**Course Outline: React Fundamentals - Building Dynamic User Interfaces (For Freshers)**

**Course Duration**: 4-6 Weeks (20-30 hours)

**Target Audience**: Beginners with basic JavaScript knowledge.

**Week 1: Introduction to React and JSX**

* **Module 1.1**: Introduction to React
  + What is React? Why use React?
  + React ecosystem and the power of components.
  + The concept of a "Single Page Application" (SPA).
* **Module 1.2**: Setting up the Development Environment
  + Installing Node.js and npm.
  + Creating a new React application using create-react-app.
  + Overview of project structure.
* **Module 1.3**: JSX Syntax
  + Understanding JSX (JavaScript XML).
  + Embedding expressions in JSX.
  + Rules of JSX (className, self-closing tags, etc.).
* **Module 1.4**: Rendering React Elements
  + Rendering elements to the DOM using ReactDOM.render().
  + Creating and rendering a simple "Hello World" React app.

**Assignment**: Build a "Welcome" page with a greeting message using JSX.

**Week 2: Components and Props**

* **Module 2.1**: Introduction to Components
  + What are Components?
  + Functional components vs. Class components.
  + How to create and use a component.
* **Module 2.2**: Understanding Props
  + What are Props?
  + Passing data between components via props.
  + Using props for dynamic content.
* **Module 2.3**: Nested Components and Composition
  + Passing data between parent and child components.
  + Component composition and how React promotes reusable UI elements.

**Assignment**: Create two components (e.g., Header and Footer) and pass a dynamic message as a prop from a parent component.

**Week 3: State and Lifecycle Methods**

* **Module 3.1**: Understanding State
  + What is state in React?
  + How to set and update state using useState.
  + When to use state (handling user inputs, dynamic content).
* **Module 3.2**: Using Lifecycle Methods (for Class Components)
  + What are lifecycle methods?
  + Key lifecycle methods: componentDidMount(), componentDidUpdate(), componentWillUnmount().
* **Module 3.3**: State Management with Functional Components and useState
  + Introduction to React hooks: useState hook.
  + Setting initial state and updating state dynamically.
* **Module 3.4**: Introduction to useEffect
  + What is useEffect and why is it used?
  + Running side effects (API calls, subscriptions).
  + Cleanup functions in useEffect.

**Assignment**: Create a simple counter app using useState and update the page title using useEffect.

**Week 4: Handling Events and Conditional Rendering**

* **Module 4.1**: Event Handling in React
  + Handling events (onClick, onChange, etc.).
  + Passing arguments to event handlers.
  + Synthetic events in React.
* **Module 4.2**: Conditional Rendering
  + Rendering elements based on conditions (e.g., if statements, ternary operators).
  + Short-circuiting and conditional rendering with && operator.
* **Module 4.3**: Handling Forms in React
  + Controlled vs. Uncontrolled components.
  + Handling form submissions and user inputs.
  + Managing form state and validation.

**Assignment**: Create a login form with conditional rendering of a success or error message based on user input.

**Week 5: Lists, Keys, and Forms in React**

* **Module 5.1**: Rendering Lists in React
  + Rendering arrays with map() function.
  + Using keys to optimize rendering performance.
* **Module 5.2**: Dynamic Forms and User Inputs
  + Building dynamic forms (add/remove tasks or items).
  + Managing form data as arrays in state.

**Assignment**: Build a simple To-Do list app with the ability to add and remove tasks dynamically.

**Week 6: Basics of Hooks (useState, useEffect) & Styling Components**

* **Module 6.1**: Using the useState Hook
  + Recap of useState.
  + Complex state with arrays and objects.
* **Module 6.2**: Using the useEffect Hook
  + More complex use of useEffect.
  + Fetching data from an API and displaying it in the UI.
* **Module 6.3**: Styling Components in React
  + Basic CSS styling in React.
  + Introduction to **CSS Modules** for scoped styling.
  + Introduction to **Styled Components** for writing CSS in JavaScript.

**Assignment**: Style your To-Do list app with CSS Modules or Styled Components.

**Final Project: Build a Recipe App**

* **Project Overview**:
  + Create a Recipe App where users can view a list of recipes, add new recipes, and delete them. Each recipe will include a title, ingredients, and preparation instructions.
  + Utilize the concepts learned throughout the course, including state, props, event handling, lists, and basic hooks.
* **Project Requirements**:
  + **Components**: Create multiple components (e.g., RecipeList, RecipeCard, AddRecipeForm).
  + **State**: Manage the list of recipes in state using useState.
  + **Event Handling**: Handle adding and deleting recipes via form inputs and button clicks.
  + **Styling**: Apply styling using CSS or Styled Components.
  + **Conditional Rendering**: Render a message when no recipes are present.
  + **Hooks**: Use useEffect to simulate fetching data from a mock API (e.g., JSON file or hardcoded list).

**Final Project Deliverable**:

* Submit the fully functional Recipe App with clean code, proper use of React concepts, and a user-friendly interface.
* **Week 1: Introduction to React and JSX**

**Week 1: Introduction to React and JSX**

### ****Module 1.1: What is React? Why Use React?****

#### **What is React?**

* **React** is an open-source JavaScript library developed by Facebook for building **user interfaces (UI)**. It is particularly designed for creating **dynamic and interactive UIs** for web applications. React enables developers to build large-scale applications where data changes frequently.
* It allows you to create **components**, which are reusable, self-contained building blocks that make up the user interface.

#### **Why Use React?**

* **Efficiency**: React uses a virtual DOM (Document Object Model) to optimize rendering, making it fast and responsive.
* **Component-Based**: React encourages the use of components, which means you can break down your UI into smaller, reusable pieces. This leads to cleaner, more maintainable code.
* **Declarative**: React makes it easy to describe how your UI should look at any given time, and it automatically updates when the data changes.
* **Popular and Well-Supported**: React has a large community, plenty of resources, and is widely adopted by developers.

#### **The React Ecosystem and the Power of Components**

* **React Ecosystem**: Along with React itself, developers use various tools and libraries that integrate with React, such as:
  + **React Router** for handling navigation and routes in single-page applications.
  + **Redux** for managing application state.
  + **React Developer Tools** for debugging React apps.
* **Components** are the heart of React applications. A component can be as simple as a button or as complex as an entire page. React lets you create **custom components** and reuse them across your application.
  + **Functional Components**: Components defined as functions.
  + **Class Components**: Components defined as JavaScript classes (older approach).

### ****Module 1.2: Setting up the Development Environment****

#### **Installing Node.js and npm**

To work with React, you first need **Node.js** (JavaScript runtime) and **npm** (Node Package Manager) installed on your machine.

1. **Install Node.js**:
   * Go to the [Node.js official website](https://nodejs.org) and download the installer for your operating system (LTS version recommended).
   * Run the installer and follow the prompts.
2. **Verify Installation**:
   * Open the terminal/command prompt and type the following commands to check if Node.js and npm are installed correctly:

node -v

npm -v

This will show the installed versions of Node.js and npm.

#### **Creating a New React Application Using** create-react-app

1. **Create React App**:
   * Use the create-react-app tool to quickly generate a new React project. In your terminal, run:

npx create-react-app my-app

* + Replace my-app with your desired project name.
  + This command installs all the dependencies and sets up the necessary project structure for React.

1. **Navigate into the project folder**:

cd my-app

1. **Run the App**:
   * Start the development server with:

npm start

* + This will open your React app in the browser at http://localhost:3000. You should see the default React app page.

#### **Overview of Project Structure**

* **public/**: Contains the static files like index.html, images, etc.
* **src/**: The source folder where the main React code lives (components, styles).
  + **App.js**: The main component of your app.
  + **index.js**: The entry point where the React app is rendered into the DOM.
* **package.json**: Manages dependencies and scripts for your project.
* **node\_modules/**: Contains installed npm packages.

### ****Module 1.3: JSX Syntax****

#### **Understanding JSX (JavaScript XML)**

* **JSX** is a syntax extension to JavaScript, which allows you to write HTML-like code within JavaScript. It looks very similar to HTML, but it’s actually a way to define **React elements**.
* JSX is **not** a requirement, but it makes writing React components much easier and more readable.

