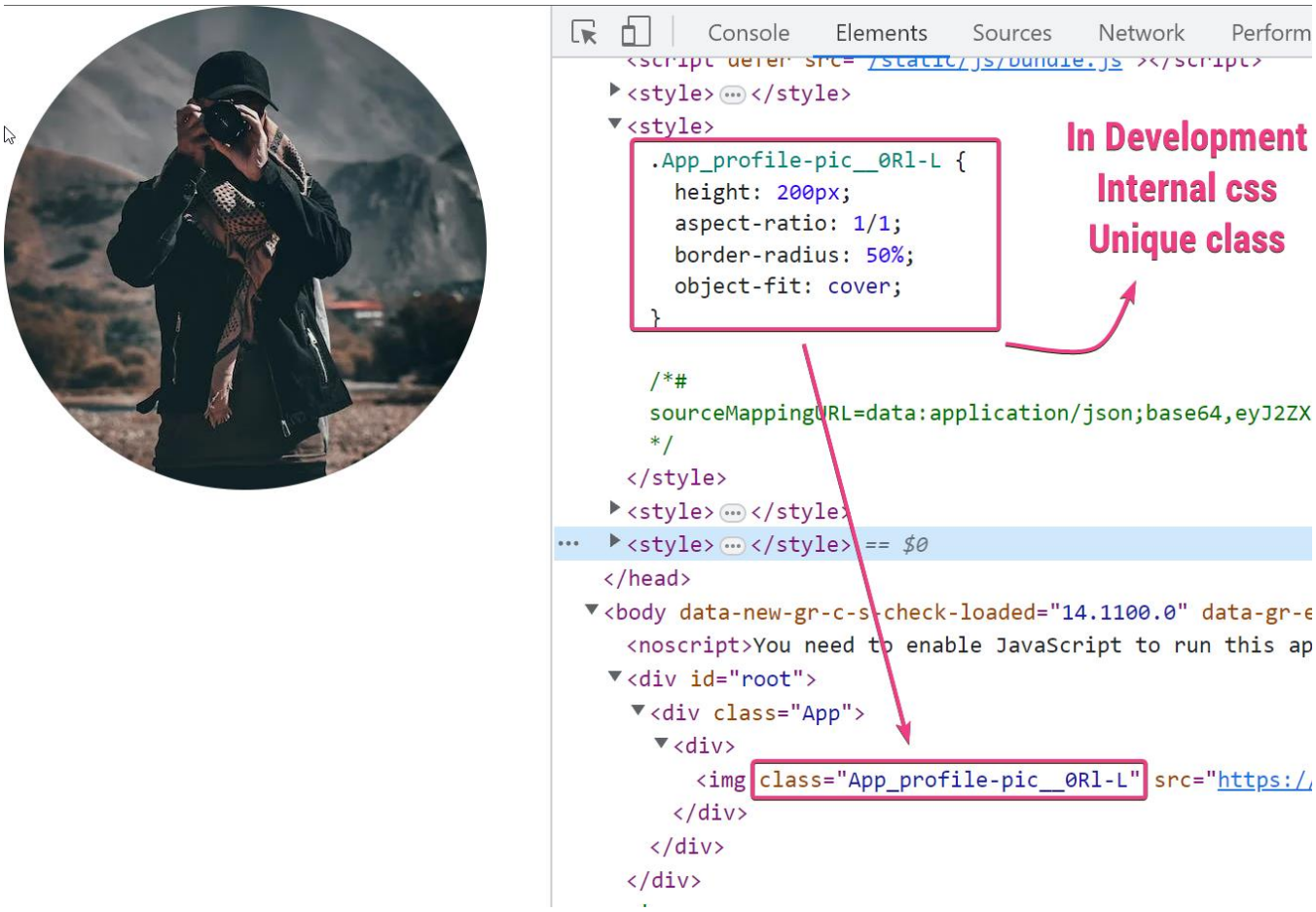


Figure 6.6: Output of Code Snippets 9 and 10 when in Development

The app in production, which is loaded as an External CSS file is shown in Figure 6.7.

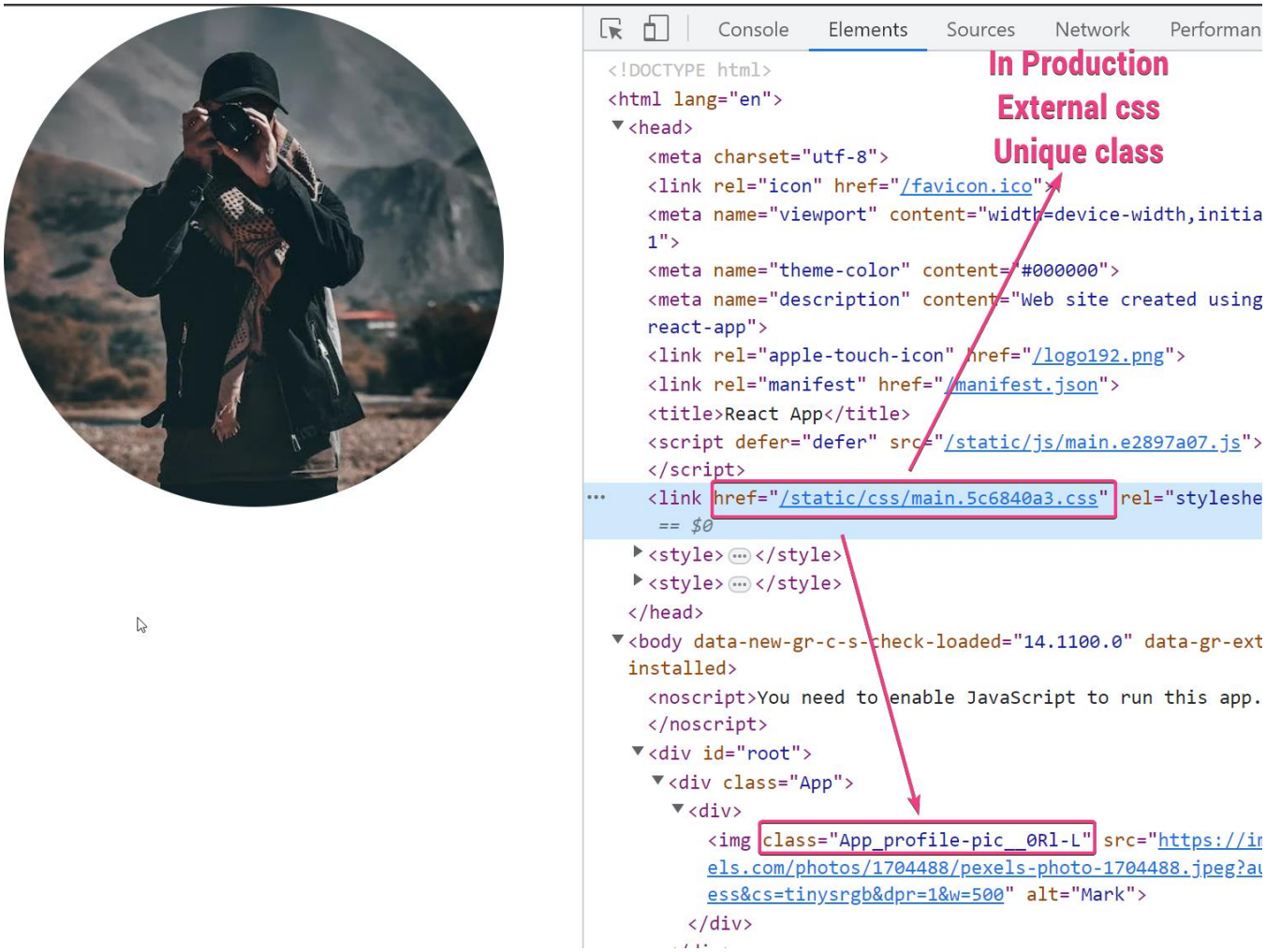


Figure 6.7: Output of Code Snippets 9 and 10 When in Production

The benefit of CSS modules over using plain External CSS is that the class name created will always be unique. Hence, the worry about whether there would be a clash of styling or unwanted style override in the app is not required.

6.2.4 External Fonts

Steps to add external fonts are as follows:

1. Visit https://fonts.google.com/ as shown in Figure 6.8.

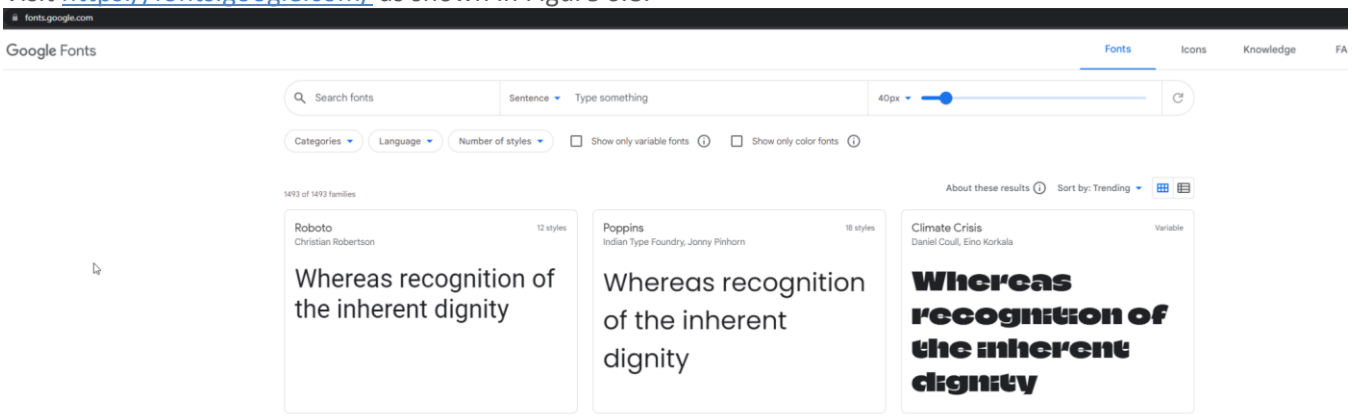


Figure 6.8: Accessing the Fonts through Google

2. Choose the required font for your app and the different font weights as shown in Figure 6.9.

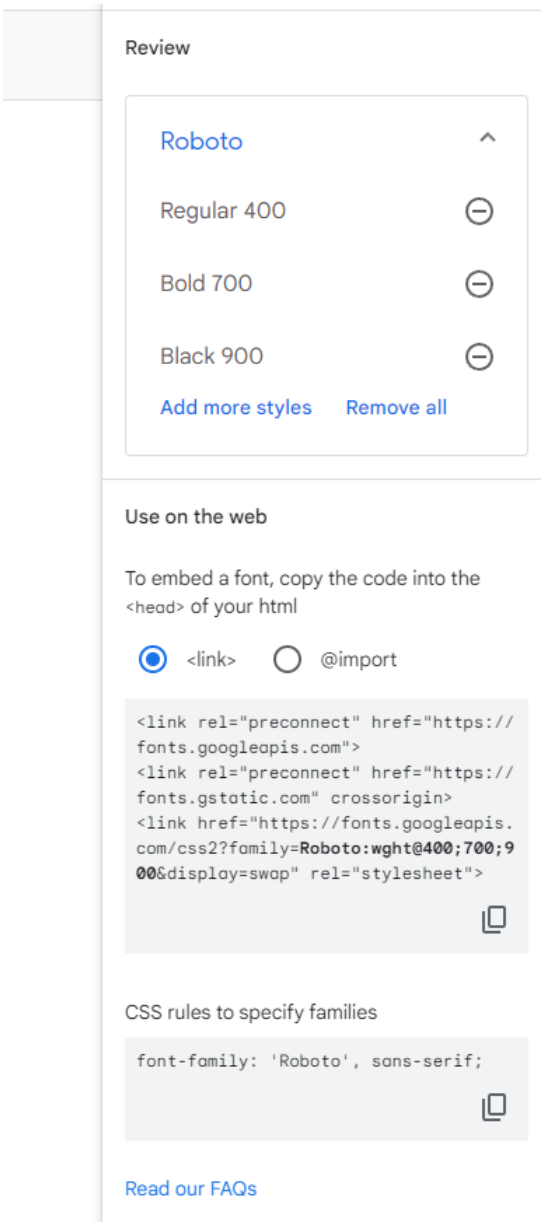


Figure 6.9: Choosing the Font and Font Weights

3. Copy the `<link>` tag that is generated as shown in Code Snippet 11.
   Code Snippet 11:

```
    <link rel="preconnect" href="https://fonts.googleapis.com" />
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin />
    <link

href="https://fonts.googleapis.com/css2?family=Roboto:wght@400;700;900&display=swap"
      rel="stylesheet"
    />
```

4. Navigate to the `public/index.html` file as shown in Figure 6.10.

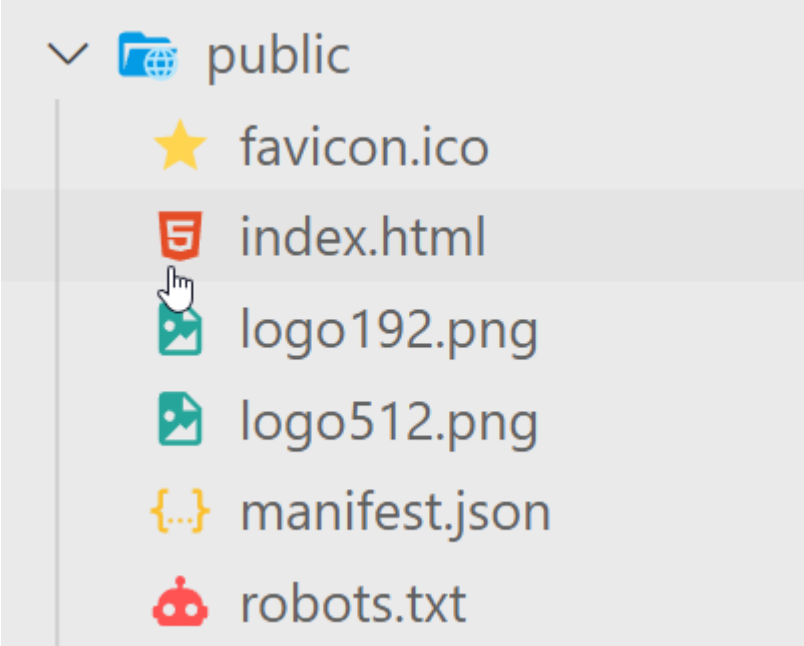


Figure 6.10: Navigating to `index.html`

5. Paste the link tags under the <head> tag in the `index.html` file as shown in Figure 6.11.

```
    Notice the use of %PUBLIC_URL% in the tags above.
    It will be replaced with the URL of the `public` folder during the build.
    Only files inside the `public` folder can be referenced from the HTML.

    Unlike "/favicon.ico" or "favicon.ico", "%PUBLIC_URL%/favicon.ico" will
    work correctly both with client-side routing and a non-root public URL.
    Learn how to configure a non-root public URL by running `npm run build`.
  -->
  <title>React App</title>
  <link rel="preconnect" href="https://fonts.googleapis.com" />
  <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin />
  <link
    href="https://fonts.googleapis.com/css2?family=Roboto:wght@400;700;900&display=swap"
    rel="stylesheet"
  />
</head>
<body>
  <noscript>You need to enable JavaScript to run this app.</noscript>      You, last week • Ini
  <div id="root"></div>
```

Figure 6.11: Adding the Link tags

6. The font-family is ready to use in the app as shown in Code Snippet 12. Code Snippet 12 is written in `App.css`.

   Code Snippet 12:

```
body{

  font-family: 'Roboto', sans-serif;

}
```

6.2.5 Bootstrap Icons

Steps to add icons to the app are as follows:

1. To add icons to the app, install the `react-bootstrap-icons` through the command line.

```
npm i react-bootstrap-icons
```

2. Import the necessary icons and call the component as shown in Code Snippet 13. Code Snippet 13 is written in `App.js`.

Code Snippet 13:

```
import { Instagram, Twitter } from "react-bootstrap-icons";
function App() {
return (
<div className="App">
<Instagram /> <Twitter />
</div>
);
}
```

Output of Code Snippet 13 is shown in Figure 6.12.

Figure 6.12: Output of Code Snippet 13

6.3 Summary

- ✓ Three different types of styling in HTML are:
  - Inline.
  - Internal.
  - External.
- ✓ Three different types of styling in ReactJS are:
  - Using styled objects.
  - External CSS.
  - CSS modules.
- ✓ External CSS is always recommended.
- ✓ Inline styles are generally used in ReactJS when the style requires to be changed dynamically.
- ✓ CSS modules always provide unique style class names.
- ✓ Icons can easily be imported as components from the `react-bootstrap-icons`.
- ✓ Fonts can be directly added to the `<Head>` tag in `index.html`.

6.4 Check Your Knowledge

1. Which is the preferred way of styling?

A	Internal styling
B	Inline styling
C	External styling
D	None

2. _____ always provides unique CSS classes.