Example:

const element = <h1>Hello, world!</h1>;

#### **Embedding Expressions in JSX**

* In JSX, you can embed JavaScript expressions using curly braces {}. This allows you to dynamically generate content based on JavaScript values.

Example:

const name = "React";

const greeting = <h1>Hello, {name}!</h1>;

#### **Rules of JSX**

1. **className instead of class**:
   * In JSX, class is replaced by className because class is a reserved word in JavaScript.

Example:

<div className="container">Content</div>

1. **Self-Closing Tags**:
   * If an element has no children, it must be self-closed.

Example:

<img src="image.jpg" alt="React" />

1. **Wrapping Elements**:
   * JSX requires that all elements inside a component return a **single parent element**.

Example:

return (

<div>

<h1>Title</h1>

<p>Description</p>

</div>

);

### ****Module 1.4: Rendering React Elements****

#### **Rendering Elements to the DOM Using** ReactDOM.render()

* The ReactDOM.render() method is used to render a React element into the DOM. It takes two arguments:
  1. The React element to render.
  2. The DOM node where the React element should be placed.

Example:

ReactDOM.render(

<h1>Hello, React!</h1>,

document.getElementById('root')

);

#### **Creating and Rendering a Simple "Hello World" React App**

1. Open src/index.js and replace the default code with the following:

import React from 'react';

import ReactDOM from 'react-dom';

const App = () => {

return <h1>Hello, World!</h1>;

};

ReactDOM.render(

<App />,

document.getElementById('root')

);

1. Save the changes, and the page should now render "Hello, World!" when you visit http://localhost:3000.

### ****Summary****

* **React** is a powerful JavaScript library for building dynamic UIs using reusable components.
* **JSX** allows you to write HTML-like code within JavaScript, making React code easy to understand and work with.
* In React, we can render UI components to the browser using **ReactDOM.render()**.

### ****Week 2 : Components and Props in React****

In React, **Components** and **Props** are two fundamental concepts. Understanding them is essential for creating dynamic, reusable, and maintainable user interfaces. Let's break them down so you can start building React apps with components and props.

### ****Introduction to Components****

* **Components** are the building blocks of React applications. A React app is essentially a tree of components that represent the UI structure. Components can be as simple as a button or as complex as a whole page layout.
* Components in React are **reusable** and can be composed together to create more complex UIs. React promotes the use of **component-based architecture**, meaning each component has its own responsibilities and can be reused across the app.

### ****What are Components?****

A **Component** in React is a function or a class that optionally accepts inputs (called **props**) and returns a React element (UI).

* **React components** can be **functional** or **class-based**, with functional components being the modern and preferred way of building components using hooks.

### ****Functional Components vs. Class Components****

#### **Functional Components** (Recommended for Beginners)

* **Functional components** are simple JavaScript functions that return JSX (the UI of the component). They are easier to understand and maintain.
* Starting from React 16.8, functional components can use hooks like useState, useEffect, etc., allowing them to manage state and lifecycle methods just like class components.

Example of a Functional Component:

import React from 'react';

const Welcome = (props) => {

return <h1>Hello, {props.name}!</h1>;

};

export default Welcome;

#### **Class Components** (Legacy Approach)

* **Class components** are ES6 classes that extend from React.Component. They are more complex and were used before hooks were introduced in React.
* They can have lifecycle methods and manage state.

Example of a Class Component:

import React, { Component } from 'react';

class Welcome extends Component {

render() {

return <h1>Hello, {this.props.name}!</h1>;

}

}

export default Welcome;

#### **When to Use Functional or Class Components**

* **Functional components** are preferred because they are simpler and are now fully capable with hooks.
* **Class components** are still valid, but they are becoming less common in modern React development.

### ****How to Create and Use a Component****

#### **Creating a Simple Component**

A component can be a function that returns JSX. Here’s how to create and use a basic component:

1. **Create a new component** called Greeting.js:

javascript

Copy code

import React from 'react';

const Greeting = () => {

return <h2>Welcome to React!</h2>;

};

export default Greeting;

1. **Use the component in another file** (e.g., App.js):

javascript

Copy code

import React from 'react';

import Greeting from './Greeting';

const App = () => {

return (

<div>

<Greeting />

</div>

);

};

export default App;

* In the above example, the Greeting component is used inside the App component.

### ****Understanding Props****

* **Props (short for "properties")** are inputs that are passed into components from their **parent components**. Props allow you to pass dynamic data into components and make them reusable.

**Props** are **immutable** (read-only). A child component cannot modify the props it receives.

#### **What are Props?**

Props allow components to be dynamic and flexible by passing data from parent to child components. For example, you might want to pass a **name** to a Greeting component and have it display a personalized message.

Example of Props:

javascript

Copy code

const Greeting = (props) => {

return <h1>Hello, {props.name}!</h1>;

};

export default Greeting;

#### **Passing Data Between Components via Props**

To pass data to a child component, you include **attributes** in the child’s JSX tag (like name="John"), which will be accessible as props inside the child component.

Example of passing props:

javascript

Copy code

import React from 'react';

import Greeting from './Greeting';

const App = () => {

return <Greeting name="John" />;

};

export default App;

* In this example, the name="John" is passed as a prop to the Greeting component.
* Inside Greeting, you access this prop using props.name.

### ****Using Props for Dynamic Content****

* Props enable dynamic content in React. Instead of hardcoding values inside components, you can pass different props for each instance of a component.

Example with dynamic content:

javascript

Copy code

const Product = (props) => {

return (

<div>

<h2>{props.productName}</h2>

<p>{props.productDescription}</p>

</div>

);

};

export default Product;

Using this component in App:

javascript

Copy code

const App = () => {

return (

<div>

<Product productName="Laptop" productDescription="A high-performance laptop" />

<Product productName="Phone" productDescription="A new smartphone" />

</div>

);

};

* Here, we’re passing different data (product name and description) to the Product component to display different products.

### ****Nested Components and Composition****

* React promotes **component composition**, which means you can combine smaller components to create complex UIs. This promotes code reuse and makes the app easier to manage.

#### **Passing Data Between Parent and Child Components**

* A **parent component** can pass data to its **child components** using props.
* The child component can access these props, but it cannot change them.

Example:

javascript

Copy code

const Parent = () => {

const message = "Welcome to React!";

return <Child message={message} />;

};

const Child = (props) => {

return <h2>{props.message}</h2>;

};

* Here, the Parent component passes a message prop to the Child component, which displays the message.

#### **Component Composition**

* React promotes the composition of components. You can nest components inside each other and combine them to build larger components.

Example:

javascript

Copy code

const Header = () => <header><h1>My Website</h1></header>;

const Footer = () => <footer><p>&copy; 2024 My Website</p></footer>;

const App = () => {

return (

<div>

<Header />

<main><p>Welcome to my website!</p></main>

<Footer />

</div>

);

};

* In this example, App is composed of Header and Footer components. This structure is easy to extend and modify.

**Week 4: Handling Events and Conditional Rendering**

### ****Handling Events and Conditional Rendering in React****

In React, **event handling** and **conditional rendering** are essential concepts for building interactive and dynamic applications. Let's explore these topics so that you can write practical code in React.

### ****Module 1: Event Handling in React****

#### **Handling Events in React**

React provides a way to handle user interactions (such as clicks, key presses, form submissions) through **event handlers**. In React, events are written in **camelCase** (e.g., onClick, onChange), and the event handler is passed as a function.

1. **Basic Event Handling**:

In React, you can handle events like onClick, onChange, and more by attaching the event handler to an element.

Example (click event):

javascript

Copy code

import React from 'react';

const ClickButton = () => {

const handleClick = () => {

alert("Button clicked!");

};

return <button onClick={handleClick}>Click Me</button>;

};

export default ClickButton;

* In this example, the handleClick function is triggered when the button is clicked.

#### **Passing Arguments to Event Handlers**

Sometimes, you may need to pass arguments to event handlers. You can achieve this using **arrow functions** or **binding**.

Example (passing arguments):

javascript

Copy code

import React from 'react';

const Greeting = (props) => {

const handleClick = (name) => {

alert(`Hello, ${name}!`);

};

return <button onClick={() => handleClick(props.name)}>Click to Greet</button>;

};

export default Greeting;

Here, the handleClick function is called with the name prop passed from the parent component.

#### **Synthetic Events in React**

React uses a system called **Synthetic Events** to handle events. A Synthetic Event is a cross-browser wrapper around the native browser events. You don’t need to worry about browser compatibility issues when working with events in React.

For example, onClick, onChange, and other events are **synthetic events** in React, but they behave the same way as native events.

### ****Module 2: Conditional Rendering in React****

#### **Rendering Elements Based on Conditions**

In React, you can conditionally render components or elements by using JavaScript operators such as if, **ternary operators**, or **logical AND (&&)**.

1. **Using if Statements**: You can use if statements inside a component’s render method (or in the component function) to decide what to render.

Example (using if statements):

javascript

Copy code

import React from 'react';

const Message = ({ isLoggedIn }) => {

if (isLoggedIn) {

return <h1>Welcome back!</h1>;

} else {

return <h1>Please log in</h1>;

}

};

export default Message;

* Here, the component conditionally renders either "Welcome back!" or "Please log in" based on the isLoggedIn prop.

1. **Using Ternary Operator**: The ternary operator is a shorthand for if...else and is often used for conditional rendering in JSX.

Example (using ternary operator):

javascript

Copy code

import React from 'react';

const Message = ({ isLoggedIn }) => {

return (

<h1>{isLoggedIn ? "Welcome back!" : "Please log in"}</h1>

);

};

export default Message;

* In this example, if isLoggedIn is true, it shows "Welcome back!", otherwise it shows "Please log in".

1. **Using Logical AND (&&) Operator**: You can use the logical AND (&&) operator to render an element based on a condition. If the condition is true, the element will be rendered; otherwise, nothing is rendered.

Example (using &&):

javascript

Copy code

import React from 'react';

const Notification = ({ showNotification }) => {

return (

<div>

{showNotification && <p>You have a new message!</p>}

</div>

);

};

export default Notification;

* Here, the message is only displayed if showNotification is true.

#### **Conditional Rendering Based on Multiple Conditions**

You can combine different conditions for more complex rendering logic.

Example (multiple conditions):

javascript

Copy code

import React from 'react';

const Status = ({ status }) => {

if (status === 'loading') {

return <h1>Loading...</h1>;

} else if (status === 'error') {

return <h1>Error occurred</h1>;

} else {

return <h1>Data loaded</h1>;

}

};

export default Status;

* This component conditionally renders different messages based on the value of the status prop.

### ****Module 3: Handling Forms in React****

#### **Controlled vs. Uncontrolled Components**

* **Controlled components**: In a controlled component, form data is handled by the state of the component. The value of the form element is controlled by the React component.
* **Uncontrolled components**: In an uncontrolled component, form data is handled by the DOM itself. You can access the form data using refs.

**Controlled Component Example** (recommended approach):

javascript

Copy code

import React, { useState } from 'react';

const Form = () => {

const [inputValue, setInputValue] = useState('');

const handleChange = (event) => {

setInputValue(event.target.value);

};

const handleSubmit = (event) => {

alert('Form submitted with input: ' + inputValue);

event.preventDefault();

};

return (

<form onSubmit={handleSubmit}>

<label>

Name:

<input type="text" value={inputValue} onChange={handleChange} />

</label>

<button type="submit">Submit</button>

</form>

);

};

export default Form;

* In this example, the form is **controlled** because the input's value is tied to the component's state (inputValue).

**Uncontrolled Component Example**:

javascript

Copy code

import React, { useRef } from 'react';

const Form = () => {

const inputRef = useRef(null);

const handleSubmit = (event) => {

alert('Form submitted with input: ' + inputRef.current.value);

event.preventDefault();

};

return (

<form onSubmit={handleSubmit}>

<label>

Name:

<input type="text" ref={inputRef} />

</label>

<button type="submit">Submit</button>

</form>

);

};

export default Form;

* Here, the input's value is not controlled by React state but by the DOM through useRef.

#### **Handling Form Submissions and User Inputs**

Handling form submissions typically involves preventing the default form submission behavior and processing the form data.

Example (form submission):

javascript

Copy code

import React, { useState } from 'react';

const Form = () => {

const [email, setEmail] = useState('');

const handleEmailChange = (event) => {

setEmail(event.target.value);

};

const handleSubmit = (event) => {

alert(`Form submitted with email: ${email}`);

event.preventDefault();

};

return (

<form onSubmit={handleSubmit}>

<label>

Email:

<input type="email" value={email} onChange={handleEmailChange} />

</label>

<button type="submit">Submit</button>

</form>

);

};

export default Form;

* In this example, the form is controlled, and we handle the input change and form submission.

### ****Module 4: Managing Form State and Validation****

#### **Form State and Validation**

Managing form state is essential to control the user inputs and provide validation feedback. You can use conditional rendering to display error messages based on the form input.

Example (form validation):

javascript

Copy code

import React, { useState } from 'react';

const Form = () => {

const [email, setEmail] = useState('');

const [error, setError] = useState('');

const handleEmailChange = (event) => {

setEmail(event.target.value);

};

const handleSubmit = (event) => {

event.preventDefault();

if (!email.includes('@')) {

setError('Please enter a valid email address');

} else {

setError('');

alert(`Form submitted with email: ${email}`);

}

};

return (

<form onSubmit={handleSubmit}>

<label>

Email:

<input type="email" value={email} onChange={handleEmailChange} />

</label>

<button type="submit">Submit</button>

{error && <p style={{ color: 'red' }}>{error}</p>}

</form>

);

};

export default Form;

* In this example, we validate the email input and display an error message if the email is invalid.

**Week 3: State and Lifecycle Methods**

### ****Understanding State****

#### **What is State in React?**

* **State** in React is an object that holds information about the component that can change over time. It represents the **dynamic part** of a component and determines how it behaves or renders.
* When the state changes, React re-renders the component to reflect the updated data.

Example:

* Think of a **counter** app where a number increases or decreases based on user interaction. The number (or counter value) is **state** because it changes based on user actions.

#### **How to Set and Update State Using** useState

React provides the useState hook for managing state in **functional components**.

1. **Setting up state**: You initialize state by calling useState() and passing an initial value.
   * const [state, setState] = useState(initialValue);
2. **Updating state**: Use the setState function returned by useState to update the state.

Example:

javascript

Copy code

import React, { useState } from 'react';

const Counter = () => {

// Declare a state variable `count` with an initial value of 0

const [count, setCount] = useState(0);

return (

<div>

<p>Count: {count}</p>

<button onClick={() => setCount(count + 1)}>Increase</button>

</div>

);

};

export default Counter;

* In the example above, count is the state variable, and setCount is the function used to update the state. When the button is clicked, the state updates and the component re-renders.

#### **When to Use State**

State is used when:

* **Handling User Inputs**: When you need to capture and store the data entered by the user.
* **Dynamic Content**: When the content of your component changes based on events (like a button click, form submission, etc.).

Example:

javascript

Copy code

const InputForm = () => {

const [input, setInput] = useState('');

const handleInputChange = (e) => {

setInput(e.target.value);

};

return (

<div>

<input type="text" value={input} onChange={handleInputChange} />

<p>You typed: {input}</p>

</div>

);

};

Here, the input state variable stores the value entered in the text field.

### ****Module 2: Using Lifecycle Methods in Class Components****

#### **What Are Lifecycle Methods?**

Lifecycle methods are special methods in **class components** that allow you to run code at specific stages of a component’s life. These stages are:

1. **Mounting**: The component is being created and inserted into the DOM.
2. **Updating**: The component is being re-rendered due to changes in props or state.
3. **Unmounting**: The component is being removed from the DOM.

#### **Key Lifecycle Methods**

1. **componentDidMount()**:
   * This method is called once, immediately after a component is mounted (i.e., inserted into the DOM).
   * It's often used to **fetch data** from an API or to set up subscriptions.

Example:

javascript

Copy code

class MyComponent extends React.Component {

componentDidMount() {

console.log('Component Mounted!');

}

render() {

return <div>Hello!</div>;

}

}

1. **componentDidUpdate()**:
   * This method is called after the component has been updated (re-rendered) due to changes in props or state.
   * You can use this method to perform actions after the component has updated, like logging or making API calls.