A	Style Objects
B	CSS Modules
C	External CSS
D	External Styles

3. _____ is the tag under which fonts are added.

A	`font`
B	`head`
C	`html`
D	`img`

4. Style objects generate internal CSS.

A	True
B	False

5. _____ case must be used in style objects.

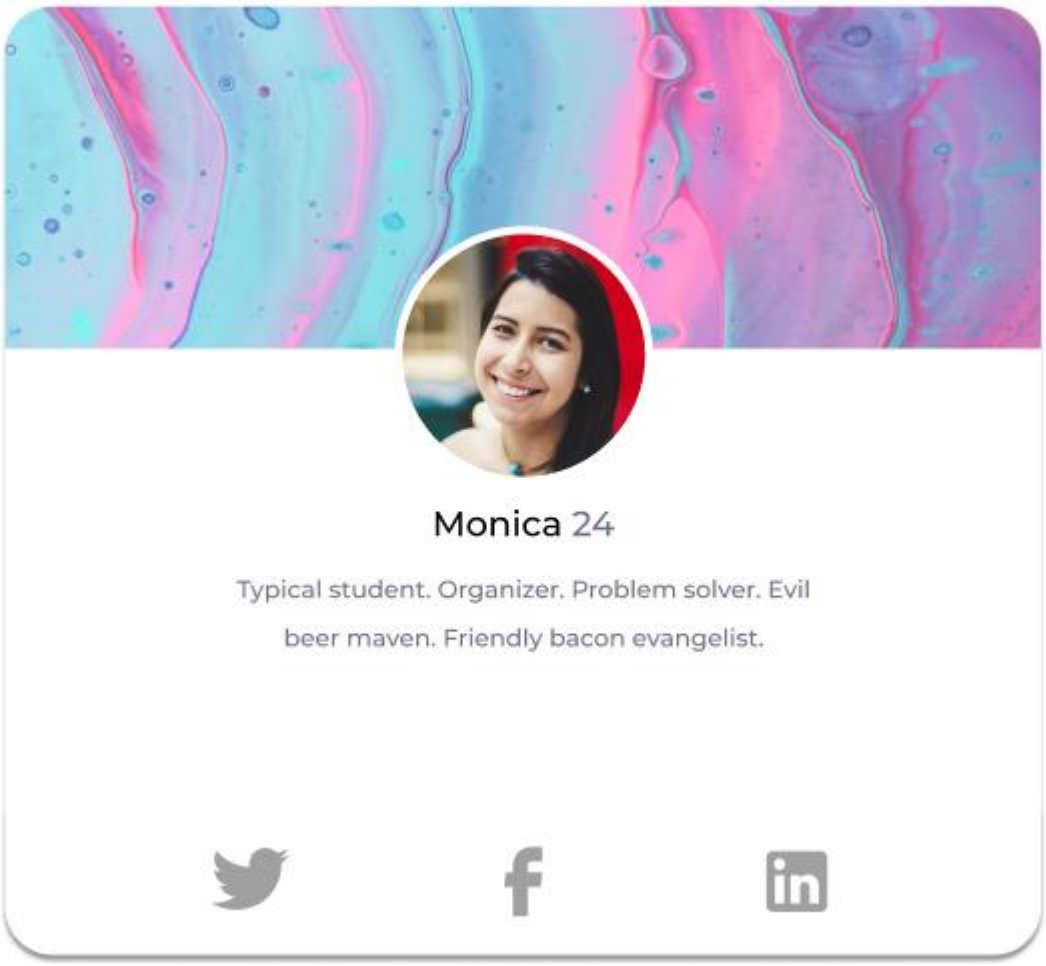
A	Kebab
B	Underscore
C	Title
D	Camel

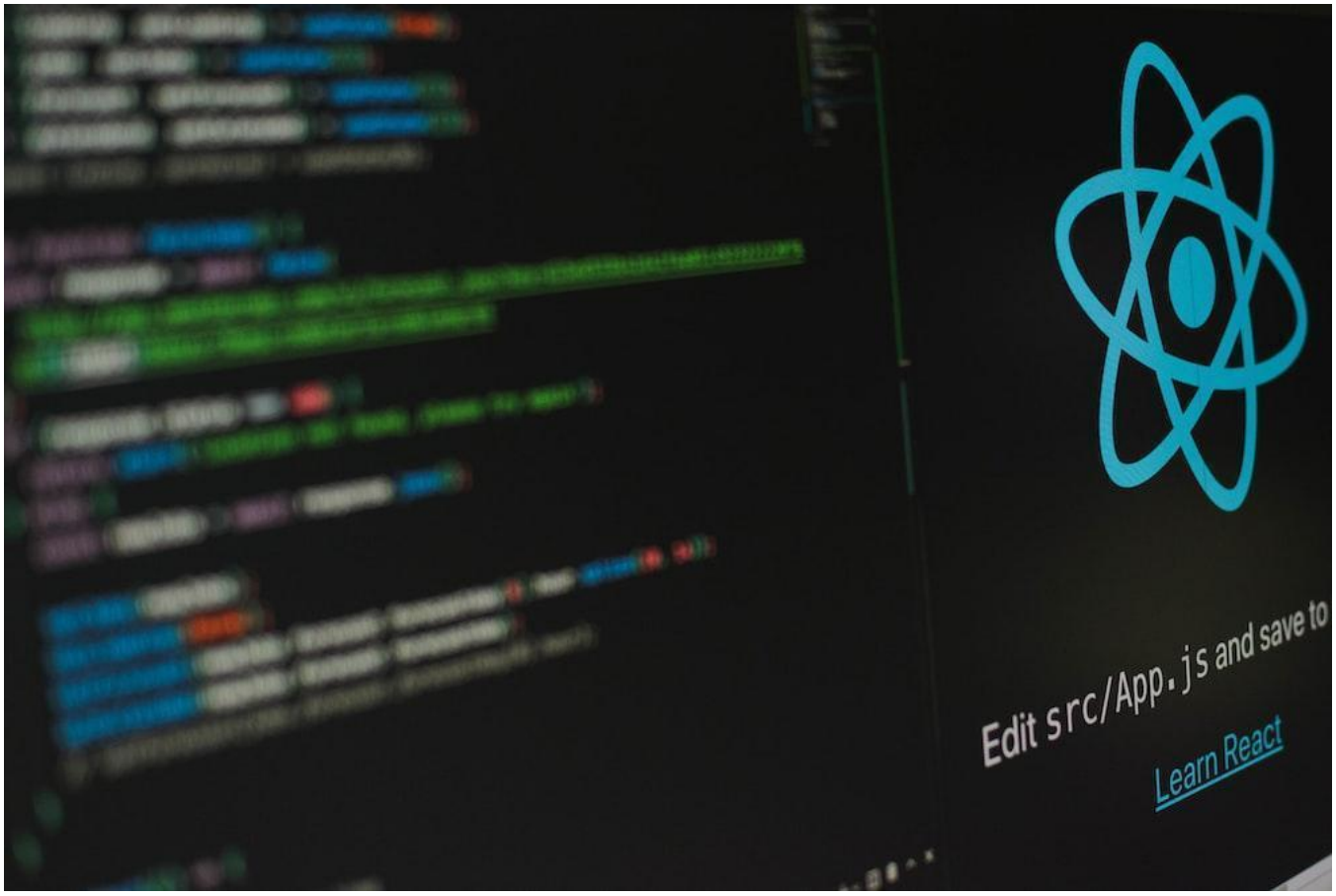
Answers

1	C
2	B
3	B
4	B
5	D

Try It Yourself

You are a developer and you must design and develop a user profile. However, you are worried that it will be overridden by other styles. Hence, complete the design as shown in the Figure by using CSS modules.





Session 7

ReactJS Component Lifecycle

Session Overview

This session provides information about the different phases of ReactJS components. It also provides information about the `useEffect` hook and implementing it in different scenarios.

Objectives

In this session, students will learn to:

- Explain different phases of the ReactJS component
- Explain how to manage lifecycles using the useEffect hook
- Illustrate the implementation of useEffect in apt scenarios

7.1 Different Phases of ReactJS Components

Humans go through different phases in life from birth to death such as childhood, adolescence, middle age, and then old age. Similarly, the lifecycle of ReactJS components also has four different phases:

Initialization Phase	In the initialization phase, properties (props) and state of the component are set up.
Mounting Phase	In the mounting phase, the DOM elements of the component are created and rendered on the DOM tree. Thus, the elements are visible to the user.
Updating Phase	The component is said to be in the updating phase when the props or state of the component change.
Unmounting Phase	In this phase, the component is removed from the DOM tree. Thus, the DOM elements associated with it is destroyed.

In the past, to handle these phases in class components, there were several lifecycle methods. However, this proved to be cumbersome to use. Many of the methods were very rarely utilized and difficult to remember, especially for the beginners. Hence, with the introduction of React hooks and functional components, one hook is able to solve all the needs, which is the `useEffect` hook.

7.2 Managing the Lifecycle Using the useEffect Hook

The `useEffect` hook takes in two arguments:

`callback function`

`dependency array`

7.2.1 Syntax

The syntax for `useEffect` is:

```
useEffect(callback function, dependency array);
```

The `useEffect` is able to handle different phases of the component based on the dependency array.

7.2.2 Empty Dependency Array

When an empty dependency array is given, the callback function is called only once, that is when the component is mounted. An example of the usage of empty dependency array is:

```
useEffect(() => {}, []);
```

7.2.3 No Dependency Array

When the dependency array is not given, the callback function is called whenever any state or props changes in the component. This happens when the component is in the updating phase. An example of the usage of a no dependency array is:

```
useEffect(() => {});
```

7.2.4 Values in Dependency Array

Whenever values are given inside the dependency array, the `callback` function will be called only when `v1` or `v2` changes. In the dependency array, there could be `n` number of values. When two values are used, mostly, `v1` and `v2` would be either `state` or `props` of the component. An example of using two values is:

```
useEffect(() => {}, [v1, v2]);
```

7.2.5 Return in `useEffect`

When the `return` function is given inside the `callback` function, the `return` function will be executed when the component starts its unmounting phase. Usually, clean-up code (that is, variables, which are no longer needed in memory will be destroyed and which helps in preventing memory leak) is given inside the return function. An example is shown in Code Snippet 1.

Code Snippet 1:

```
useEffect(() => {
  return () => {};
}, []);
```

7.3 Implementing useEffect in Apt Scenarios

With the introduction of React v18, `strictmode` in ReactJS renders every component twice to ensure that there are no errors. This rendering creates a few queries in our mind such as whether all setup codes will run twice. the answer is Yes. Only in development phase, the codes will run twice and not in production. So, when the code reaches the user, all the components are run only once.

Hence, to test the app without `strictmode` comment out in the `index.js` file as shown in Code Snippet 2.

Code Snippet 2:

```
import React from "react";
import ReactDOM from "react-dom/client";
import "./index.css";
import App from "./App";
import reportWebVitals from "./reportWebVitals";
import { BrowserRouter } from "react-router-dom";
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
  // <React.StrictMode>
  <BrowserRouter>
    <App />
  </BrowserRouter>
  // </React.StrictMode>
);
// If you want to start measuring performance in your app, pass a function
```

```
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

Consider Code Snippet 3, which is written in `App.js`, wherein when the dependency array is not given:

- Whenever any state changes, the `callback` function is called. This indicates that whenever either `like` or `dislike` changes, the `console.log` is created/updated because both are states of the component.
- When either the `like` or `dislike` button is clicked, it changes the `like` and `dislike` values respectively, which leads the `console.log` of the `like` value.

Code Snippet 3:

```
function Counter() {
  let [like, setLike] = useState(0);
  let [dislike, setDisLike] = useState(0);
  useEffect(() => {
    console.log("like is updated", like);
  });
  return (
    <div>
      <button onClick={() => setLike(like + 1)}>👍 {like}</button>
      <button onClick={() => setDisLike(dislike + 1)}>👎 {dislike}</button>
    </div>
  );
}
```

Output of Code Snippet 3 is shown in Figure 7.1.

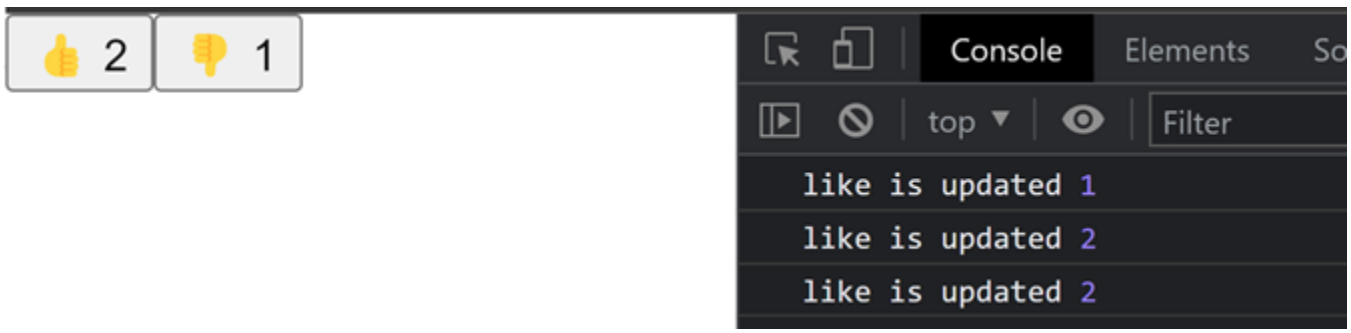


Figure 7.1: Output of Code Snippet 3

In Code Snippet 4, which is written in `App.js`, when the dependency array contains `like`, the `callback` function is called only when `like` changes. Thus, when the `like` button is clicked, the `console.log` is created.

Code Snippet 4:

```
function Counter() {
  let [like, setLike] = useState(0);
  let [dislike, setDisLike] = useState(0);
  useEffect(() => {
    console.log("like is updated", like);
  }, [like]);
  return (
    <div>
      <button onClick={() => setLike(like + 1)}>👍 {like}</button>
      <button onClick={() => setDisLike(dislike + 1)}>👎 {dislike}</button>
    </div>
  );
}
```

Output for Code Snippet 4 is shown Figure 7.2.

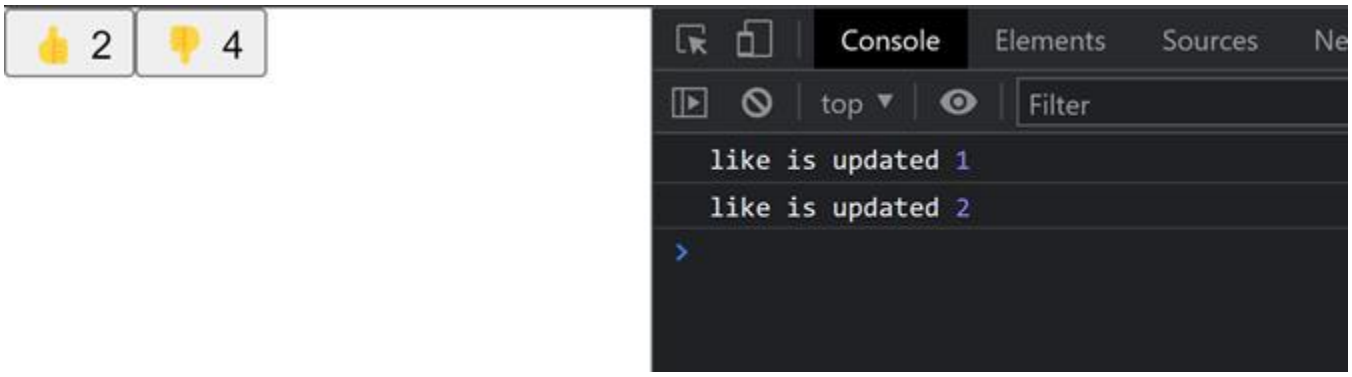


Figure 7.2: Output of Code Snippet 4

`"useEffect runs"` is called only once because its empty dependency array is used. Thus, the `callback` function is called only once when the component is mounted. Therefore, the `setInterval` is started and the value is updated on the screen and also prints "`Timer called`" every second.

As shown in Code Snippet 5, which is written in `App.js`, when `setTime` is called:

- A new technique is used set the value of time.
- The `setTime` accepts not only values but also the `callback` function.

- In the `callback` function, the value of previous (`prev`) is received as the argument to update the `like` value. The `like` value is then returned by adding one to it.

Code Snippet 5:

```
function Timer() {
  const [time, setTime] = useState(0);
  useEffect(() => {
    console.log("useEffect runs");
    setInterval(() => {
      console.log("Timer called");
      setTime((prev) => prev + 1);
    }, 1000);
  }, []);
  return (
    <div>
      <h1>{time}</h1>
    </div>
  );
}
function App() {
  return (
    <div className="App">
      <Timer />
    </div>
  );
}
```

The output of Code Snippet 5 is shown in Figure 7.3.

5

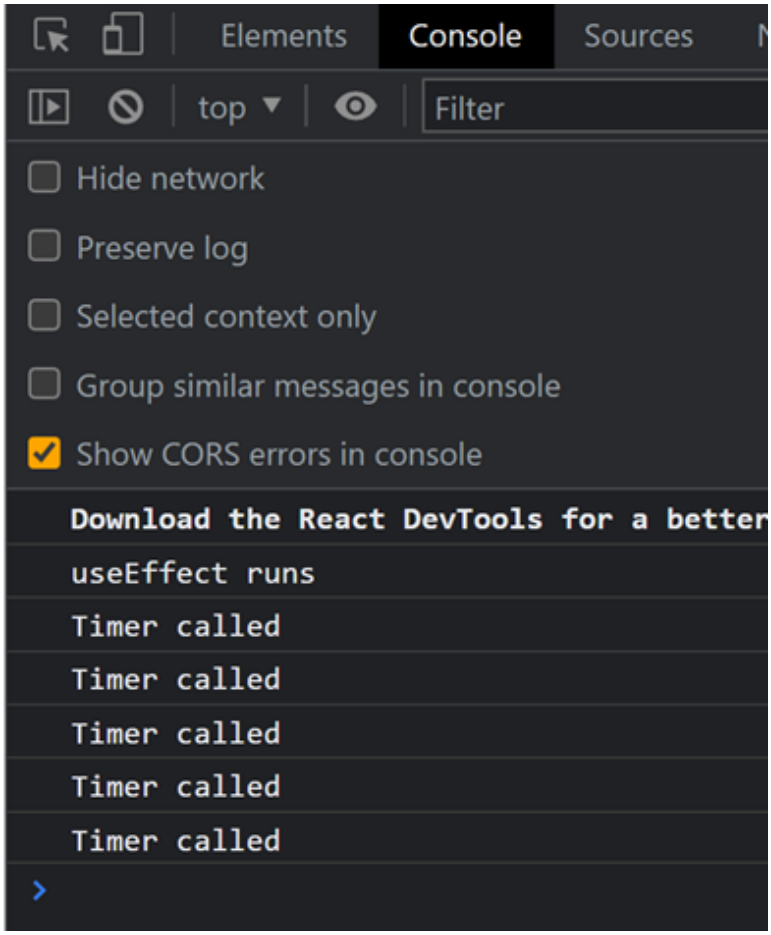


Figure 7.3: Output of Code Snippet 5

In the following line, a technique called Conditional Rendering is used.