Example:

javascript

Copy code

class MyComponent extends React.Component {

componentDidUpdate(prevProps, prevState) {

if (this.state.someValue !== prevState.someValue) {

console.log('State changed!');

}

}

render() {

return <div>{this.state.someValue}</div>;

}

}

1. **componentWillUnmount()**:
   * This method is called right before a component is unmounted (removed from the DOM).
   * You can use it for cleanup activities like **canceling subscriptions** or **clearing timers**.

Example:

javascript

Copy code

class MyComponent extends React.Component {

componentWillUnmount() {

console.log('Component will unmount');

}

render() {

return <div>Goodbye!</div>;

}

}

### ****Module 3: State Management with Functional Components and**** useState

#### **State Management with** useState

In **functional components**, we use the useState hook to manage state instead of lifecycle methods like in class components. useState allows you to add state to functional components.

* **useState(initialValue)** initializes the state with a specified value.
* The state can be updated by calling the function returned by useState.

#### **Setting Initial State and Updating State Dynamically**

You can set initial state and update it dynamically with the setter function returned from useState.

Example:

javascript

Copy code

import React, { useState } from 'react';

const Counter = () => {

const [count, setCount] = useState(0);

const increase = () => setCount(count + 1);

const decrease = () => setCount(count - 1);

return (

<div>

<p>Count: {count}</p>

<button onClick={increase}>Increase</button>

<button onClick={decrease}>Decrease</button>

</div>

);

};

export default Counter;

### ****Module 4: Introduction to**** useEffect

#### **What is** useEffect **and Why Is It Used?**

useEffect is a **hook** in React that allows you to **perform side effects** in your components. Side effects are operations like data fetching, subscriptions, and manual DOM manipulations.

* By default, useEffect runs after the first render and every time the component updates.
* You can also control when useEffect runs by passing a **dependency array**.

#### **Running Side Effects (API Calls, Subscriptions)**

You can use useEffect to perform side effects such as making **API calls**.

Example (API call using fetch):

javascript

Copy code

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

const DataFetcher = () => {

const [data, setData] = useState(null);

useEffect(() => {

fetch('https://api.example.com/data')

.then((response) => response.json())

.then((data) => setData(data));

}, []); // Empty array ensures this effect runs only once, after the initial render.

return <div>{data ? JSON.stringify(data) : 'Loading...'}</div>;

};

export default DataFetcher;

In this example, useEffect runs the API call once when the component mounts.

#### **Cleanup Functions in** useEffect

You can return a **cleanup function** inside useEffect to clean up resources (e.g., canceling subscriptions, clearing timers).

Example (clearing a timer):

javascript

Copy code

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

const Timer = () => {

const [seconds, setSeconds] = useState(0);

useEffect(() => {

const timer = setInterval(() => setSeconds((prev) => prev + 1), 1000);

// Cleanup function: Clear the timer when the component unmounts

return () => clearInterval(timer);

}, []); // Empty array to run this effect once after the initial render

return <div>Seconds: {seconds}</div>;

};

export default Timer;

Here, the timer is set up in useEffect, and the cleanup function clears the interval when the component is unmounted.

### ****Handling Events and Conditional Rendering in React for Freshers****

In React, **event handling** and **conditional rendering** are essential concepts for building interactive and dynamic applications. Let's explore these topics so that you can write practical code in React.

### ****Module 1: Event Handling in React****

#### **Handling Events in React**

React provides a way to handle user interactions (such as clicks, key presses, form submissions) through **event handlers**. In React, events are written in **camelCase** (e.g., onClick, onChange), and the event handler is passed as a function.

1. **Basic Event Handling**:

In React, you can handle events like onClick, onChange, and more by attaching the event handler to an element.

Example (click event):

javascript

Copy code

import React from 'react';

const ClickButton = () => {

const handleClick = () => {

alert("Button clicked!");

};

return <button onClick={handleClick}>Click Me</button>;

};

export default ClickButton;

* In this example, the handleClick function is triggered when the button is clicked.

#### **Passing Arguments to Event Handlers**

Sometimes, you may need to pass arguments to event handlers. You can achieve this using **arrow functions** or **binding**.

Example (passing arguments):

javascript

Copy code

import React from 'react';

const Greeting = (props) => {

const handleClick = (name) => {

alert(`Hello, ${name}!`);

};

return <button onClick={() => handleClick(props.name)}>Click to Greet</button>;

};

export default Greeting;

Here, the handleClick function is called with the name prop passed from the parent component.

#### **Synthetic Events in React**

React uses a system called **Synthetic Events** to handle events. A Synthetic Event is a cross-browser wrapper around the native browser events. You don’t need to worry about browser compatibility issues when working with events in React.

For example, onClick, onChange, and other events are **synthetic events** in React, but they behave the same way as native events.

### ****Module 2: Conditional Rendering in React****

#### **Rendering Elements Based on Conditions**

In React, you can conditionally render components or elements by using JavaScript operators such as if, **ternary operators**, or **logical AND (&&)**.

1. **Using if Statements**: You can use if statements inside a component’s render method (or in the component function) to decide what to render.

Example (using if statements):

javascript

Copy code

import React from 'react';

const Message = ({ isLoggedIn }) => {

if (isLoggedIn) {

return <h1>Welcome back!</h1>;

} else {

return <h1>Please log in</h1>;

}

};

export default Message;

* Here, the component conditionally renders either "Welcome back!" or "Please log in" based on the isLoggedIn prop.

1. **Using Ternary Operator**: The ternary operator is a shorthand for if...else and is often used for conditional rendering in JSX.

Example (using ternary operator):

javascript

Copy code

import React from 'react';

const Message = ({ isLoggedIn }) => {

return (

<h1>{isLoggedIn ? "Welcome back!" : "Please log in"}</h1>

);

};

export default Message;

* In this example, if isLoggedIn is true, it shows "Welcome back!", otherwise it shows "Please log in".

1. **Using Logical AND (&&) Operator**: You can use the logical AND (&&) operator to render an element based on a condition. If the condition is true, the element will be rendered; otherwise, nothing is rendered.

Example (using &&):

javascript

Copy code

import React from 'react';

const Notification = ({ showNotification }) => {

return (

<div>

{showNotification && <p>You have a new message!</p>}

</div>

);

};

export default Notification;

* Here, the message is only displayed if showNotification is true.

#### **Conditional Rendering Based on Multiple Conditions**

You can combine different conditions for more complex rendering logic.

Example (multiple conditions):

javascript

Copy code

import React from 'react';

const Status = ({ status }) => {

if (status === 'loading') {

return <h1>Loading...</h1>;

} else if (status === 'error') {

return <h1>Error occurred</h1>;

} else {

return <h1>Data loaded</h1>;

}

};

export default Status;

* This component conditionally renders different messages based on the value of the status prop.

### ****Module 3: Handling Forms in React****

#### **Controlled vs. Uncontrolled Components**

* **Controlled components**: In a controlled component, form data is handled by the state of the component. The value of the form element is controlled by the React component.
* **Uncontrolled components**: In an uncontrolled component, form data is handled by the DOM itself. You can access the form data using refs.

**Controlled Component Example** (recommended approach):

javascript

Copy code

import React, { useState } from 'react';

const Form = () => {

const [inputValue, setInputValue] = useState('');

const handleChange = (event) => {

setInputValue(event.target.value);

};

const handleSubmit = (event) => {

alert('Form submitted with input: ' + inputValue);

event.preventDefault();

};

return (

<form onSubmit={handleSubmit}>

<label>

Name:

<input type="text" value={inputValue} onChange={handleChange} />

</label>

<button type="submit">Submit</button>

</form>

);

};

export default Form;

* In this example, the form is **controlled** because the input's value is tied to the component's state (inputValue).

**Uncontrolled Component Example**:

javascript

Copy code

import React, { useRef } from 'react';

const Form = () => {

const inputRef = useRef(null);

const handleSubmit = (event) => {

alert('Form submitted with input: ' + inputRef.current.value);

event.preventDefault();

};

return (

<form onSubmit={handleSubmit}>

<label>

Name:

<input type="text" ref={inputRef} />

</label>

<button type="submit">Submit</button>

</form>

);

};

export default Form;

* Here, the input's value is not controlled by React state but by the DOM through useRef.

#### **Handling Form Submissions and User Inputs**

Handling form submissions typically involves preventing the default form submission behavior and processing the form data.

Example (form submission):

javascript

Copy code

import React, { useState } from 'react';

const Form = () => {

const [email, setEmail] = useState('');

const handleEmailChange = (event) => {

setEmail(event.target.value);

};

const handleSubmit = (event) => {

alert(`Form submitted with email: ${email}`);

event.preventDefault();

};

return (

<form onSubmit={handleSubmit}>

<label>

Email:

<input type="email" value={email} onChange={handleEmailChange} />

</label>

<button type="submit">Submit</button>

</form>

);

};

export default Form;

* In this example, the form is controlled, and we handle the input change and form submission.

### ****Module 4: Managing Form State and Validation****

#### **Form State and Validation**

Managing form state is essential to control the user inputs and provide validation feedback. You can use conditional rendering to display error messages based on the form input.

Example (form validation):

javascript

Copy code

import React, { useState } from 'react';

const Form = () => {

const [email, setEmail] = useState('');

const [error, setError] = useState('');

const handleEmailChange = (event) => {

setEmail(event.target.value);

};

const handleSubmit = (event) => {

event.preventDefault();

if (!email.includes('@')) {

setError('Please enter a valid email address');

} else {

setError('');

alert(`Form submitted with email: ${email}`);

}

};

return (

<form onSubmit={handleSubmit}>

<label>

Email:

<input type="email" value={email} onChange={handleEmailChange} />

</label>

<button type="submit">Submit</button>

{error && <p style={{ color: 'red' }}>{error}</p>}

</form>

);

};

export default Form;

* In this example, we validate the email input and display an error message if the email is invalid.

**Week 5: Lists, Keys, and Forms in React**

### ****Lists, Keys, and Forms in React for Freshers****

In React, **lists**, **keys**, and **forms** are essential concepts that help in rendering dynamic data, managing form inputs, and optimizing the rendering performance. Let’s break down each concept and explain how you can work with them practically.

### ****Module 1: Rendering Lists in React****

#### **Rendering Arrays with** map() **Function**

In React, you often need to render lists of items dynamically. You can do this by using JavaScript's built-in map() function, which allows you to iterate over an array and render a React element for each item.

1. **Basic List Rendering**: If you have an array of data, you can use map() to create a list of JSX elements.

Example (rendering a simple list of names):

javascript

Copy code

import React from 'react';

const NameList = () => {

const names = ['Alice', 'Bob', 'Charlie', 'David'];

return (

<ul>

{names.map((name, index) => (

<li key={index}>{name}</li> // Each item needs a unique "key"

))}

</ul>

);

};

export default NameList;

* **map() function**: It iterates over the names array and returns a <li> element for each item in the array.
* **key prop**: Each element in a list needs a unique key to help React identify which items have changed, are added, or are removed. This helps with optimizing rendering performance.

#### **Using Keys to Optimize Rendering Performance**

Keys help React keep track of elements in a list so it can efficiently update the UI when the data changes. If you don’t provide a key, React will have to re-render the entire list, which can be inefficient.

* **Avoid using array indices as keys in most cases**: Using indices as keys can cause issues when the list is reordered, added, or removed. Instead, use unique identifiers when possible.

Example (better use of keys with unique identifiers):

javascript

Copy code

import React from 'react';

const TaskList = () => {

const tasks = [

{ id: 1, task: 'Do the laundry' },

{ id: 2, task: 'Clean the house' },

{ id: 3, task: 'Buy groceries' }

];

return (

<ul>

{tasks.map((task) => (

<li key={task.id}>{task.task}</li> // Using unique "id" as key

))}

</ul>

);

};

export default TaskList;

* Here, the id field of each task is used as the key, which ensures better performance and avoids issues that might arise with using array indices.

### ****Module 2: Dynamic Forms and User Inputs****

#### **Building Dynamic Forms (Add/Remove Tasks or Items)**

One of the most common tasks in React is building **dynamic forms**, where the user can interact with the form to add or remove items (e.g., tasks, products). This often involves using **arrays** to store data in state and rendering lists based on the state.

Example (adding/removing tasks dynamically):

javascript

Copy code

import React, { useState } from 'react';

const TaskManager = () => {

const [tasks, setTasks] = useState([]);

const [newTask, setNewTask] = useState('');

// Add a new task to the list

const addTask = () => {

if (newTask.trim() !== '') {

setTasks([...tasks, { id: tasks.length + 1, task: newTask }]);

setNewTask(''); // Reset input field

}

};

// Remove a task by id

const removeTask = (taskId) => {

setTasks(tasks.filter((task) => task.id !== taskId));

};

return (

<div>

<input

type="text"

value={newTask}

onChange={(e) => setNewTask(e.target.value)}

placeholder="Enter a new task"

/>

<button onClick={addTask}>Add Task</button>

<ul>

{tasks.map((task) => (

<li key={task.id}>

{task.task}

<button onClick={() => removeTask(task.id)}>Remove</button>

</li>

))}

</ul>

</div>

);

};

export default TaskManager;

* **State for tasks**: We use useState to store the list of tasks and newTask to capture the new task input.
* **Adding tasks**: When the "Add Task" button is clicked, the new task is added to the tasks array in state using the setTasks function.
* **Removing tasks**: Each task has a "Remove" button that, when clicked, removes the task from the tasks array based on its id.

#### **Managing Form Data as Arrays in State**

React allows you to manage form data, including arrays of inputs, by storing the data in state and updating it when the user interacts with the form. You can store dynamic form fields in an array and add/remove items as needed.

Example (managing multiple inputs dynamically):

javascript

Copy code

import React, { useState } from 'react';

const DynamicForm = () => {

const [inputs, setInputs] = useState([{ id: 1, value: '' }]);

// Handle change in input field

const handleInputChange = (e, id) => {

setInputs(

inputs.map((input) =>

input.id === id ? { ...input, value: e.target.value } : input

)

);

};

// Add a new input field

const addInput = () => {

setInputs([...inputs, { id: inputs.length + 1, value: '' }]);

};

// Remove an input field

const removeInput = (id) => {

setInputs(inputs.filter((input) => input.id !== id));

};

return (

<div>

{inputs.map((input) => (

<div key={input.id}>

<input

type="text"

value={input.value}

onChange={(e) => handleInputChange(e, input.id)}

/>

<button onClick={() => removeInput(input.id)}>Remove</button>

</div>

))}

<button onClick={addInput}>Add Input</button>

</div>

);

};

export default DynamicForm;

* **State as an array**: The state inputs is an array that stores multiple input objects, each with a unique id and a value.
* **Adding inputs**: The addInput function adds a new input field to the inputs array.
* **Removing inputs**: The removeInput function removes a specific input field based on its id.

### ****Module 3: Handling Forms with Dynamic Input Fields****

#### **Managing Dynamic Form Data and State Updates**

You often need to manage dynamic input fields, where the number of fields can change based on user interaction. React makes it easy to handle these types of dynamic forms by updating the state whenever the form changes.

Example (dynamic form with multiple inputs):

javascript

Copy code

import React, { useState } from 'react';

const DynamicInputForm = () => {

const [fields, setFields] = useState([{ id: 1, value: '' }]);

const handleChange = (e, id) => {

setFields(fields.map((field) => (field.id === id ? { ...field, value: e.target.value } : field)));

};

const addField = () => {

setFields([...fields, { id: fields.length + 1, value: '' }]);

};

const removeField = (id) => {

setFields(fields.filter((field) => field.id !== id));

};

return (

<div>

<form>

{fields.map((field) => (

<div key={field.id}>

<input

type="text"

value={field.value}

onChange={(e) => handleChange(e, field.id)}

/>

<button type="button" onClick={() => removeField(field.id)}>

Remove

</button>

</div>

))}

<button type="button" onClick={addField}>

Add Field

</button>

</form>

</div>

);

};

export default DynamicInputForm;

* **Dynamic fields**: The form can grow or shrink dynamically as the user adds or removes input fields.
* **State updates**: The fields array in the state is updated whenever a field is added, removed, or its value changes.