```
{show ? <Timer /> : null}
```

As shown in Code Snippet 6, based on the condition, the `<Timer />` component is shown or removed from the DOM. Hence, the following occurs:

- When show is true, the `<Timer />` is shown. Thus, when the `Start` button is clicked, the timer is shown.
- When show is false, the `<Timer />` is removed from the DOM. Thus, when the stop button is clicked, the timer is removed.

When tested by clicking the stop button, the console still prints "`Timer called`", which means there is a memory leak. It indicates that the Interval did not stop even after the `<Timer />` component was removed. Thus, to solve this, `Return in useEffect` is required.

Code Snippet 6 is written in `App.js`.

Code Snippet 6:

```
function App() {
  const [show, setShow] = useState(false);
  return (
    <div className="App">
```

```
      <button onClick={() => setShow(true)}>Start</button>
      <button onClick={() => setShow(false)}>Stop</button>
      {show ? <Timer /> : null}
    </div>
  );
}
```

The outputs of Code Snippet 6 are shown in Figure 7.4 and Figure 7.5.

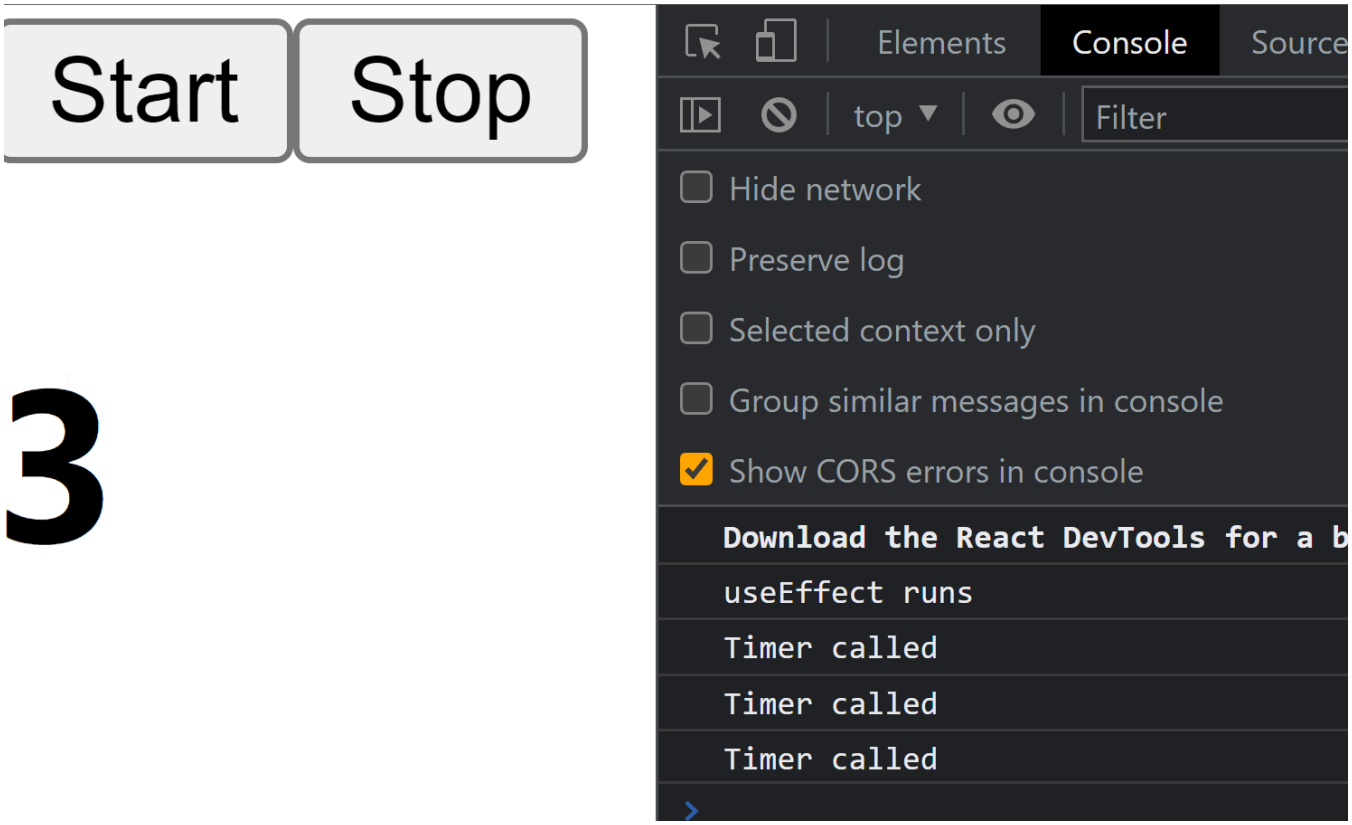


Figure 7.4: Output of Code Snippet 6 with Timer Shown

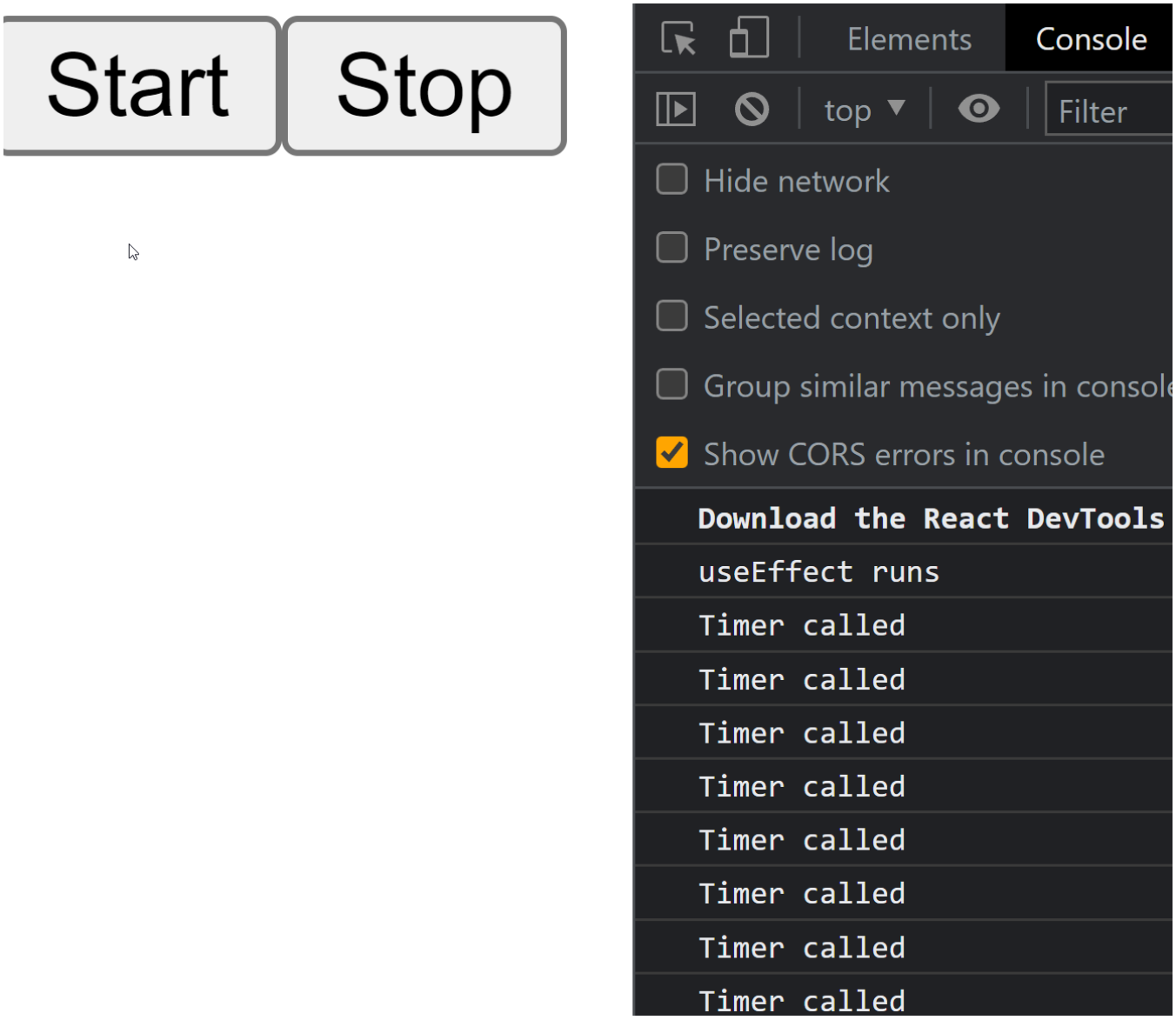


Figure 7.5: Output Of Code Snippet 6 with Timer Not Shown

In Code Snippet 7, inside `return`, the cleanup code (`clearInterval`) is used to stop the interval. When the stop button is clicked, the following occurs:

- The `<Timer />` is destroyed and "`Timer called`" is not printed.
- The message, "`Timer stopped`" is printed as shown in Figure 7.6, which signifies that the memory leak is handled.

Code Snippet 7 is written in `App.js`.

Code Snippet 7:

```
function Timer() {
  const [time, setTime] = useState(0);
  useEffect(() => {
    console.log("useEffect runs");
```