**Week 6: Basics of Hooks (useState, useEffect) & Styling Components**

### ****Basics of Hooks (useState, useEffect) & Styling Components for Freshers****

In React, **Hooks** are functions that allow you to use state and other React features in functional components. Two of the most commonly used hooks are useState and useEffect. Additionally, styling components in React can be done in various ways, including basic CSS, CSS Modules, and Styled Components. Let’s break down these topics step by step, so you can understand and use them effectively in your React projects.

### ****Module 1: Using the**** useState ****Hook****

#### **Recap of** useState

useState is a Hook that allows you to add state to functional components. Before hooks, state was only available in class components, but with hooks, you can now manage state in functional components as well.

* useState returns an **array** with two elements:
  + The **current state** value
  + A function to **update** the state

**Basic Example of useState:**

javascript

Copy code

import React, { useState } from 'react';

const Counter = () => {

// Declare a state variable "count" with initial value 0

const [count, setCount] = useState(0);

return (

<div>

<h1>Count: {count}</h1>

<button onClick={() => setCount(count + 1)}>Increase</button>

</div>

);

};

export default Counter;

* **useState(0)** initializes the state variable count to 0.
* **setCount(count + 1)** updates the state value when the button is clicked.

#### **Complex State with Arrays and Objects**

You can use useState to manage more complex states, such as arrays and objects. When working with arrays or objects, it's important to update the state immutably, meaning you should never modify the state directly.

**Example with Arrays:**

javascript

Copy code

import React, { useState } from 'react';

const TodoList = () => {

const [todos, setTodos] = useState([]);

const addTodo = (todo) => {

setTodos([...todos, todo]); // Spread operator to add new todo

};

return (

<div>

<button onClick={() => addTodo('New Task')}>Add Task</button>

<ul>

{todos.map((todo, index) => (

<li key={index}>{todo}</li>

))}

</ul>

</div>

);

};

export default TodoList;

**Example with Objects:**

javascript

Copy code

import React, { useState } from 'react';

const UserProfile = () => {

const [user, setUser] = useState({ name: '', age: '' });

const updateUser = (key, value) => {

setUser((prevUser) => ({

...prevUser, // Spread operator to keep existing values

[key]: value, // Dynamically update the key

}));

};

return (

<div>

<input

type="text"

value={user.name}

onChange={(e) => updateUser('name', e.target.value)}

/>

<input

type="number"

value={user.age}

onChange={(e) => updateUser('age', e.target.value)}

/>

<h2>{user.name} - {user.age} years old</h2>

</div>

);

};

export default UserProfile;

* **Spreading objects/arrays** ensures you don’t mutate the original state, which allows React to properly re-render your components.

### ****Module 2: Using the**** useEffect ****Hook****

#### **Introduction to** useEffect

useEffect is another essential Hook in React. It allows you to perform side effects in your components, such as fetching data, updating the DOM, or setting up subscriptions. useEffect runs after the component renders, and you can control when it runs by passing a dependency array.

* The basic syntax of useEffect looks like this:

javascript

Copy code

useEffect(() => {

// Code to run after render

}, [dependencies]); // dependencies array (optional)

**Example (Basic useEffect):**

javascript

Copy code

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

const Timer = () => {

const [count, setCount] = useState(0);

// useEffect that runs every time the component renders

useEffect(() => {

const interval = setInterval(() => setCount(count + 1), 1000);

return () => clearInterval(interval); // Cleanup on unmount

}, [count]); // Runs whenever "count" changes

return <h1>{count}</h1>;

};

export default Timer;

* useEffect runs after each render, and the dependency array [count] tells React to only rerun the effect if count has changed.
* **Cleanup function**: You can return a cleanup function to clean up resources (e.g., clearing an interval or unsubscribing from a service) when the component unmounts or before the effect runs again.

#### **Fetching Data from an API and Displaying It**

useEffect is often used to fetch data when a component mounts. You can use the fetch API or libraries like axios to retrieve data and update the component state.

**Example (Fetching Data with useEffect):**

javascript

Copy code

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

const FetchData = () => {

const [data, setData] = useState(null);

const [loading, setLoading] = useState(true);

useEffect(() => {

fetch('https://jsonplaceholder.typicode.com/posts')

.then((response) => response.json())

.then((json) => {

setData(json);

setLoading(false);

})

.catch((error) => console.error('Error fetching data:', error));

}, []); // Empty dependency array means it runs only once, when the component mounts

if (loading) {

return <h2>Loading...</h2>;

}

return (

<div>

<h1>Posts</h1>

<ul>

{data.map((post) => (

<li key={post.id}>{post.title}</li>

))}

</ul>

</div>

);

};

export default FetchData;

* **Empty dependency array []**: Ensures that the effect runs only once after the component mounts, similar to componentDidMount in class components.

### ****Module 3: Styling Components in React****

Styling your React components is an important part of creating interactive and visually appealing UIs. React offers multiple ways to style components:

#### **Basic CSS Styling in React**

You can use regular CSS files to style React components. Import the CSS file into your React component and apply the classes to the JSX elements.

**Example (CSS in React):**

1. **styles.css**:

css

Copy code

button {

background-color: blue;

color: white;

padding: 10px;

}

1. **React Component (App.js)**:

javascript

Copy code

import React from 'react';

import './styles.css'; // Import CSS file

const Button = () => {

return <button>Click Me</button>;

};

export default Button;

#### **Introduction to CSS Modules for Scoped Styling**

CSS Modules allow you to write CSS in a modular way, where styles are scoped to the component that imports them. This helps avoid global styles clashing.

**Example (CSS Modules):**

1. **Button.module.css**:

css

Copy code

.button {

background-color: red;

color: white;

padding: 10px;

}

1. **Button Component (Button.js)**:

javascript

Copy code

import React from 'react';

import styles from './Button.module.css'; // Import the module

const Button = () => {

return <button className={styles.button}>Click Me</button>;

};

export default Button;

* **CSS Modules**: By importing Button.module.css as styles, the class names are locally scoped to the component. This prevents any class name conflicts.

#### **Introduction to Styled Components**

Styled Components is a library that allows you to write actual CSS inside your JavaScript code. You can define styled components and use them directly within your React JSX.

**Example (Styled Components):**

bash

Copy code

npm install styled-components

javascript

Copy code

import React from 'react';

import styled from 'styled-components';

// Create a styled button

const Button = styled.button`

background-color: green;

color: white;

padding: 10px;

`;

const App = () => {

return <Button>Click Me</Button>;

};

export default App;

* **Styled Components**: The styled.button creates a styled version of the button element. You can add CSS directly inside the backticks, making it easier to manage styles with JavaScript.

Here’s a step-by-step guide to creating a **Product Management App** using React, covering all the specified features.

**Step 1: Setup React Project**

1. **Create a new React project**:

bash

Copy code

npx create-react-app product-app

cd product-app

npm start

1. Clean up unnecessary files:
   * Delete contents of App.css and App.js.
   * Remove App.test.js, logo.svg, and other boilerplate files.

**Step 2: Plan Components**

The app will use the following components:

1. **ProductList**: Displays the list of products.
2. **ProductForm**: A form to add a new product.
3. **ProductItem**: Represents a single product.

**Step 3: Create the Main App Component**

App.js will manage the state and pass necessary props to child components.

jsx

Copy code

import React, { useState, useEffect } from "react";

import ProductList from "./components/ProductList";

import ProductForm from "./components/ProductForm";

function App() {

const [products, setProducts] = useState([]);

// Simulate fetching data from a mock API

useEffect(() => {

const mockData = [

{ id: 1, name: "Laptop", price: "$1000" },

{ id: 2, name: "Phone", price: "$500" },

];

setTimeout(() => setProducts(mockData), 1000);

}, []);

// Add a new product

const addProduct = (newProduct) => {

setProducts([...products, newProduct]);

};

// Delete a product

const deleteProduct = (id) => {

setProducts(products.filter((product) => product.id !== id));

};

return (

<div className="app">

<h1>Product Management App</h1>

<ProductForm addProduct={addProduct} />

<ProductList products={products} deleteProduct={deleteProduct} />

</div>

);

}

export default App;

**Step 4: Create the ProductForm Component**

Handles adding new products via form inputs.