```
    const timer = setInterval(() => {
      console.log("Timer called");
      setTime((prev) => prev + 1);
    }, 1000);
    return () => {
      console.log("Timer stopped");
      clearInterval(timer);
    };
  }, []);
  return (
    <div>
      <h1>{time}</h1>
    </div>
  );
}
```

Output of Code Snippet 7 is shown in Figure 7.6.

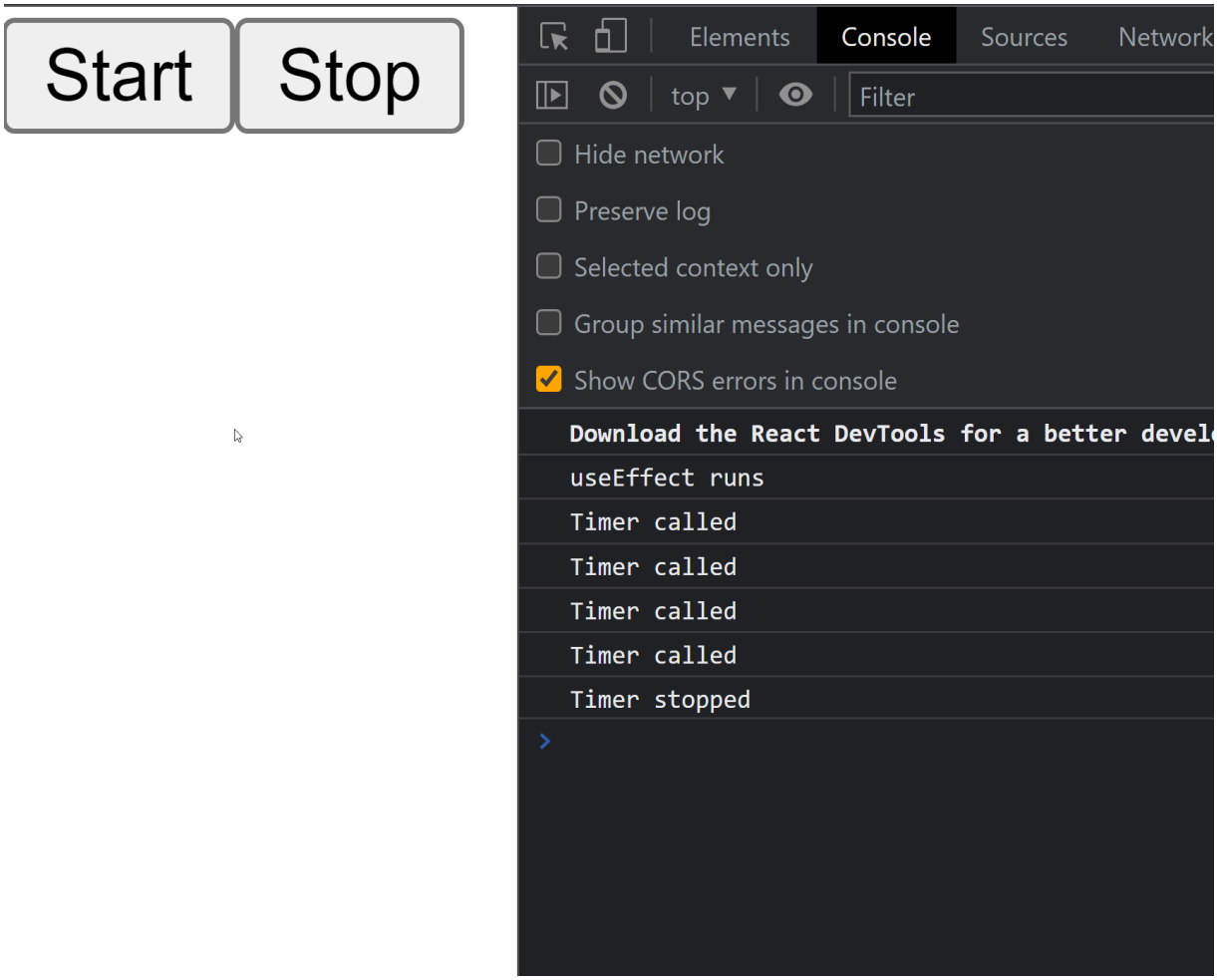


Figure 7.6: Output of Code Snippet 7

In Session 8, students will learn the real-time application of `useEffect` in detail. `useEffect` is generally meant to synchronize the ReactJS component phase with external systems such as fetching data from API. In other words, data is fetched when the component is ready and mounted. In such scenarios, `useEffect` works very well.

7.4 Summary

- ✓ Four phases of the lifecycle of ReactJS components are:
  - Initialization.
  - Mounting.
  - Updating.
  - Unmounting.
- ✓ Using the `useEffect` hook is an efficient way to handle the lifecycle phases of the component.
- ✓ Dependency array dictates in which phase of the component the `callback` function will be called.
- ✓ Conditional Rendering is a technique where based on a condition, a component is shown or removed from the DOM.
- ✓ Memory leaks need to be handled in the clean part of `useEffect`.

7.5 Check Your Knowledge

1. Unmounting phase is handled with _____ of `useEffect`.

A	Empty Dependency Array
B	Values In Dependency Array
C	Return function
D	No Dependency array

2. Empty Dependency Array with `useEffect` is used to handle _____ phase of the component.

A	Mounting
B	Updating
C	Unmounting
D	Initializing

3. The component is in _____ phase when state or props changes.

A	Mounting
B	Updating
C	Unmounting
D	Initialization

4. Values in Dependency Array are given to write clean up code in `useEffect`.

A	True
B	False

5. When No Dependency array is given, the component will be in the updating phase either when props or state changes.

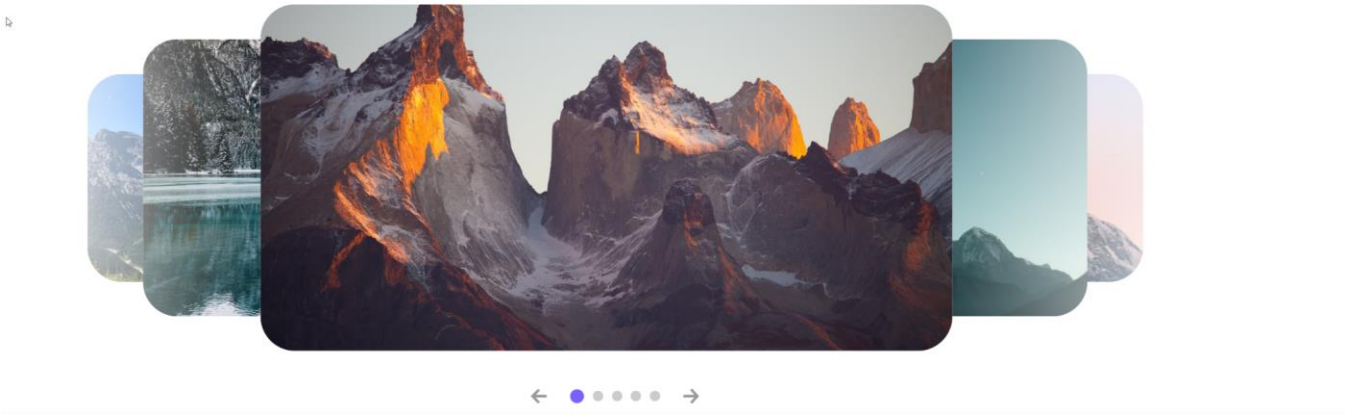
A	True
B	False

Answers

1	C
2	B
3	B
4	B
5	A

Try It Yourself

Build a carousel as shown in the Figure, which keeps changing slides to the right. On hovering, it pauses and when clicked on the small circles it slides to the right side, displaying the next image.



Session 8

Creating a CRUD Application Using ReactJS Forms

Session Overview

This session provides information about creating a CRUD application by using ReactJS forms.

Objectives

In this session, students will learn to:

- Explain CRUD
- Illustrate the development of a simple CRUD application using ReactJS forms
- Illustrate the development of a complex CRUD application using ReactJS forms

8.1 What is CRUD?

CRUD stands for Create, Read, Update, and Delete. All applications in the world are CRUD apps. In other words, any feature in an app requires one of these four operations in building it as shown in Figure 8.1.

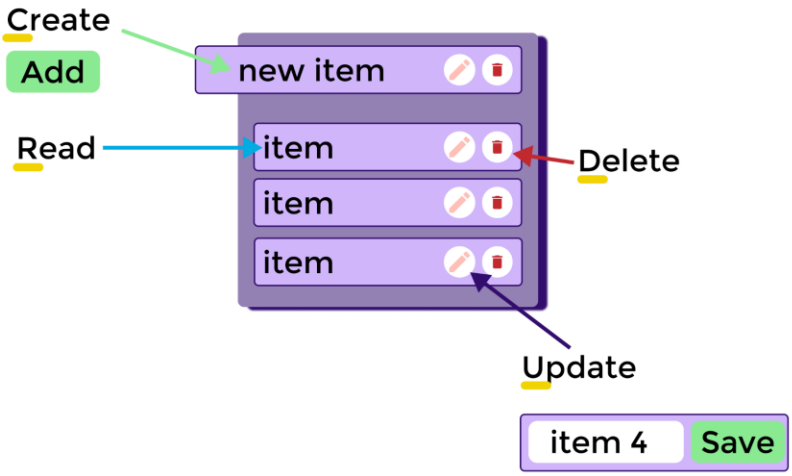


Figure 8.1: Graphical Representation of CRUD

8.1.1 CRUD for Web Applications

In everyday applications, users perform one or more of the following:

C - Create
Example: Creating posts.

R - Read
Example: Reading news feeds.

U - Update
Example: Update/ edit the messages sent.

D - Delete
Example: Deleting unintentional tweets.

8.2 Creating a Simple CRUD Application

Each operation in CRUD uses a special method as shown in Figure 8.2.

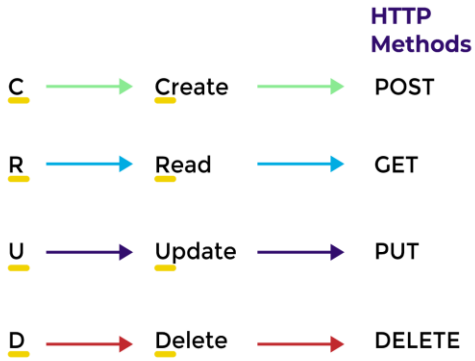


Figure 8.2: Methods Used by CRUD Operations

Students will build a CRUD application, which provides options to create, update, list, and delete users.

To accomplish this task, students will use the codes from Session 5 as a starting point. The code used in Session 5 is available in `session8_starter.zip` file. The code contains the required Cascading Style Sheets (CSS) and JavaScript (JS) to start the application. Download the `session 8_starter.zip` present under Course Files on OnlineVarsity.

The output of `session8_starter.zip` file is shown in Figure 8.3.

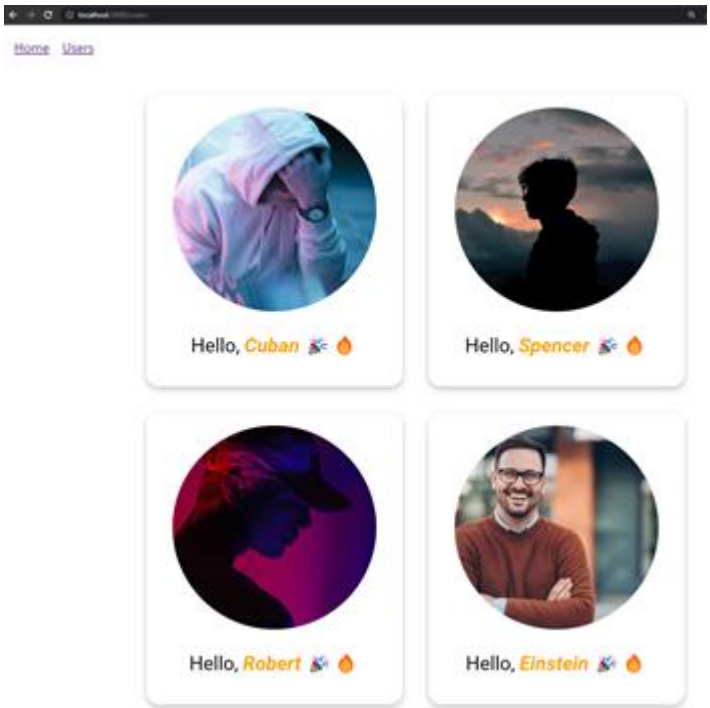


Figure 8.3: Output of `session8_starter.zip` File

8.2.1 Organizing the App

Until now, bulk of the code was in `App.js`, which requires to change. Students must break down the code into multiple files to ensure maintainability and readability in the long run.

To break down the code into multiple files, students can use Visual Studio Code Editor (VSCode).

To move the `NotFound` component to a different file, the steps are as follows:

1. Highlight code of the `NotFound` component and click the bulb icon as shown in Figure 8.4.

Figure 8.4: `NotFound` Component

2. Click on **Move to new file** as shown in Figure 8.5.

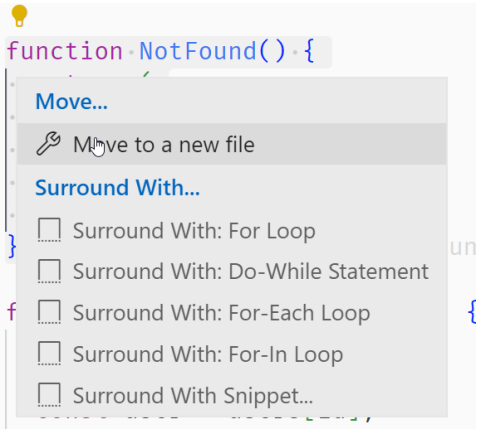


Figure 8.5: Moving to a New File

3. VSCode creates a new file with a filename same as the name of the component name as shown in Figure 8.6.

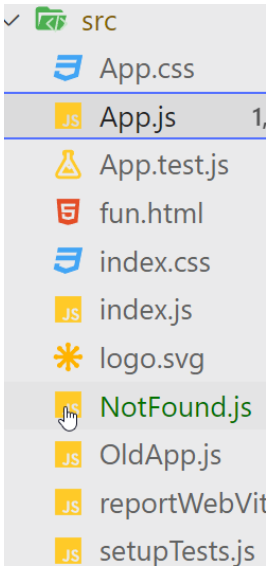


Figure 8.6: New File Named `NotFound.js`

4. Click `NotFound.js`. As one can see VSCode has added an `export` keyword as shown in Figure 8.7, so that students can import the component in any file and utilize it throughout the project.

```
export function NotFound() {
  return (
    <div>
      <h1>404 Not found</h1>
    </div>
  );
}
```

Figure 8.7: Export Keyword Added to the `NotFound` Function

5. Repeat the operation for all the components and move them to their individual files as shown in Figure 8.8.

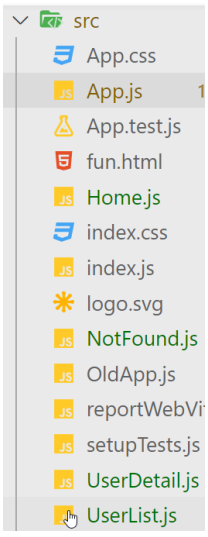


Figure 8.8: List of All the Moved Components

8.2.2 Displaying the User data on the Web Page

To remove the dependency of the local data so that user details are not passed props, use the code as shown in Code Snippet 1, which is written in UserList.js.

```
<Route path="/users" element={<UserList />} />
```

Code Snippet 1:

```
import { User } from "./User";
export function UserList() {
  const users = [];
  return (
    <div className="user-list-container">
      {users.map((usr, index) => (
        <User name={usr.name} pic={usr.pic} id={index} key={index} />
      ))}
    </div>
  );
}
```

Set Up `mockapi.io`

Students will make `UserList` independent. It will ask for data from the `API` and not be dependent on props. Hence, to achieve this, students must store data in `mockapi.io` and receive the data in ReactJS using the fetch method the steps are as follows:

1. To create an `API` endpoint, navigate to http://mockapi.io/, click **LOGIN,** and sign in with a GITHUB account.
   The dashboard appears as shown in Figure 8.9.

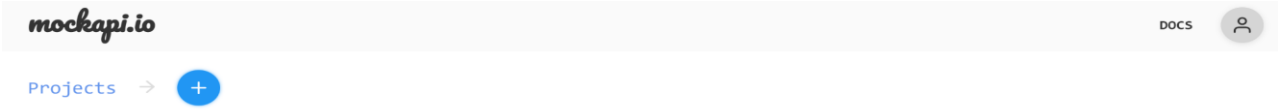


Figure 8.9: Dashboard

2. Click the plus button to create a project. Name the project **Users app** and click the **Create** button.
3. Click the **Users app** and click **NEW RESOURCE** to create an API.
4. Enter the following details as shown in Figure 8.10 and click **CREATE**:
   - Resource name as **users**

- Schema

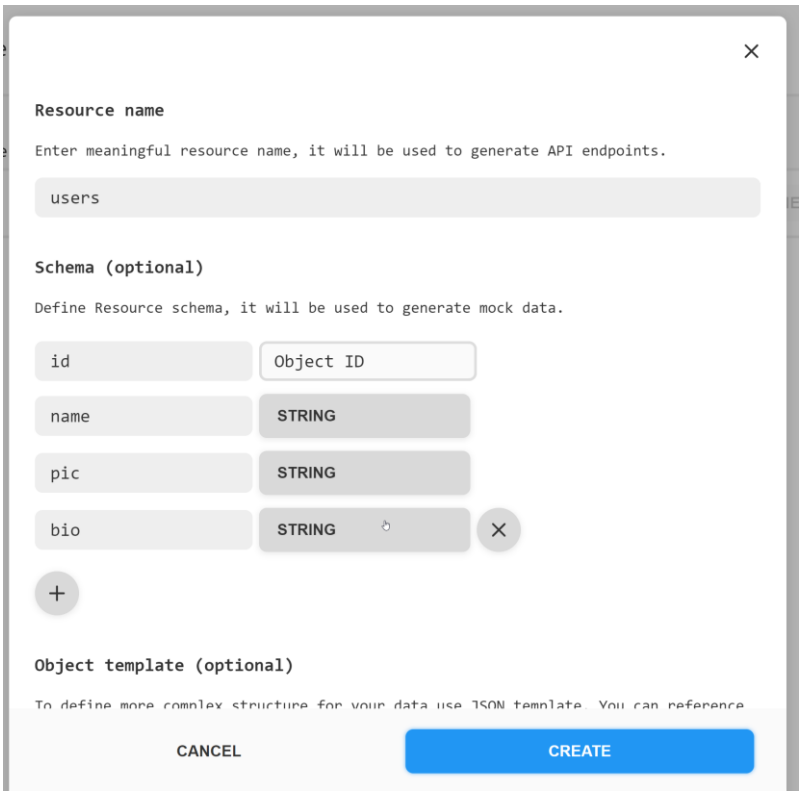


Figure 8.10: Resource Name and Schema Details

5. The users `API` is created as shown in Figure 8.11. Click the **Data** button to add data.

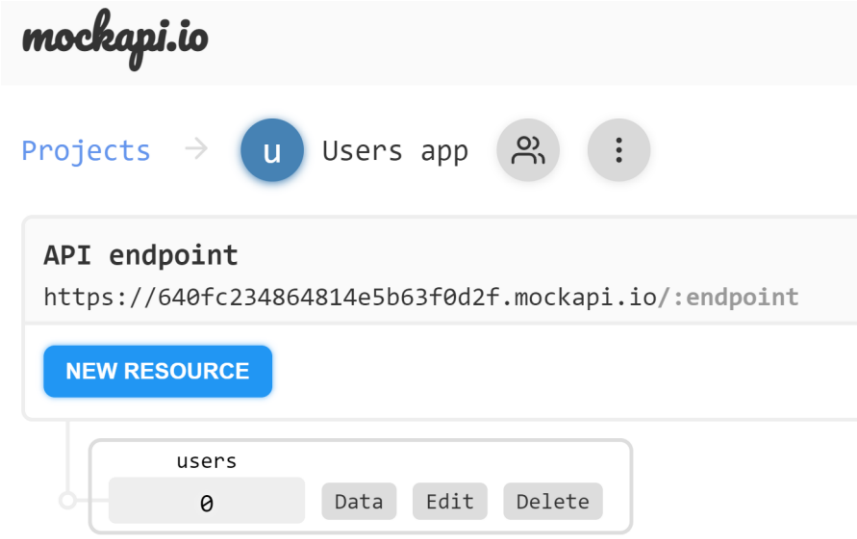


Figure 8.11: Resource Name and Schema Details

6. Paste `JSON` data in the window as shown in Figure 8.12 and click the **`UPDATE`** button.

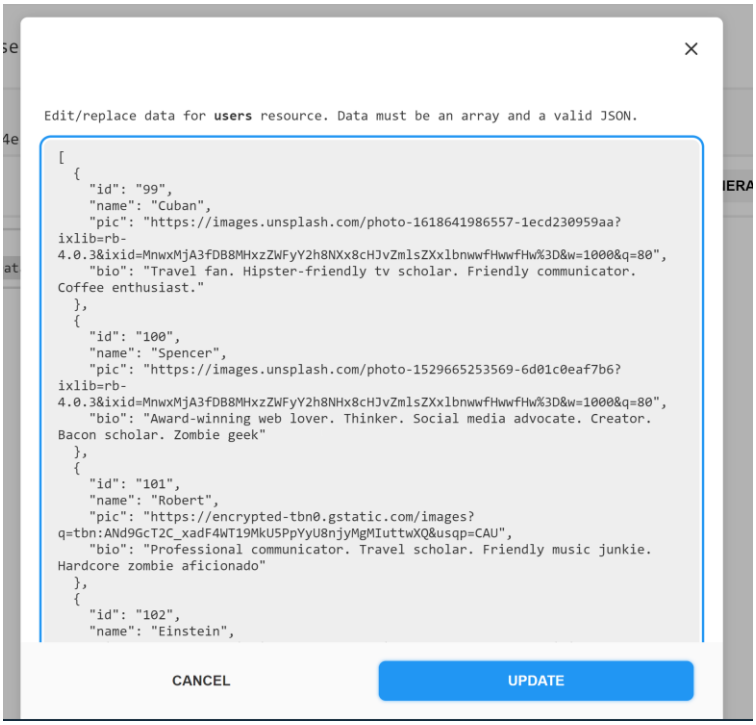


Figure 8.12: Pasting JSON Data

7. In users, four users' data is added as shown in Figure 8.13. Clicking the word **users** (which is a link) will open a new window.

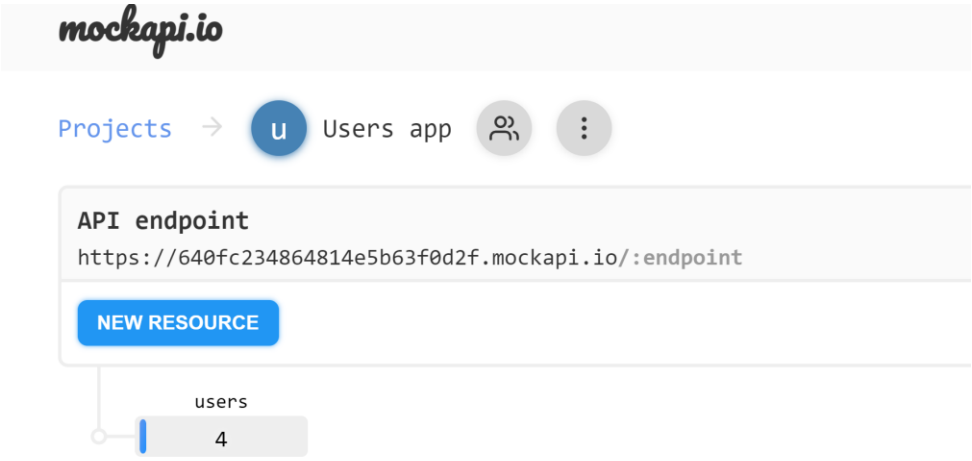


Figure 8.13: Users Link

8. Copy the **URL** since this is the API that will be used to perform CRUD operation in the app. The API created will be unique.

8.2.3 Integrate the users API with UserList

To display the value for the API fetch method used, first, the data is converted into JSON and logged to the console as shown in Code Snippet 2, which is written in `UserList.js`.

Code Snippet 2:

```
export function UserList() {
  const users = [];
```

```
  fetch("https://640fc234864814e5b63f0d2f.mockapi.io/users")
    .then((data) => data.json())
    .then((userList) => console.log(userList));
  return (
    <div className="user-list-container">
      {users.map((usr, index) => (
        <User name={usr.name} pic={usr.pic} id={index} key={index} />
      ))}
    </div>
  );
}
```

The output of Code Snippets 1 and 2 is shown in Figure 8.14.

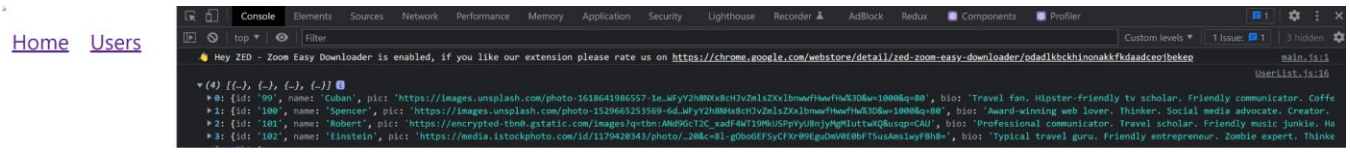


Figure 8.14: Output of Code Snippets 1 and 2

However, the console prints the array twice. To avoid this, the `useEffect` hook is used as shown in Code Snippet 3, which is written in `UserList.js`.

Note: As discussed in Session 7, the console will be executed twice in development, but once in production.

Code Snippet 3:

```
import { useEffect, useState } from "react";
export function UserList() {
  const users = [];
  useEffect(() => {
    fetch("https://640fc234864814e5b63f0d2f.mockapi.io/users")
      .then((data) => data.json())
      .then((userList) => console.log(userList));
  }, []);
  return (
    <div className="user-list-container">
      {users.map((usr, index) => (
        <User name={usr.name} pic={usr.pic} id={index} key={index} />
      ))}
    </div>
  );
```

```
}
```

The output of Code Snippet 3 is shown in Figure 8.15.

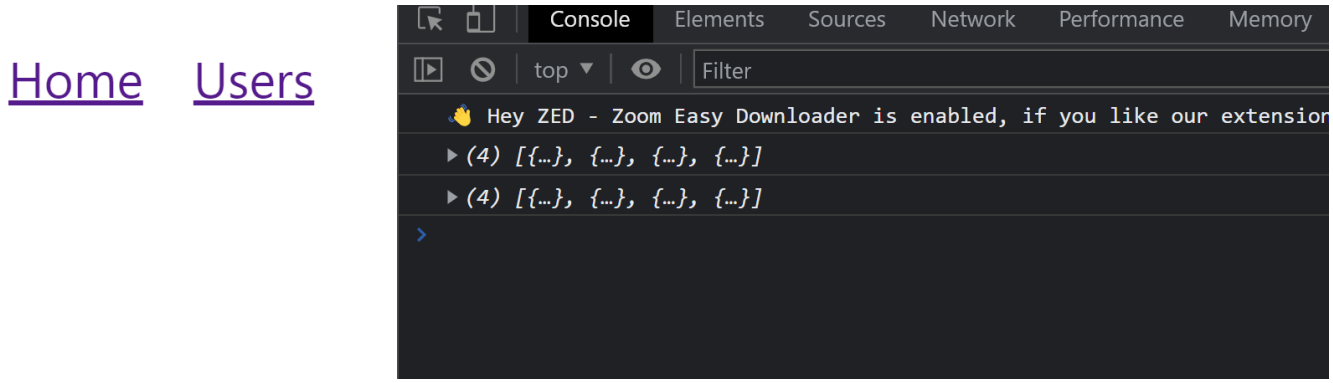


Figure 8.15: Output of Code Snippet 3

To display this data on the screen, the `useState` hook is used to update `users` through `setUsers` as shown in Code Snippet 4, which is written in UserList.js.

Code Snippet 4:

```
export function UserList() {
  const [users, setUsers] = useState([]);
  useEffect(() => {
    fetch("https://640fc234864814e5b63f0d2f.mockapi.io/users")
      .then((data) => data.json())
      .then((userList) => setUsers(userList));
  }, []);
  return (
    <div className="user-list-container">
      {users.map((usr, index) => (
        <User name={usr.name} pic={usr.pic} id={index} key={index} />
      ))}
    </div>
  );
}
```

The output of Code Snippet 4 is shown in Figure 8.16.

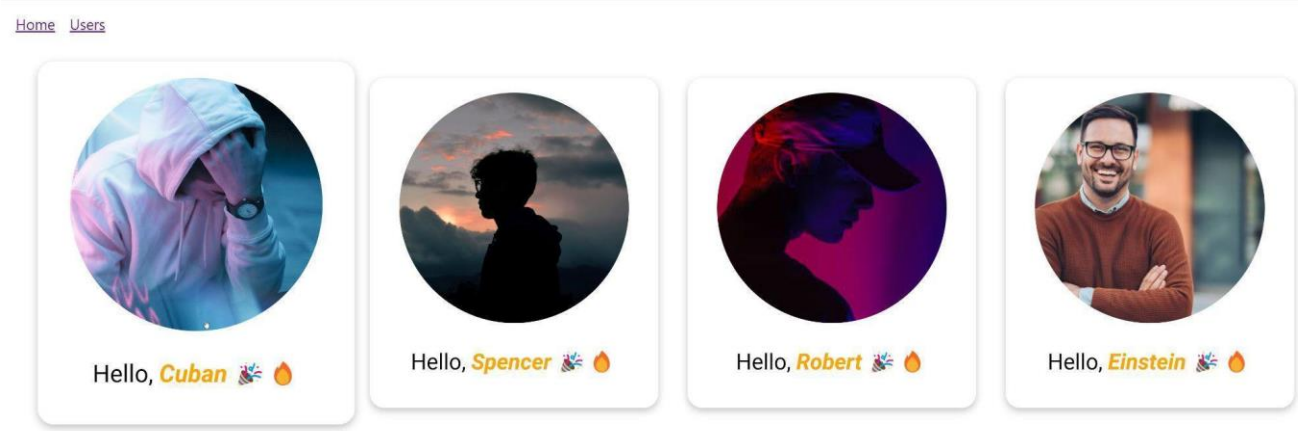


Figure 8.16: Output of Code Snippet 4

8.2.4 Displaying the User Detail from API

Similar to the `UserList` being independent of props, ensure to make `UserDetailcomponent` independent from `props` and `request` for its own data. To achieve this, utilize the feature of `mockapi.io`. In other words, if the `id` of the user is passed in the URL, only that user is printed on the screen as shown in Figure 8.17.

Figure 8.17: The User Details Printed

Thus, on clicking (`onClick`) the user card, the URL must change from utilizing `index` to `id`, that is from `/users/0 -> /users/99`. Hence, as shown in Code Snippet 5 `id={usr.id}` is passed instead of `index`. Code Snippet 5 is written in `UserList.js`.

Code Snippet 5:

```
export function UserList() {
  const [users, setUsers] = useState([]);
  useEffect(() => {
    fetch("https://640fc234864814e5b63f0d2f.mockapi.io/users")
      .then((data) => data.json())
      .then((userList) => setUsers(userList));
  }, []);
  return (
    <div className="user-list-container">
      {users.map((usr) => (
        <User name={usr.name} pic={usr.pic} id={usr.id} key={usr.id} />
```

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

Code Snippet 6 is written in `User.js`.

Code Snippet 6:

```
import { useNavigate } from "react-router-dom";
export function User({ name, pic, id }) {
  const navigate = useNavigate();
  return (
    <section
      onClick={() => navigate(`/users/${id}`)}
      className="user-container"
    >
      <img className="user-profile-pic" src={pic} alt={name} />
      <h2 className="user-name">
        Hello, <span className="user-first-name">{name}</span> 🎉🔥
      </h2>
    </section>
  );
}
```

Now on clicking (`onClick`) of the user card, Cuban, the URL changes to `/users/99`. However, no detail is displayed on the screen because data from the API is not requested, and in the process of finding the user with the index 99. To fix this part, update the Routing setup and `UserDetail`.

In the Routing setup, remove the dependency on `users` props in the `UserDetail` component.

```
<Route path="/users/:id" element={<UserDetail />} />
```

In `UserDetail`, props are removed, and replaced with the `useState user` variable as shown in Code Snippet 7. Code Snippet 7 is written in `UserDetail.js`.

Code Snippet 7:

```
import { useParams } from "react-router-dom";
import { useEffect, useState } from "react";
export function UserDetail() {
  const { id } = useParams();
  console.log(id);
  // const user = users[id];
```

```
  const [user, setUser] = useState({});
  return (
    <section className="user-detail-container">
      <img className="user-profile-pic" src={user.pic} alt={user.name} />
      <div>
        <h2 className="user-name">{user.name}</h2>
        <p>{user.bio}</p>
      </div>
    </section>
  );
}
```

Similar to what was applied for `UserList`, use the `fetch` call. However, this time, additionally pass in the `id` as shown in Code Snippet 8. Apart from that, all other steps remain the same, such as using the `useEffect` and `useState` hooks in combination. Code Snippet 8 is written in `UserDetail.js`.

Code Snippet 8:

```
export function UserDetail() {
  const { id } = useParams();
  console.log(id);
  // const user = users[id];
  const [user, setUser] = useState({});
  useEffect(() => {
    fetch(`https://640fc234864814e5b63f0d2f.mockapi.io/users/${id}`)
      .then((data) => data.json())
      .then((userInfo) => setUser(userInfo));
  }, []);
  return (
    <section className="user-detail-container">
      <img className="user-profile-pic" src={user.pic} alt={user.name} />
      <div>
        <h2 className="user-name">{user.name}</h2>
        <p>{user.bio}</p>
      </div>
    </section>
  );
}
```

Thus, when the Cuban user card is clicked, the `API` gets called and the data is displayed on the screen.

8.2.5 Deleting a User

Before implementing the function of deleting a user, plan to have three buttons:

- Info
- Delete
- Edit

Creating the Info button is easy. Move the click the handler from the section to the Info button as shown in Code Snippet 9. Code Snippet 9 is written in `User.js`.

Code Snippet 9:

```
export function User({ name, pic, id }) {
  const navigate = useNavigate();
  return (
    <section className="user-container">
      <img className="user-profile-pic" src={pic} alt={name} />
      <h2 className="user-name">
        Hello, <span className="user-first-name">{name}</span> 🎉🔥
      </h2>
      <button onClick={() => navigate(`/users/${id}`)}>Info</button>
    </section>
  );
}
```

The output of Code Snippet 9 is shown in Figure 8.18.

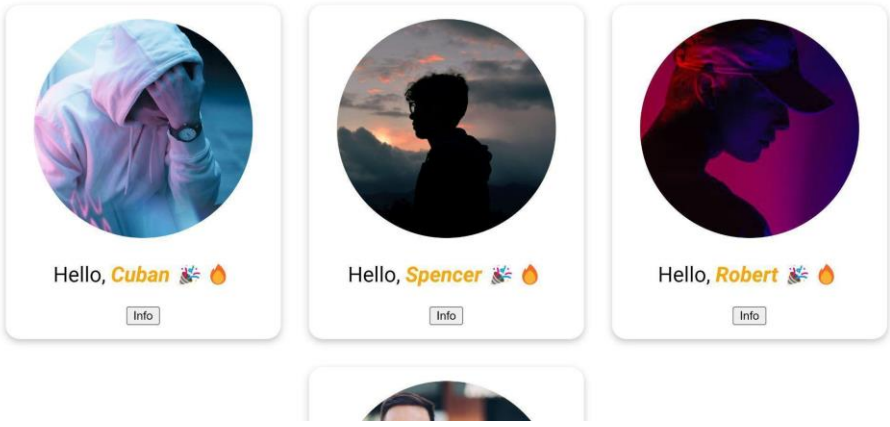


Figure 8.18 Output of Code Snippet 9

To create a delete button, instead of doing the usual way where create button is directly inside the `User`, pass the JSX as props as shown in Code Snippet 10. Here, JSX is passed to `deleteButton` prop. Code Snippet 10 is written in `UserList.js`.

Code Snippet 10:

```
export function UserList() {
```

```
  const [users, setUsers] = useState([]);
  useEffect(() => {
    fetch("https://640fc234864814e5b63f0d2f.mockapi.io/users")
      .then((data) => data.json())
      .then((userList) => setUsers(userList));
  }, []);
  return (
    <div className="user-list-container">
      {users.map((usr) => (
        <User
          name={usr.name}
          pic={usr.pic}
          id={usr.id}
          key={usr.id}
          deleteButton={<button>Delete</button>}
        />
      ))}
    </div>
  );
}
```

In the `User` component, the `deleteButton` is received as props and rendered on the screen using the template syntax `({deleteButton})` as shown in Code Snippet 11, which is written in `User.js`. Hence, this pattern in React is known as Render props.

Code Snippet 11:

```
export function User({ name, pic, id, deleteButton }) {
  const navigate = useNavigate();
  return (
    <section className="user-container">
      <img className="user-profile-pic" src={pic} alt={name} />
      <h2 className="user-name">
        Hello, <span className="user-first-name">{name}</span> 🎉🎉
      </h2>
      <button onClick={() => navigate(`/users/${id}`)}>Info</button>
      {deleteButton}
    </section>
  );
```

```
}
```

The advantage of passing the `JSX` to prop is, the `deleteButton` has access to all the variables inside `UserList`. Next, call the `API` as shown in Code Snippet 12, which is written in `UserList.js`. When the delete button is clicked:

- The `deleteUser` function is called with `id` of the user.
- The `DELETE` method is used inside fetch to complete the operation.

Code Snippet 12:

```
export function UserList() {
  const [users, setUsers] = useState([]);
  useEffect(() => {
    fetch("https://640fc234864814e5b63f0d2f.mockapi.io/users")
      .then((data) => data.json())
      .then((userList) => setUsers(userList));
  }, []);
  const deleteUser = (id) => {
    fetch(`https://640fc234864814e5b63f0d2f.mockapi.io/users/${id}`, {
      method: "DELETE",
    }).then((data) => data.json());
  };
  return (
    <div className="user-list-container">
      {users.map((usr) => (
        <User
          name={usr.name}
          pic={usr.pic}
          id={usr.id}
          key={usr.id}
          deleteButton={
            <button onClick={() => deleteUser(usr.id)}>Delete</button>
          }
        />
      ))}
    </div>
  );
}
```

The output of Code Snippet 12 is shown in Figure 8.19.

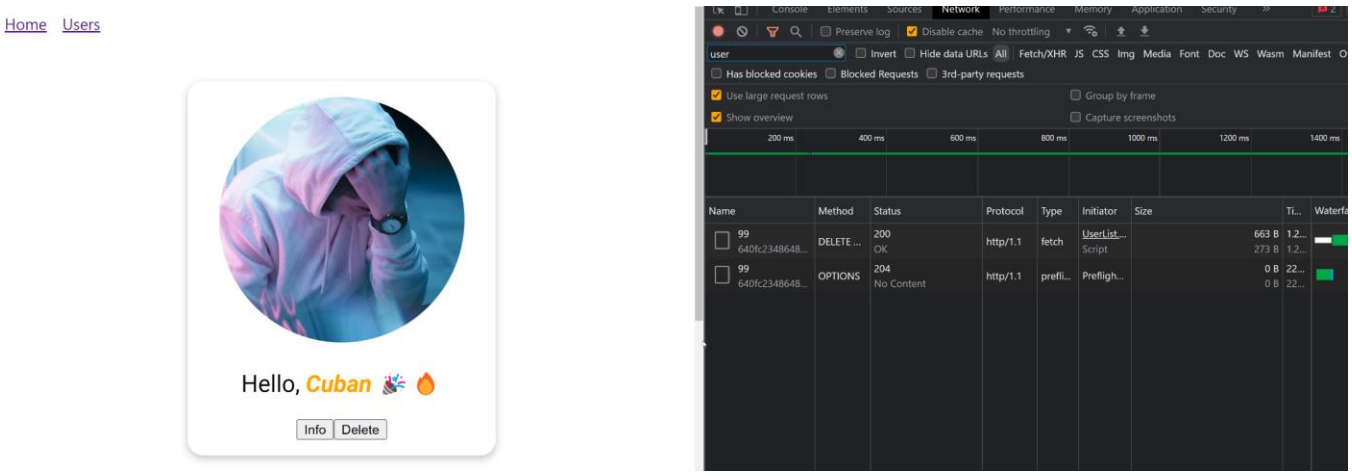


Figure 8.19: Output of Code Snippet 12

In Figure 8.20, it is observed that although API call completes successfully, the user card still remains in the view. This is because a data refresh is required. Hence, separate `getUsers` and call it inside `useEffect` to initially load the `userList` and also call it inside `deleteUser` as shown in Code Snippet 13 (written in `UserList.js`) to refresh the user data. Thus, it fetches the latest `userList` after the delete operation is completed.

Code Snippet 13:

```
export function UserList() {
  const [users, setUsers] = useState([]);
  const getUsers = () => {
    fetch("https://640fc234864814e5b63f0d2f.mockapi.io/users")
      .then((data) => data.json())
      .then((userList) => setUsers(userList));
  };
  useEffect(() => getUsers(), []);
  const deleteUser = (id) => {
    fetch(`https://640fc234864814e5b63f0d2f.mockapi.io/users/${id}`, {
      method: "DELETE",
    })
      .then((data) => data.json())
      .then(() => getUsers());
  };
  return (
    <div className="user-list-container">
      {users.map((usr, index) => (
        <User
          name={usr.name}
```

```
            pic={usr.pic}
            id={usr.id}
            key={index}
            deleteButton={
              <button onClick={() => deleteUser(usr.id)}>Delete</button>
            }
          />
        ))}
      </div>
    );
}
```

Observe that after the user is deleted, the screen also gets updated with the latest data.

8.2.6 Creating a Form for Adding User

To implement the functionality of adding a user, create a separate page, and link to navigate to the same as shown in Code Snippet 14, which is written in `App.js`.

Code Snippet 14:

```
function App() {
  return (
    <div className="App">
      <nav className="nav-list">
            …
          <Link to="/users/add">Add User</Link>
      </nav>
      <Routes>
            …
          <Route path="/users/add" element={<AddUser />} />
      </Routes>
    </div>
  );
}
```

The output of Code Snippet 14 is shown in Figure 8.20.

Figure 8.20: Output of Code Snippet 14

In `AddUser` component, create three input fields one for username, profile pic and bio as shown in Code Snippet 15, which is written in `AddUser.js`.

Code Snippet 15:

```
export function AddUser() {
  return (
    <div className="add-user-form">
      <input type="text" placeholder="Name" />
      <input type="text" placeholder="Pic" />
      <input type="text" placeholder="Bio" />
      <button>Add user</button>
    </div>
  );
}
```

The `Add user` button is added so that when it is clicked, the `POST API` call is triggered in the upcoming code snippets.

Three placeholders for `useState` and `onChange` are added respectively in order to store the value that the users type in. Moreover, when the `Add User` link is clicked, the `Add User` function is called to console the stored value as shown in Code Snippet 16, which is written in `AddUser.js`.

Code Snippet 16:

```
export function AddUser() {
  const [name, setName] = useState("");
  const [pic, setPic] = useState("");
  const [bio, setBio] = useState("");
  const addUser = () => {
    const newUser = {
      name,
      pic,
      bio,
    };
    console.log(newUser);
  };
  return (
    <div className="add-user-form">
      <input
        onChange={(event) => setName(event.target.value)}
        type="text"
```

```
        placeholder="Name"
      />
      <input
        onChange={(event) => setPic(event.target.value)}
        type="text"
        placeholder="Profile Pic Url"
      />
      <input
        onChange={(event) => setBio(event.target.value)}
        type="text"
        placeholder="Bio"
      />
      <button onClick={addUser}>Add user</button>
    </div>
  );
}
```

The output of Code Snippet 16 is shown in Figure 8.21.

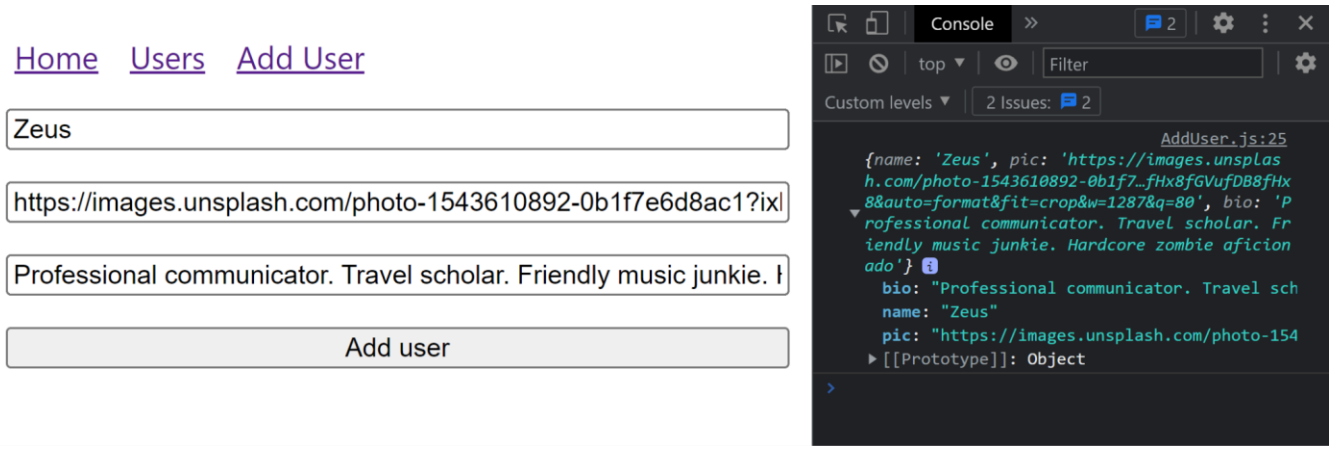


Figure 8.21: Output of Code Snippet 16

As shown in Figure 8.21, the value typed in the form after pressing the Add User button appears on the console.

Pass the `newUser` data to `POST API` to create a user. Pointers for the student are:

- The `newUser` data must be stringified (stringed together).
- Passed as JavaScript Object Notation (JSON) to the fetch method as shown in Code Snippet 17, which is written in `AddUser.js`.

Code Snippet 17:

```
const addUser = () => {
  const newUser = {
    name,
```

```
      pic,
      bio,
    };
    console.log(newUser);
    fetch(`https://640fc234864814e5b63f0d2f.mockapi.io/users`, {
      method: "POST",
      body: JSON.stringify(newUser),
      headers: {
        "Content-Type": "application/json",
      },
    }).then(() => navigate("/users"));
  };
```

The processing details of Code Snippet 17 in the Network is shown in Figure 8.22.

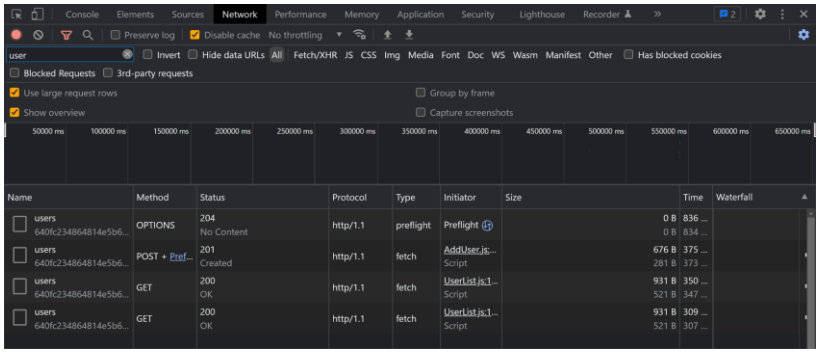


Figure 8.22: Processing of Code Snippet 17

The output of Code Snippet 17 is shown in Figure 8.23.

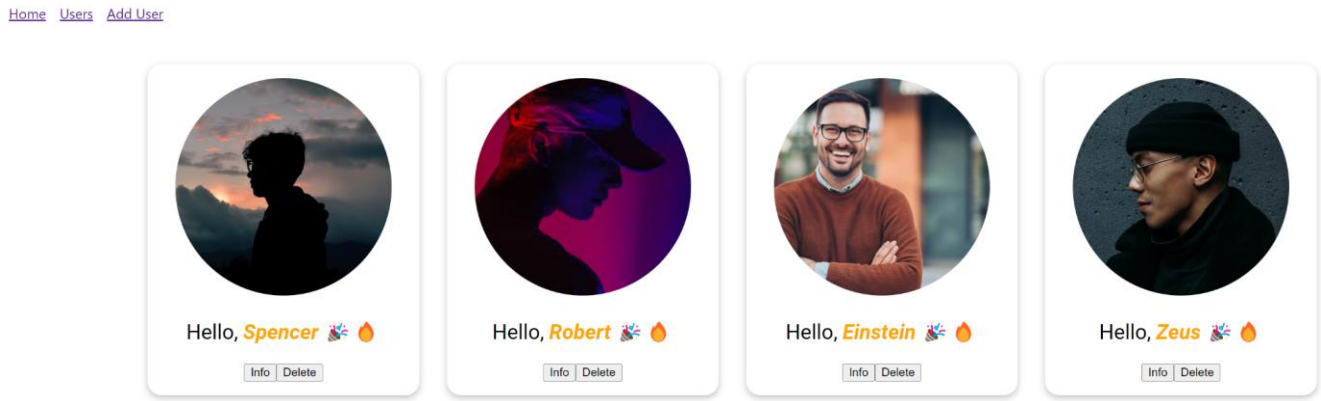


Figure 8.23: Output of Code Snippet 17

By observing Figure 8.23, it is evident that the POST API completes successfully because as one navigates to the `/users`, the latest data with new user Zeus is rendered on the screen.

8.3 Creating a More Complex CRUD Application

To simplify more complex CRUD applications, the help of two libraries are used:

- `formik` - Simplifies the form model and the integration for adding validations.
- `yup` - Makes it easy to add validations to the form.

8.3.1 Designing the Application and Data Model

To design the application and data model, the steps are as follows:

1. Install `formik` using command line.

```
npm i formik
```

9 Remove the three `useState` information and simplify it using `useFormik`. The following are shown in Code Snippet 18, which is written in `AddUser.js`:

- Inside `useFormik`, the form model is provided to the `initialValues` as shown in Code Snippet 18.
- Pause the `addUser` function and update the same later.
- `onChange` is simplified by providing the method `formik`.
- Use `handleChange` instead of having a different `setState` for each of the fields.
- Since the `formik.handleChange` method is common, ensure to help `formik` identify the field that is being updated by adding the name attribute on every input field as shown in Code Snippet 18.

Code Snippet 18:

```
import { useFormik } from "formik";
import { useNavigate } from "react-router-dom";
export function AddUser() {
  const formik = useFormik({
    initialValues: {
      name: "",
      pic: "",
      bio: "",
    },
  });
  const navigate = useNavigate();
  const addUser = () => {};
  return (
    <div className="add-user-form">
      <input onChange={formik.handleChange} type="text" placeholder="Name"
/>
      <input
        onChange={formik.handleChange}
        type="text"
```

```
        placeholder="Profile Pic Url"
      />
      <input onChange={formik.handleChange} type="text" placeholder="Bio"
/>
      <button onClick={addUser}>Add user</button>
    </div>
  );
}
```

8.3.2 Integrating formik to the Add User Form

Add an inbuilt validation: `required` to name `input` field and test it as shown in Code Snippet 19, before adding custom validations. Code Snippet 19 is written in `AddUser.js`.

What are validations?

Validations are the error messages that are shown if the form is submitted incomplete or incorrectly. For example:

- Username must be unique
- The email must be valid
- Password must contain special signs, password, must match, and so on

Why Validations?

Validations are given so that:

- The user does not submit any junk data to the app. For example: allowing users to submit the form without filling in the username.
- Users get confident that all the information provided by them is valid and they did not miss any critical information.
  For example: After submitting an application for a passport, if the government employee says after a month that the form submitted was wrong, this would cause frustration to the user. Hence, providing the error messages while entering the information, then and there improves the user experience.

Code Snippet 19:

```
<input
  onChange={formik.handleChange}
  type="text"
  placeholder="Name"
  required
/>
```

Unfortunately, the validation does not get triggered because the input fields are not surrounded by form elements. Hence, once the change is made as shown in Code Snippet 20, now validation is triggered when users try to submit the form without filling in the name field as shown in Figure 8.25.

Code Snippet 20:

```
<form className="add-user-form">
```

```
      ...
    </form>
```

The output of Code Snippets 19 and 20 is shown in Figure 8.24.

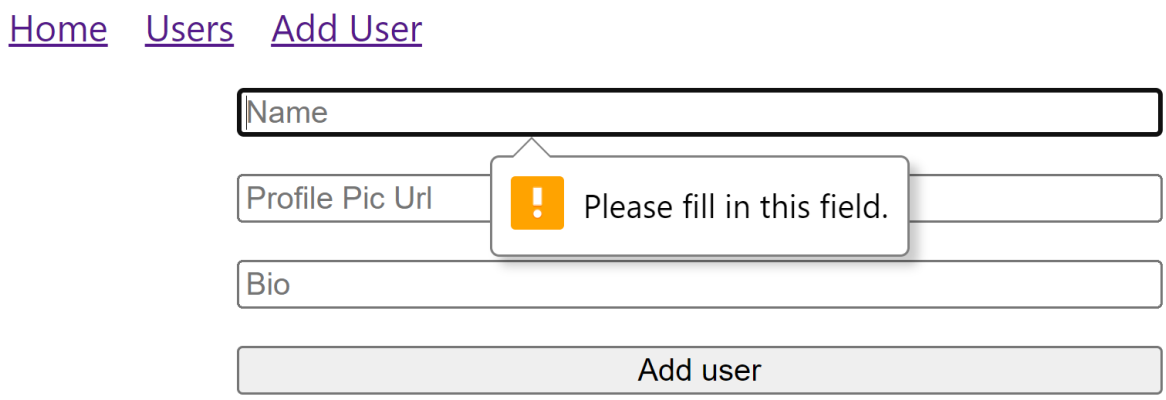


Figure 8.24: Output of Code Snippets 19 and 20

To console the form values, add `type="submit"` Add user button. Hence, on clicking this button, the `onSubmit` event would be triggered, which in turn calls the `formik.handleSubmit` as shown in Code Snippet 21, written in `AddUser.js`.

Finally, `formik` takes over from there and calls the `onSubmit` callback. The `onSubmit` callback gets the form `values` that are consoled as shown Figure 8.26.

Code Snippet 21:

```
export function AddUser() {
  const formik = useFormik({
    initialValues: {
      name: "",
      pic: "",
      bio: "",
    },
    onSubmit: (values) => {
      console.log(values);
    },
  });
  const navigate = useNavigate();
  const addUser = () => {};
  return (
    <form onSubmit={formik.handleSubmit} className="add-user-form">
      <input
        name="name"
```

```
        type="text"
        placeholder="Name"
        required
      />
      <input
        onChange={formik.handleChange}
        name="pic"
        type="text"
        placeholder="Profile Pic Url"
      />
      <input
        onChange={formik.handleChange}
        name="bio"
        type="text"
        placeholder="Bio"
      />
      <button type="submit">Add user</button>
    </form>
  );
}
```

The output of Code Snippet 21 is shown in Figure 8.25.

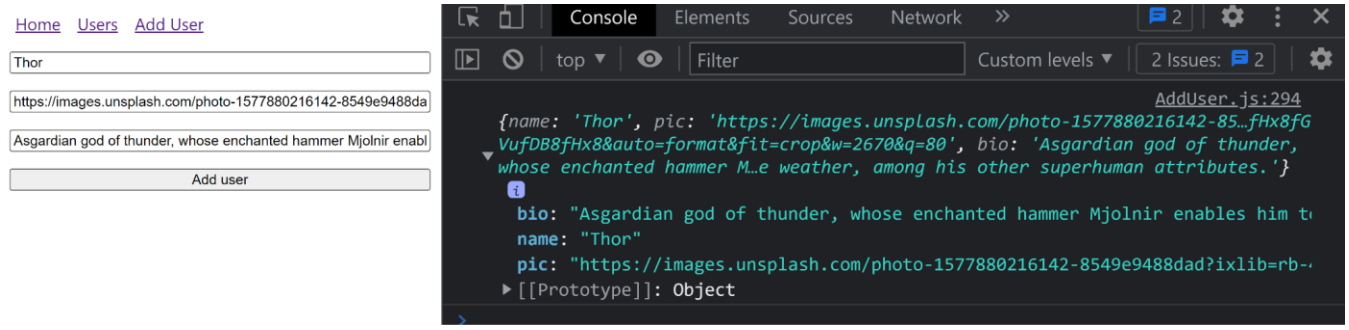


Figure 8.25: Output of Code Snippet 21

8.3.3 Validating Form Data Before Submission

Install `yup` using the command for adding validations to the add user form.