**File: src/components/ProductForm.js**

jsx

Copy code

import React, { useState } from "react";

function ProductForm({ addProduct }) {

const [name, setName] = useState("");

const [price, setPrice] = useState("");

const handleSubmit = (e) => {

e.preventDefault();

if (name && price) {

const newProduct = {

id: Date.now(),

name,

price,

};

addProduct(newProduct);

setName("");

setPrice("");

}

};

return (

<form onSubmit={handleSubmit} className="product-form">

<input

type="text"

placeholder="Product Name"

value={name}

onChange={(e) => setName(e.target.value)}

/>

<input

type="text"

placeholder="Product Price"

value={price}

onChange={(e) => setPrice(e.target.value)}

/>

<button type="submit">Add Product</button>

</form>

);

}

export default ProductForm;

**Step 5: Create the ProductList Component**

Displays the list of products or a message when the list is empty.

**File: src/components/ProductList.js**

jsx

Copy code

import React from "react";

import ProductItem from "./ProductItem";

function ProductList({ products, deleteProduct }) {

if (products.length === 0) {

return <p>No products available. Please add some!</p>;

}

return (

<div className="product-list">

{products.map((product) => (

<ProductItem

key={product.id}

product={product}

deleteProduct={deleteProduct}

/>

))}

</div>

);

}

export default ProductList;

**Step 6: Create the ProductItem Component**

Handles displaying and deleting a single product.

**File: src/components/ProductItem.js**

jsx

Copy code

import React from "react";

function ProductItem({ product, deleteProduct }) {

return (

<div className="product-item">

<span>{product.name} - {product.price}</span>

<button onClick={() => deleteProduct(product.id)}>Delete</button>

</div>

);

}

export default ProductItem;

**Step 7: Add Styling**

Add basic styling to improve the user interface.

**File: src/App.css**

css

Copy code

body {

font-family: Arial, sans-serif;

margin: 20px;

background-color: #f9f9f9;

}

.app {

max-width: 600px;

margin: 0 auto;

background: #fff;

padding: 20px;

box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);

}

h1 {

text-align: center;

color: #333;

}

.product-form {

display: flex;

gap: 10px;

margin-bottom: 20px;

}

.product-form input {

flex: 1;

padding: 10px;

border: 1px solid #ccc;

border-radius: 4px;

}

.product-form button {

padding: 10px 20px;

border: none;

background: #28a745;

color: white;

border-radius: 4px;

cursor: pointer;

}

.product-form button:hover {

background: #218838;

}

.product-list {

margin-top: 20px;

}

.product-item {

display: flex;

justify-content: space-between;

padding: 10px;

border: 1px solid #ccc;

margin-bottom: 10px;

border-radius: 4px;

}

.product-item button {

background: #dc3545;

color: white;

border: none;

padding: 5px 10px;

border-radius: 4px;

cursor: pointer;

}

.product-item button:hover {

background: #c82333;

}

**Step 8: Final Testing**

1. Run the app:

bash

Copy code

npm start

1. Test the following:
   * Add new products via the form.
   * View the list of products.
   * Delete a product.
   * Ensure the "No products available" message appears when the list is empty.

**Project Summary**

This project demonstrates:

1. **Components**: The app is divided into reusable components.
2. **State Management**: useState manages the list of products.
3. **Event Handling**: Events handle adding and deleting products.
4. **Styling**: CSS enhances the visual design.
5. **Conditional Rendering**: A message is displayed when no products are available.
6. **Hooks**: useEffect simulates fetching data from a mock API.

### ****Final Project: Build a Recipe App****

In this final project, you will create a **Recipe App** that allows users to view, add, and delete recipes. The app will showcase several core React concepts you've learned so far, including **state management**, **event handling**, **conditional rendering**, **components**, and **hooks** (like useState and useEffect). This will help solidify your understanding of how to build dynamic and interactive web applications with React.

### ****Project Overview****

* **Objective**: Build a Recipe App where users can view a list of recipes, add new recipes, and delete existing ones.
  + Each recipe will contain:
    - **Title** (e.g., "Spaghetti Bolognese")
    - **Ingredients** (e.g., "Tomato, Beef, Spaghetti")
    - **Preparation Instructions** (e.g., "Cook pasta, brown the beef, combine and serve")
* **Key Features**:
  + Display a list of recipes.
  + Add new recipes through a form.
  + Delete recipes.
  + Use **React components**, **state**, **props**, **event handling**, **conditional rendering**, and **basic hooks** to manage and display dynamic data.
  + Use **CSS** or **Styled Components** for styling.

### ****Project Requirements and Implementation Steps****

#### **1. Components**

Break the application down into **separate components** to make it modular and reusable. Here are some possible components you’ll need to create:

* **RecipeList**: Displays a list of all recipes.
* **RecipeCard**: Displays details for each individual recipe (title, ingredients, instructions).
* **AddRecipeForm**: Allows users to add a new recipe to the list.
* **NoRecipesMessage**: Displays a message when there are no recipes.

Each of these components will be responsible for a specific part of the UI and will receive or send data via **props** and **state**.

**Example of RecipeCard component**:

const RecipeCard = ({ title, ingredients, instructions, onDelete }) => {

return (

<div className="recipe-card">

<h3>{title}</h3>

<p><strong>Ingredients:</strong> {ingredients}</p>

<p><strong>Instructions:</strong> {instructions}</p>

<button onClick={onDelete}>Delete Recipe</button>

</div>

);

};

#### **2. State Management with** useState

Use the useState hook to manage the state of your recipes within the app. The recipes should be stored as an array of objects. Each object will represent a recipe with properties like id, title, ingredients, and instructions.

**Example of managing recipes in state**:

import React, { useState } from 'react';

const RecipeApp = () => {

const [recipes, setRecipes] = useState([

{

id: 1,

title: 'Spaghetti Bolognese',

ingredients: 'Tomato, Beef, Spaghetti',

instructions: 'Cook pasta, brown beef, combine and serve.'

},

]);

const addRecipe = (newRecipe) => {

setRecipes([...recipes, newRecipe]);

};

const deleteRecipe = (id) => {

setRecipes(recipes.filter((recipe) => recipe.id !== id));

};

return (

<div>

<RecipeList recipes={recipes} onDelete={deleteRecipe} />

<AddRecipeForm onAdd={addRecipe} />

</div>

);

};

* **State for Recipes**: You initialize the state with a default recipe, and as the user adds or deletes recipes, you update this state.
* **Adding Recipes**: Use a form and the addRecipe function to add a new recipe to the list.
* **Deleting Recipes**: Pass a function (deleteRecipe) to each RecipeCard to delete the corresponding recipe when the button is clicked.

#### **3. Event Handling**

Handle the **add recipe** and **delete recipe** actions using event handlers. For example:

* **Add Recipe**: When the user submits a form, a new recipe is added to the recipes array.
* **Delete Recipe**: When the user clicks the delete button, the selected recipe is removed from the list.

**Example of handling form submission**:

const AddRecipeForm = ({ onAdd }) => {

const [title, setTitle] = useState('');

const [ingredients, setIngredients] = useState('');

const [instructions, setInstructions] = useState('');

const handleSubmit = (e) => {

e.preventDefault();

const newRecipe = {

id: Date.now(), // Unique ID

title,

ingredients,

instructions,

};

onAdd(newRecipe);

setTitle('');

setIngredients('');

setInstructions('');

};

return (

<form onSubmit={handleSubmit}>

<input

type="text"

placeholder="Recipe Title"

value={title}

onChange={(e) => setTitle(e.target.value)}

/>

<textarea

placeholder="Ingredients"

value={ingredients}

onChange={(e) => setIngredients(e.target.value)}

/>

<textarea

placeholder="Instructions"

value={instructions}

onChange={(e) => setInstructions(e.target.value)}

/>

<button type="submit">Add Recipe</button>

</form>

);

};

* **handleSubmit**: Collects form data, creates a new recipe object, and calls the onAdd function passed from the parent to update the recipe list.

#### **4. Conditional Rendering**

Conditional rendering is used to display content based on certain conditions. For example:

* Display a **message** when there are no recipes.
* Only display the **Recipe List** if there are recipes to show.