```
npm i yup
```

To add validation to the form, add `validationSchema`. This `validationSchema` ensures that only when all the validations pass, only then the `onSubmit` callback would be called.

The `userValidationSchema` is provided to `validationSchema` to implement in `AddUser`. Inside `userValidationSchema` as shown in Code Snippet 22 (written in `AddUser.js`), the code indicates that:

- The `name` field must be a string, and it is required (must not be blank).
- The `pic` field must be a string, it must follow `URL` pattern, and it is a requirement (must not be blank).
- The `bio` field must be a string, it must be at least 10 characters long, and it is requirement (must not be blank).

Moreover, to display the error message on the screen, `{JSON.stringify(formik.errors)}` is given inside `AddUser`.

Code Snippet 22:

```
import { object, string } from "yup";
const userValidationSchema = object({
  name: string().required(),
  pic: string().url().required(),
  bio: string().min(10).required(),
});
export function AddUser() {
  const formik = useFormik({
    initialValues: {
      name: "",
      pic: "",
      bio: "",
    },
    onSubmit: (values) => {
      console.log(values);
    },
    validationSchema: userValidationSchema,
  });
  const navigate = useNavigate();
  const addUser = () => {};
  return (
    <form onSubmit={formik.handleSubmit} className="add-user-form">
      …
      <p>Errors</p>
      <pre>{JSON.stringify(formik.errors)}</pre>
    </form>
  );
}
```

The output of Code Snippet 22 (without errors) is shown in Figure 8.26.

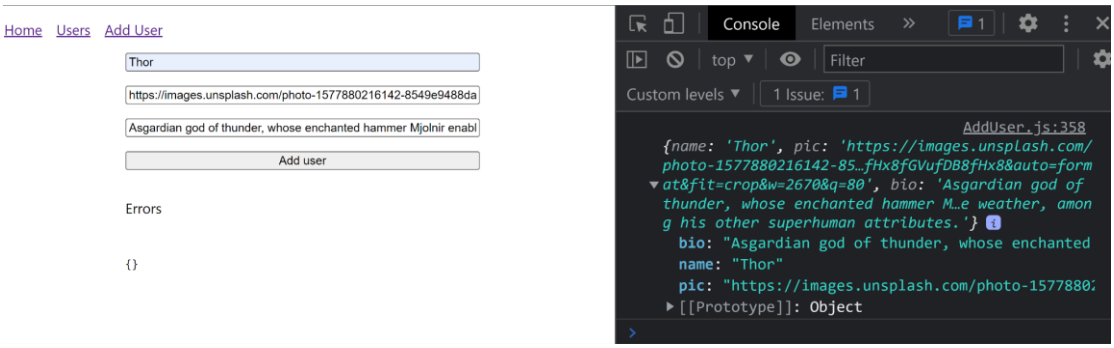


Figure 8.26: Output of Code Snippet 22

When there are errors, the `form` value is not allowed to be consoled, and when there are no errors the form value is consoled as shown in Figure 8.27.

8.3.4 Handling Errors and Displaying Errors

Currently, there is a problem while displaying error messages. Code Snippet 23 allows to preemptively show all the error messages just when the user starts typing on any of the fields as shown in Figure 8.28.

Code Snippet 23:

```
export function AddUser() {
  const formik = useFormik({
    initialValues: {
      name: "",
      pic: "",
      bio: "",
    },
    onSubmit: (values) => {
      console.log(values);
    },
    validationSchema: userValidationSchema,
  });
  const navigate = useNavigate();
  const addUser = () => {};
  return (
    <form onSubmit={formik.handleSubmit} className="add-user-form">
      <input
        onChange={formik.handleChange}
        name="name"
        type="text"
        placeholder="Name"
```

```
        required
      />
      <p>{formik.errors.name}</p>
      <input
        onChange={formik.handleChange}
        name="pic"
        type="text"
        placeholder="Profile Pic Url"
      />
      <p>{formik.errors.pic}</p>
      <input
        onChange={formik.handleChange}
        name="bio"
        type="text"
        placeholder="Bio"
      />
      <p>{formik.errors.bio}</p>
      <button type="submit">Add user</button>
    </form>
  );
}
```

Output of Code Snippet 23 is shown in Figure 8.27.

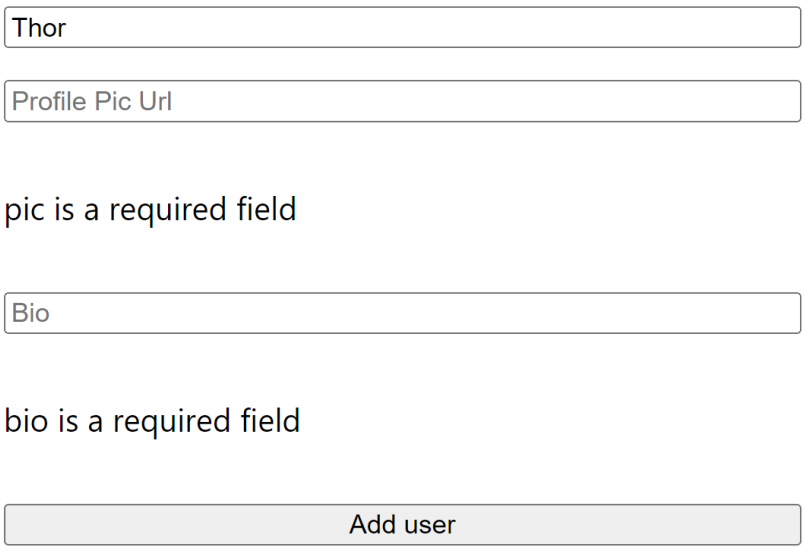


Figure 8.27: Output of Code Snippet 23

Thus, to avoid this preemptive behavior, tracking of the fact whether the user has touched the field or not is required. To achieve this, the `onBlur` event is used. This event is triggered when a user clicks on the field and then clicks anywhere outside of the field. As shown in Code Snippet 24 (written in AddUser.js), the `onBlur` event is provided to `formik.handleBlur`, which in turn makes the `formik.touched` property of that field to be `true`.

Code Snippet 24:

```
export function AddUser() {
  const formik = useFormik({
    initialValues: {
      name: "",
      pic: "",
      bio: "",
    },
    onSubmit: (values) => {
      console.log(values);
    },
    validationSchema: userValidationSchema,
  });
  const navigate = useNavigate();
  const addUser = () => {};
  return (
    <form onSubmit={formik.handleSubmit} className="add-user-form">
      <input
        onChange={formik.handleChange}
       onBlur={formik.handleBlur}
        name="name"
        type="text"
        placeholder="Name"
        required
      />
      {formik.touched.name && formik.errors.name ? (
        <p>{formik.errors.name}</p>
      ) : null}
      <input
        onChange={formik.handleChange}
       onBlur={formik.handleBlur}
        name="pic"
```

```
        type="text"
        placeholder="Profile Pic Url"
      />
      {formik.touched.pic && formik.errors.pic ? (
        <p>{formik.errors.pic}</p>
      ) : null}
      <input
        onChange={formik.handleChange}
        onBlur={formik.handleBlur}
        name="bio"
        type="text"
        placeholder="Bio"
      />
      {formik.touched.bio && formik.errors.bio ? (
        <p>{formik.errors.bio}</p>
      ) : null}
      <button type="submit">Add user</button>
    </form>
  );
}
```

Following Code Snippet 24, the error message is seen on the screen only when the user has touched the field and also if the field has an error.

Since the validation is working as expected, now bring back the `POST API` call inside `addUser` as shown in Code Snippet 25 (`AddUser.js`) and pass the `newUser` when the form is submitted successfully.

Code Snippet 25:

```
export function AddUser() {
  const formik = useFormik({
    initialValues: {
      name: "",
      pic: "",
      bio: "",
    },
    onSubmit: (newUser) => {
      addUser(newUser);
    },
    validationSchema: userValidationSchema,
```

```
  });
  const navigate = useNavigate();
  const addUser = (newUser) => {
    console.log(newUser);
    fetch(`https://640fc234864814e5b63f0d2f.mockapi.io/users`, {
      method: "POST",
      body: JSON.stringify(newUser),
      headers: {
        "Content-Type": "application/json",
      },
    }).then(() => navigate("/users"));
  };
  return (
    <form onSubmit={formik.handleSubmit} className="add-user-form">
      <input
        onChange={formik.handleChange}
        onBlur={formik.handleBlur}
        name="name"
        type="text"
        placeholder="Name"
        required
      />
      {formik.touched.name && formik.errors.name ? (
        <p>{formik.errors.name}</p>
      ) : null}
      <input
        onChange={formik.handleChange}
        onBlur={formik.handleBlur}
        name="pic"
        type="text"
        placeholder="Profile Pic Url"
      />
      {formik.touched.pic && formik.errors.pic ? (
        <p>{formik.errors.pic}</p>
      ) : null}
      <input
```

```
        onChange={formik.handleChange}
        onBlur={formik.handleBlur}
        name="bio"
        type="text"
        placeholder="Bio"
      />
      {formik.touched.bio && formik.errors.bio ? (
        <p>{formik.errors.bio}</p>
      ) : null}
      <button type="submit">Add user</button>
    </form>
  );
}
```

The output of Code Snippets 24, 25, and 26 is shown in Figure 8.28.

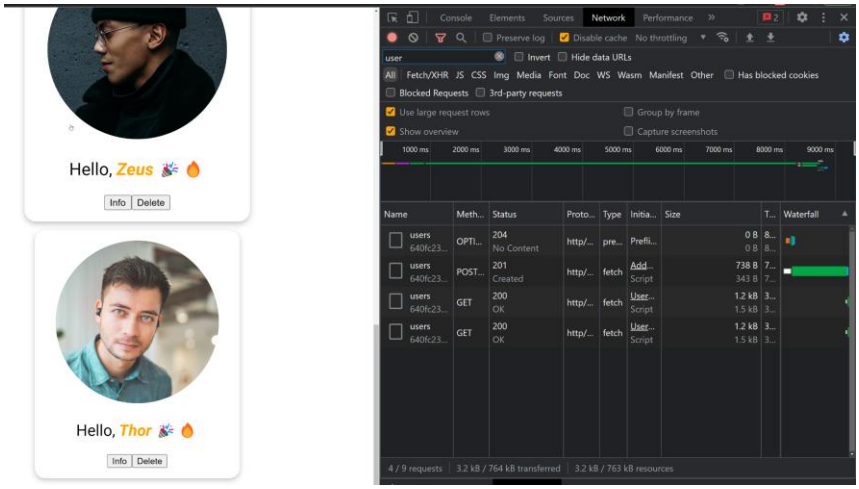


Figure 8.28: Output of Code Snippets 24, 25, and 26

The code in Code Snippet 25 could be further improved in terms of readability by using the destructuring technique as shown in Code Snippet 26.

Code Snippet 26:

```
export function AddUser() {
  const { handleSubmit, handleChange, errors, touched,
          handleBlur } = useFormik({
    initialValues: {
      name: "",
      pic: "",
      bio: "",
    },
```

```
    onSubmit: (newUser) => {
      addUser(newUser);
    },
    validationSchema: userValidationSchema,
  });
  const navigate = useNavigate();
  const addUser = (newUser) => {
    console.log(newUser);
    fetch(`https://640fc234864814e5b63f0d2f.mockapi.io/users`, {
      method: "POST",
      body: JSON.stringify(newUser),
      headers: {
        "Content-Type": "application/json",
      },
    }).then(() => navigate("/users"));
  };
  return (
    <form onSubmit={handleSubmit} className="add-user-form">
      <input
        onChange={handleChange}
        onBlur={handleBlur}
        name="name"
        type="text"
        placeholder="Name"
        required
      />
      {touched.name && errors.name ? <p>{errors.name}</p> : null}
      <input
        onChange={handleChange}
        onBlur={handleBlur}
        name="pic"
        type="text"
        placeholder="Profile Pic Url"
      />
      {touched.pic && errors.pic ? <p>{errors.pic}</p> : null}
      <input onChange={handleChange}
```

```
            onBlur={handleBlur} name="bio" type="text" placeholder="Bio" />
      {touched.bio && errors.bio ? <p>{errors.bio}</p> : null}
      <button type="submit">Add user</button>
    </form>
  );
}
```

8.3.5 Update User

To implement the functionality of updating a user, create a separate `editButton` button, similar to that of `deleteButton` in the `UserList` component as shown in Code Snippet 27, which is written in `UserList.js`.

Code Snippet 27:

```
        <User
          name={usr.name}
          pic={usr.pic}
          id={usr.id}
          key={index}
          deleteButton={
            <button onClick={() => deleteUser(usr.id)}>Delete</button>
          }
          editButton={
            <button onClick={() => navigate(`/users/edit/${usr.id}`)}>
              Edit
            </button>
          }
      />
```

Add the `editButton` next to the delete button as shown in Code Snippet 28 (written in `User.js`). Thus, on clicking the edit button, it displays a “404 - error” page.

Code Snippet 28:

```
export function User({ name, pic, id, deleteButton, editButton }) {
      …
      return (
          <section className="user-container">
            …
            {deleteButton}
            {editButton}
          </section>
```

```
        );
}
```

To fix it, similar to what was done in the `AddUser` form, add a separate page for `EditUser`. Hence, the routing setup is done.

```
        <Route path="/users/edit/:id" element={<EditUser />} />
```

Create a simple `EditUser` component as shown in Code Snippet 29 (written in `EditUser.js`) to test out, if the routing works.

Code Snippet 29:

```
import { useFormik } from "formik";
import { useNavigate, useParams } from "react-router-dom";
import { object, string } from "yup";
import { useEffect} from "react";
export function EditUser() {
  const { id } = useParams();
  console.log(id);
  return <h1>Editing user {id}</h1>;
}
```

On clicking the edit button on Cuban, it is routed to the edit page with a message `Editing user 100` on the screen. This confirms that the connection is working in the code.

To implement `EditUser`, start with `AddUser`, as the template. Hence, keep the code same as that of `AddUser` and only change the text inside the button to **`Save`** as shown in Code Snippet 30 (written in EditUser.js).

Code Snippet 30:

```
export function EditUser() {
…
return (
    <form onSubmit={handleSubmit} className="add-user-form">
      …
       <button type="submit">Save</button>
    </form>
  );
}
```

In `EditUser`, it is required to pre-fill the form with data of the user on which the edit button is clicked. Hence, as a first step, test out the pre-fill by providing `initialValues` to the form, and assigning the value for each input field appropriately (Example: `value={values.name}`) as shown in Code Snippet 31 (written in `EditUser.js`).

Code Snippet 31:

```
export function EditUser() {
  const { id } = useParams();
  console.log(id);
  const { handleSubmit, handleChange, errors, touched, values, handleBlur
 } = useFormik({
    initialValues: {
      name: "Spencer",
      pic: "https://images.unsplash.com/photo-1529665253569-
6d01c0eaf7b6?ixlib=rb-
4.0.3&ixid=MnwxMjA3fDB8MHxzZWFyY2h8NHx8cHJvZmlsZXxlbnwwfHwwfHw%3D&w=1000&q=80",
      bio: "Award-winning web lover. Thinker. Social media advocate. Creator.
Bacon scholar. Zombie geek",
    },
    onSubmit: (newUser) => {
      addUser(newUser);
    },
    validationSchema: userValidationSchema,
  });
  const navigate = useNavigate();
  const addUser = (newUser) => {
    console.log(newUser);
    fetch(`https://640fc234864814e5b63f0d2f.mockapi.io/users`, {
      method: "POST",
      body: JSON.stringify(newUser),
      headers: {
        "Content-Type": "application/json",
      },
    }).then(() => navigate("/users"));
  };
  return (
    <form onSubmit={handleSubmit} className="add-user-form">
      <input
        value={values.name}
        onChange={handleChange}
        onBlur={handleBlur}
        name="name"
```

```
        type="text"
        placeholder="Name"
        required
      />
      {touched.name && errors.name ? <p>{errors.name}</p> : null}
      <input
        value={values.pic}
        onChange={handleChange}
        onBlur={handleBlur}
        name="pic"
        type="text"
        placeholder="Profile Pic Url"
      />
      {touched.pic && errors.pic ? <p>{errors.pic}</p> : null}
      <input
        value={values.bio}
        onChange={handleChange}
        onBlur={handleBlur}
        name="bio"
        type="text"
        placeholder="Bio"
      />
      {touched.bio && errors.bio ? <p>{errors.bio}</p> : null}
      <button type="submit">Save</button>
    </form>
  );
}
```

The form is now pre-filled with the user details of Spencer. Hence, take it one step further by fetching the user data from the `API`, as in the case of `UserDetails`. Thus, as shown in Code Snippet 32 (written in `EditUser.js`), the data is fetched and assigned to the `initialValues`.

Code Snippet 32:

```
export function EditUser() {
  const { id } = useParams();
  console.log(id);
  const [user, setUser] = useState({});
  useEffect(() => {
```

```
    fetch(`https://640fc234864814e5b63f0d2f.mockapi.io/users/${id}`)
      .then((data) => data.json())
      .then((userInfo) => setUser(userInfo));
  }, []);
  console.log(user, user.name);
  const { handleSubmit, handleChange, errors, touched, values, handleBlur
    } = useFormik({
    initialValues: {
      name: user.name,
      pic: user.pic,
      bio: user.bio,
    },
    onSubmit: (newUser) => {
      addUser(newUser);
    },
    validationSchema: userValidationSchema,
  });
  const navigate = useNavigate();
  const addUser = (newUser) => {
    console.log(newUser);
    fetch(`https://640fc234864814e5b63f0d2f.mockapi.io/users`, {
      method: "POST",
      body: JSON.stringify(newUser),
      headers: {
        "Content-Type": "application/json",
      },
    }).then(() => navigate("/users"));
  };
  return (
    <form onSubmit={handleSubmit} className="add-user-form">
      {user.name}
      <input
        value={values.name}
        onChange={handleChange}
        onBlur={handleBlur}
        name="name"
```

```
        type="text"
        placeholder="Name"
        required
      />
      {touched.name && errors.name ? <p>{errors.name}</p> : null}
      <input
        value={values.pic}
        onChange={handleChange}
        onBlur={handleBlur}
        name="pic"
        type="text"
        placeholder="Profile Pic Url"
      />
      {touched.pic && errors.pic ? <p>{errors.pic}</p> : null}
      <input
        value={values.bio}
        onChange={handleChange}
        onBlur={handleBlur}
        name="bio"
        type="text"
        placeholder="Bio"
      />
      {touched.bio && errors.bio ? <p>{errors.bio}</p> : null}
      <button type="submit">Save</button>
    </form>
  );
}
```

Output of Code Snippets 30 to 32 is shown in Figure 8.29.

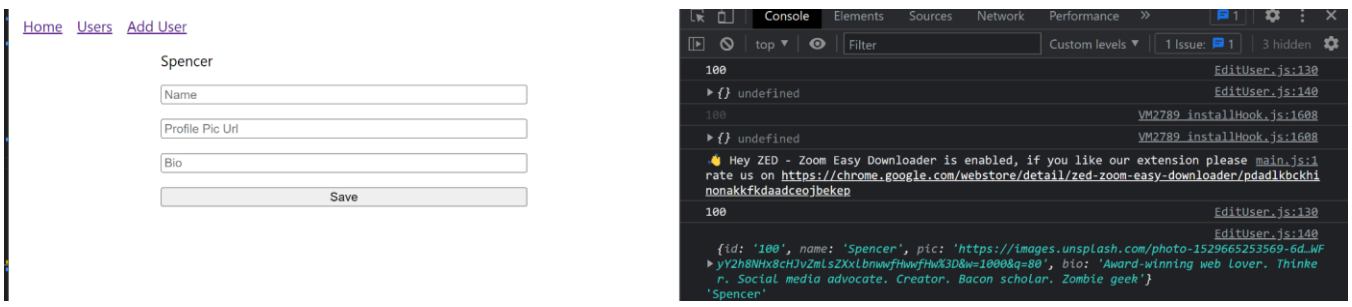


Figure 8.29: Output of Code Snippets 30, 31, and 32

Unfortunately, the data is not displayed inside the form fields, but it is consoled and shown on the screen since `{user.name}` is mentioned below the form element in Code Snippet 32. This happens because the data takes time and by the time data is retrieved, the `initialValues` is already set as follows:

```
{ name: undefined, pic: undefined,  bio: undefined }
```

Thus, the form fields are empty. To fix this, it is required to wait for the data to load and then render the edit form. Hence, to achieve this, break the component into two where `EditUser` and `EditUserForm`. `EditUser` will fetch the data and provide it as props to `EditUserForm`. Additionally this time, `EditUserForm` is conditionally rendered when the data is available as shown in Code Snippet 33 (written in `EditUser.js`).

Code Snippet 33:

```
export function EditUser() {
  const { id } = useParams();
  console.log(id);
  const [user, setUser] = useState(null);
  useEffect(() => {
    fetch(`https://640fc234864814e5b63f0d2f.mockapi.io/users/${id}`)
      .then((data) => data.json())
      .then((userInfo) => setUser(userInfo));
  }, []);
  return user ? <EditUserForm user={user} /> : <p>Loading...</p>;
}
export function EditUserForm({ user }) {
  const { handleSubmit, handleChange, errors, touched, values, handleBlur
 } = useFormik({
    initialValues: {
      name: user.name,
      pic: user.pic,
      bio: user.bio,
    },
    onSubmit: (newUser) => {
      addUser(newUser);
    },
    validationSchema: userValidationSchema,
  });
  const navigate = useNavigate();
  const addUser = (newUser) => {
    console.log(newUser);
```

```
    fetch(`https://640fc234864814e5b63f0d2f.mockapi.io/users`, {
      method: "POST",
      body: JSON.stringify(newUser),
      headers: {
        "Content-Type": "application/json",
      },
    }).then(() => navigate("/users"));
  };
  return (
    <form onSubmit={handleSubmit} className="add-user-form">
      {user.name}
      <input
        value={values.name}
        onChange={handleChange}
        onBlur={handleBlur}
        name="name"
        type="text"
        placeholder="Name"
        required
      />
      {touched.name && errors.name ? <p>{errors.name}</p> : null}
      <input
        value={values.pic}
        onChange={handleChange}
        onBlur={handleBlur}
        name="pic"
        type="text"
        placeholder="Profile Pic Url"
      />
      {touched.pic && errors.pic ? <p>{errors.pic}</p> : null}
      <input
        value={values.bio}
        onChange={handleChange}
        onBlur={handleBlur}
        name="bio"
        type="text"
```

```
        placeholder="Bio"
      />
      {touched.bio && errors.bio ? <p>{errors.bio}</p> : null}
      <button type="submit">Save</button>
    </form>
  );
}
```

Hence, as shown in Figure 8.30 and Figure 8.31, `Loading...` is displayed when data is being fetched and the form is displayed with data when the `fetch` operation is complete.

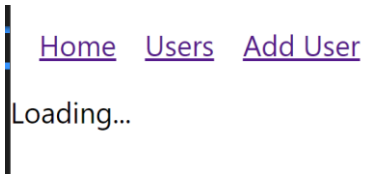


Figure 8.30: Loading the User Data

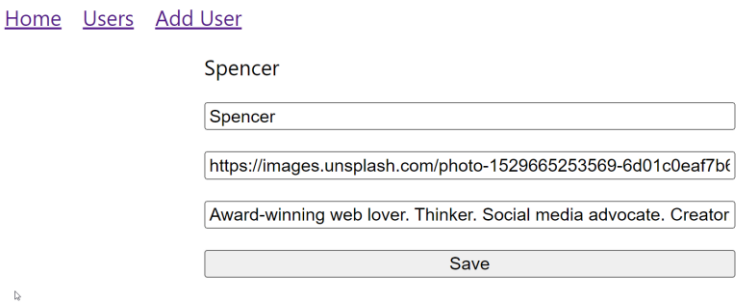


Figure 8.31: Output of Code Snippet 33 with Fetched User Details

Finally, modify the `addUser` as `updateUser` where the `PUT` method is used to update the user data as shown in Code Snippet 34 (written in `EditUser.js`).

Code Snippet 34:

```
export function EditUser() {
  const { id } = useParams();
  console.log(id);
  const [user, setUser] = useState(null);
  useEffect(() => {
    fetch(`https://640fc234864814e5b63f0d2f.mockapi.io/users/${id}`)
      .then((data) => data.json())
      .then((userInfo) => setUser(userInfo));
  }, []);
  return user ? <EditUserForm user={user} /> : <p>Loading...</p>;
}
export function EditUserForm({ user }) {
```

```
const { handleSubmit, handleChange, errors, touched, values } = useFormik({
  initialValues: {
    name: user.name,
    pic: user.pic,
    bio: user.bio,
  },
  onSubmit: (updatedUser) => {
    updateUser(updatedUser);
  },
  validationSchema: userValidationSchema,
});
const navigate = useNavigate();

const updateUser = (updatedUser) => {
  console.log(updatedUser);
  fetch(`https://640fc234864814e5b63f0d2f.mockapi.io/users/${user.id}`, {
    method: "PUT",
    body: JSON.stringify(updatedUser),
    headers: {
      "Content-Type": "application/json",
    },
  }).then(() => navigate("/users"));
};
return (
  <form onSubmit={handleSubmit} className="add-user-form">
    <input
      value={values.name}
      onChange={handleChange}
      onBlur={handleBlur}
      name="name"
      type="text"
      placeholder="Name"
      required
    />
    {touched.name && errors.name ? <p>{errors.name}</p> : null}
    <input
```

```
        value={values.pic}
        onChange={handleChange}
        onBlur={handleBlur}
        name="pic"
        type="text"
        placeholder="Profile Pic Url"
      />
      {touched.pic && errors.pic ? <p>{errors.pic}</p> : null}
      <input
        value={values.bio}
        onChange={handleChange}
        onBlur={handleBlur}
        name="bio"
        type="text"
        placeholder="Bio"
      />
      {touched.bio && errors.bio ? <p>{errors.bio}</p> : null}
      <button type="submit">Save</button>
    </form>
  );
}
```

The output of Code Snippet 34 is shown in Figure 8.32.

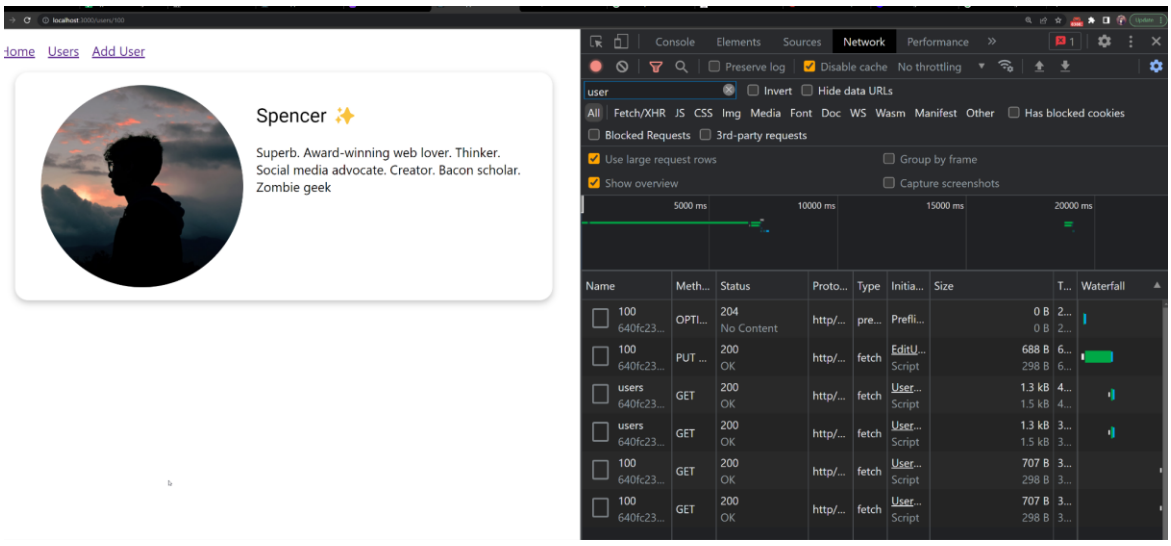


Figure 8.32: Output of Code Snippet 34

8.4 Summary

- ✓ To organize the app, the code is split into different files using the move to file feature in Visual Studio Code Editor.
- ✓ The `mockapi.io` is used to mock the data and easily test CRUD operation in the app.
- ✓ Four major CRUD methods are: `GET, POST, PUT, and DELETE.`
- ✓ Using `formik and yup` makes building forms easier.
- ✓ Validations are added to:
    - Avoid junk data.
    - Improve user experience.

8.5 Check Your Knowledge

1. Which HTTP method is to be used for updating data?

A	`POST`
B	`PUT`
C	`DELETE`
D	`GET`

2. Which library makes the implementation of validation easier?

A	`formik`
B	`mockapi.io`
C	`yup`
D	`POST`

3. After deleting data what must be done to keep the data fresh on the screen?

A	Refresh the screen
B	Call the `GET` API associated with the data
C	Call the `POST` API associated with the data
D	Close the window and reopen the window

4. The `formik` is used for handling forms and in modelling the data.

A	True
B	False

5. All the apps in the world are CRUD apps.

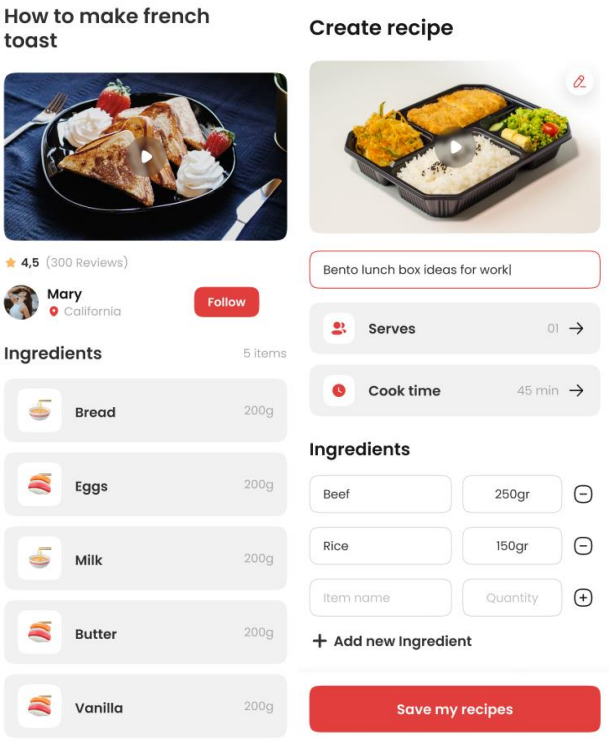
A	True
B	False

Answers

1	B
2	C
3	B
4	A
5	A

Try It Yourself

Build a recipe app with all the CRUD operations as per the design provided in the Figure with form validations using `formik` and `yup`.



Appendix

Case Study:

Design a Travel Website for Mark who is a owner of a travel company named Traveler's Destination. The Website has fully featured functions that will activate the travelling bug with vibrant imagery. This website should contain the highlights of some important places along with high quality photography and allow people to book their dream destination within their budgets. The Website should provide full customer support by providing options of easy payment, booking of destinations as per their convenience. The Website should also provide a full virtual tour of a place through different videos and images.

People all over the world are fond of trip and tourism. People frequently find it difficult to search for the best places. To address the issue, Mark has adopted the travel Website which will offer the best places among others.

The main idea of this case study is to create a Website using the technologies learnt in all sessions to continuously and give pleasurable quality excursions/trips on time and on budget for all customers. This will help to develop enthusiastically satisfied customers in the long term.

The Website designed should have following features:

- Full featured Wishlist cart
- Place pagination
- User profile with bookings
- Admin place management
- Admin user management
- Admin Booked Order details page
- Mark booking orders
- Checkout process (shipping, payment method, and so on)
- PayPal / credit card integration
- Database (places & users)

The Technology to be used:

- ReactJS
- Node