**Example of conditional rendering for no recipes**:

const RecipeList = ({ recipes, onDelete }) => {

if (recipes.length === 0) {

return <NoRecipesMessage />;

}

return (

<div>

{recipes.map((recipe) => (

<RecipeCard

key={recipe.id}

{...recipe}

onDelete={() => onDelete(recipe.id)}

/>

))}

</div>

);

};

* **NoRecipesMessage**: Displays a simple message like "No recipes yet, please add some!"

#### **5. Use** useEffect **for Mock API Simulation**

You can simulate fetching recipes from a mock API using the useEffect hook. This will give users a more realistic experience of how a real app fetches data.

**Example of useEffect fetching mock data**:

javascript

Copy code

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

const RecipeApp = () => {

const [recipes, setRecipes] = useState([]);

useEffect(() => {

// Simulate fetching data from an API

const fetchRecipes = () => {

const mockRecipes = [

{

id: 1,

title: 'Pancakes',

ingredients: 'Flour, Milk, Eggs',

instructions: 'Mix ingredients and cook on skillet.',

},

{

id: 2,

title: 'Salad',

ingredients: 'Lettuce, Tomatoes, Cucumber',

instructions: 'Toss ingredients together and serve.',

},

];

setRecipes(mockRecipes);

};

fetchRecipes();

}, []); // Empty dependency array ensures this runs only once when the component mounts

return <RecipeList recipes={recipes} />;

};

* **useEffect**: Fetches mock data when the app loads (simulating a request to a server), and the state is updated with the fetched recipes.

### ****6. Styling the Application****

You can use **CSS** or **Styled Components** to style the application.

**Basic Example with CSS**:

/\* App.css \*/

.recipe-card {

border: 1px solid #ccc;

padding: 10px;

margin: 10px;

}

button {

background-color: red;

color: white;

padding: 5px;

}

**Styled Components Example**:

import styled from 'styled-components';

const Button = styled.button`

background-color: red;

color: white;

padding: 5px;

`;

const RecipeCard = styled.div`

border: 1px solid #ccc;

padding: 10px;

margin: 10px;

`;

### ****Project: Food Order Delivery System****

#### **Objective**

Create a food order delivery application where users can browse food items, add items to their cart, place an order, and manage the order state. Redux Toolkit will handle the global state management.

#### **Features**

1. **Browse Food Items**: Display a list of available food items.
2. **Add to Cart**: Add food items to the cart with quantity adjustments.
3. **Cart Management**: View and edit cart items.
4. **Place Order**: Submit the order.
5. **Order Summary**: Display an order confirmation page with the order details.

#### **Implementation Steps**

### ****Step 1: Setting up the Project****

1. **Create a new React app**:

bash

Copy code

npx create-react-app food-delivery-system

cd food-delivery-system

npm install @reduxjs/toolkit react-redux

1. **Project Structure**:

css

Copy code

src/

├── components/

│ ├── FoodItem.js

│ ├── FoodList.js

│ ├── Cart.js

│ ├── OrderSummary.js

├── features/

│ ├── cart/

│ │ ├── cartSlice.js

├── App.js

├── index.js

### ****Step 2: Redux Slice for Cart Management****

* **cartSlice.js**: Define the slice for cart management.

javascript

Copy code

import { createSlice } from '@reduxjs/toolkit';

const cartSlice = createSlice({

name: 'cart',

initialState: {

items: [],

totalAmount: 0,

},

reducers: {

addItemToCart: (state, action) => {

const existingItem = state.items.find((item) => item.id === action.payload.id);

if (existingItem) {

existingItem.quantity += action.payload.quantity;

existingItem.totalPrice += action.payload.price \* action.payload.quantity;

} else {

state.items.push({

...action.payload,

totalPrice: action.payload.price \* action.payload.quantity,

});

}

state.totalAmount += action.payload.price \* action.payload.quantity;

},

removeItemFromCart: (state, action) => {

const itemIndex = state.items.findIndex((item) => item.id === action.payload);

if (itemIndex !== -1) {

const item = state.items[itemIndex];

state.totalAmount -= item.totalPrice;

state.items.splice(itemIndex, 1);

}

},

clearCart: (state) => {

state.items = [];

state.totalAmount = 0;

},

},

});

export const { addItemToCart, removeItemFromCart, clearCart } = cartSlice.actions;

export default cartSlice.reducer;

### ****Step 3: Store Configuration****

* **index.js**: Set up the Redux store.

javascript

Copy code

import React from 'react';

import ReactDOM from 'react-dom';

import { Provider } from 'react-redux';

import { configureStore } from '@reduxjs/toolkit';

import cartReducer from './features/cart/cartSlice';

import App from './App';

const store = configureStore({

reducer: {

cart: cartReducer,

},

});

ReactDOM.render(

<Provider store={store}>

<App />

</Provider>,

document.getElementById('root')

);

### ****Step 4: Create Components****

#### **1. FoodItem Component**

* Displays individual food items with an "Add to Cart" button.

javascript

Copy code

import React from 'react';

import { useDispatch } from 'react-redux';

import { addItemToCart } from '../features/cart/cartSlice';

const FoodItem = ({ id, name, price }) => {

const dispatch = useDispatch();

const handleAddToCart = () => {

dispatch(addItemToCart({ id, name, price, quantity: 1 }));

};

return (

<div style={{ border: '1px solid #ccc', padding: '10px', margin: '10px' }}>

<h3>{name}</h3>

<p>Price: ${price.toFixed(2)}</p>

<button onClick={handleAddToCart}>Add to Cart</button>

</div>

);

};

export default FoodItem;

#### **2. FoodList Component**

* Displays a list of food items.

javascript

Copy code

import React from 'react';

import FoodItem from './FoodItem';

const foodData = [

{ id: 1, name: 'Pizza', price: 12.99 },

{ id: 2, name: 'Burger', price: 9.99 },

{ id: 3, name: 'Pasta', price: 8.99 },

];

const FoodList = () => {

return (

<div>

<h2>Food Menu</h2>

<div>

{foodData.map((food) => (

<FoodItem key={food.id} {...food} />

))}

</div>

</div>

);

};

export default FoodList;

#### **3. Cart Component**

* Displays the cart items and total price.

javascript

Copy code

import React from 'react';

import { useSelector, useDispatch } from 'react-redux';

import { removeItemFromCart, clearCart } from '../features/cart/cartSlice';

const Cart = () => {

const cart = useSelector((state) => state.cart);

const dispatch = useDispatch();

return (

<div>

<h2>Shopping Cart</h2>

{cart.items.length === 0 && <p>Your cart is empty.</p>}

{cart.items.map((item) => (

<div key={item.id}>

<p>

{item.name} - ${item.totalPrice.toFixed(2)} ({item.quantity})

</p>

<button onClick={() => dispatch(removeItemFromCart(item.id))}>Remove</button>

</div>

))}

<h3>Total: ${cart.totalAmount.toFixed(2)}</h3>

<button onClick={() => dispatch(clearCart())}>Clear Cart</button>

</div>

);

};

export default Cart;

#### **4. OrderSummary Component**

* Displays the order confirmation.

javascript

Copy code

import React from 'react';

import { useSelector } from 'react-redux';

const OrderSummary = () => {

const cart = useSelector((state) => state.cart);

return (

<div>

<h2>Order Summary</h2>

{cart.items.map((item) => (

<p key={item.id}>

{item.name} - {item.quantity} x ${item.price.toFixed(2)}

</p>

))}

<h3>Total: ${cart.totalAmount.toFixed(2)}</h3>

</div>

);

};

export default OrderSummary;

#### **5. App Component**

* Combine all components into the main application.

javascript

Copy code

import React from 'react';

import FoodList from './components/FoodList';

import Cart from './components/Cart';

import OrderSummary from './components/OrderSummary';

const App = () => {

return (

<div>

<h1>Food Delivery System</h1>

<FoodList />

<Cart />

<OrderSummary />

</div>

);

};

export default App;

### ****Step 5: Styling****

Use basic CSS or libraries like TailwindCSS or Bootstrap for better visuals.

#### **Outcome**

* A complete food delivery application where users can browse a food menu, manage their cart, and view the order summary. Redux Toolkit manages global state efficiently, ensuring a scalable and maintainable architecture.