**Hooks:**

React Hooks are a new feature of React.js that makes it possible to use state and other React features without writing a class.

Hooks allow us to "hook" into React features such as state and lifecycle methods.

Hooks allow function components to have access to state and other React features. Because of this, class components are generally no longer needed. Hooks were introduced in React 16.8 as a way to write reusable and stateful logic without using class components.

**1.Use State:**

The useState hook in React is used to **add state functionality to functional components**. It allows you to declare and manage state variables within your component. Here's an easy example to illustrate the usage of useState:

import React, { useState } from 'react';

function Counter() {

*// Declare a state variable called 'count' and initialize it to 0*

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

const increment = () => {

*// Update the 'count' state by increasing it by 1*

setCount(count + 1);

};

return (

<div>

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

<button onClick={increment}>Increment</button>

</div>

);

}

export default Counter;

**2. UseEffect**

In React, the useEffect hook is a built-in function that allows you to perform side effects in functional components. Side effects refer to actions, such as **interacting with the browser (DOM), making HTTP requests, setting up timers, Data Fetching updates and clean ups.**

The useEffect hook is similar to lifecycle methods in class components, but it is designed specifically for functional components. Here's an easy example to demonstrate the usage of useEffect:

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

function Counter() {

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

const increment = () => {

*// Update the 'count' state by increasing it by 1*

setCount(count + 1);

};

useEffect(() => {

*// This effect runs after every render*

console.log(`Clicked ${count} times`);

}, [count]);

return (

<div>

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

<button onClick={increment}>Increment</button>

</div>

);

}

export default Counter;

Here's how the dependency array works:

1. **Empty Dependency Array ([]):**

useEffect(() => { // Effect code }, []);

**Behavior:** The effect runs once after the initial render.

**Use Case:** This is useful when you want the effect to mimic componentDidMount, running only after the component mounts.

1. **Non-Empty Dependency Array:**

useEffect(() => { // Effect code }, [dependency1, dependency2]);

**Behavior:** The effect runs after the initial render and whenever any value in the dependency array changes between renders.

**Use Case:** This is useful when you want the effect to run based on changes in specific variables or state values.

1. **No Dependency Array:**

useEffect(() => { // Effect code });

**Behavior:** The effect runs after every render.

**Use Case:** Be cautious when using this approach, as it can lead to the effect running frequently, potentially causing performance issues. It's often better to specify dependencies to control when the effect runs.

**Achieve Life cycle methods using useEffect:**

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

const MyComponent = () => {

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

  // Equivalent to componentDidMount

  useEffect(() => {

    console.log("Component did mount");

    // Perform actions after the component is mounted

    // Cleanup function (equivalent to componentWillUnmount)

    return () => {

      console.log("Component will unmount");

      // Additional cleanup logic goes here

    };

  }, []); // Empty dependency array means this effect runs once after the initial render

  // Equivalent to componentDidUpdate

  useEffect(() => {

    console.log("Component did update");

    // Additional logic to run when 'count' changes

    // Cleanup function for this effect (if needed)

    return () => {

      console.log("Cleanup for count effect");

      // Additional cleanup logic for this effect goes here

    };

  }, [count]); // Dependency array includes 'count', so it runs whenever 'count' changes

  // Render component

  return (

    <div>

      <p>Count: {count}</p>

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

    </div>

  );

};

export default MyComponent;

**3. UseContext:**

The **useContext** hook in React is used to consume values from a context created by the **createContext** API.

Context provides a way to pass data through the component tree without the need to pass props down manually at every level. It is especially useful for sharing data that is required by many components at different levels of the component tree.

It allows you to access and use context values within functional components without the need for prop drilling. Here's a simple example to demonstrate the usage of useContext:

import React, { createContext, useContext } from 'react';

*// Create a context*

const ThemeContext = createContext();

*// A component that provides the context value*

function App() {

const theme = 'dark';

return (

<ThemeContext.Provider value={theme}>

<ChildComponent />

</ThemeContext.Provider>

);

}

*// A child component that consumes the context value*

function ChildComponent() {

const theme = useContext(ThemeContext);

return <div>Current theme: {theme}</div>;

}

export default App;

**4. useRef:**

The useRef hook in React is used to **create a mutable reference** that persists across renders of a functional component. It is used to access or store references to DOM elements, manage previous values of a state, or to store any value that needs to be retained between renders without triggering a re-render.

In React, a ref is a way to access the properties or methods of a React element. **inputRef** (or any variable with a name ending in "Ref") in the context of React is typically a reference to a DOM element, which can be a part of the React component. Refs are used to interact with the DOM directly, especially when you need to get or modify the value of a DOM element.

import React, { useRef } from "react";

function Counter() {

const counterRef = useRef(0);

const increment = () => {

counterRef.current += 1;

console.log(counterRef.current);

};

return (

<div>

<p> Counter: {counterRef.current} </p>

<button onClick={increment}> Increment </button>

</div>

);

}

export default Counter;

**5. useReducer:**

**useReducer** is a React hook used for managing state in functional components. It is an **alternative to using useState** and provides a way to **handle more complex state logic** in the components. It is often used when the state transitions are more involved and need to be managed in a centralized manner.

The useReducer hook takes in a reducer function and an initial state, and returns the current state and a dispatch function to trigger state updates. The reducer function is responsible for specifying how the state should change in response to different actions.

import React, { useReducer } from "react";

const initialState = { count: 0 };

const reducer = (state, action) => {

switch (action.type) {

case "INCREMENT":

return { count: state.count + 1 };

case "DECREMENT":

return { count: state.count - 1 };

default:

return state;

}

};

const Counter = () => {

const [state, dispatch] = useReducer(reducer, initialState);

const increment = () => {

dispatch({ type: "INCREMENT" });

};

const decrement = () => {

dispatch({ type: "DECREMENT" });

};

return (

<div>

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

<button onClick={increment}>Increment</button>

<button onClick={decrement}>Decrement</button>

</div>

);

};

export default Counter;

**6 & 7: UseCallback and useMemo Hooks:**

The **useCallback** hook is used to memoize functions in React. When a function is wrapped with **useCallback**, it will only be recreated if one of its dependencies changes. This is particularly useful when passing functions down to child components to prevent unnecessary re-renders.

The **useMemo** hook is used to memoize the results of a computation. It's particularly useful when you have a costly calculation that you want to avoid re-running on every render. Instead, you can use **useMemo** to store the result of the computation and only recompute it if the dependencies have changed.

import React, { useState, useCallback, useMemo } from 'react';

function App() {

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

*// Increment function wrapped with useCallback*

const increment = useCallback(() => {

setCount(count + 1);

}, [count]);

*// Calculate square of 'count' using useMemo*

const square = useMemo(() => {

console.log('Calculating square...');

return count \* count;

}, [count]);

return (

<div>

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

<p>Square: {square}</p>

<button onClick={increment}>Increment Count</button>

</div>

);

}

export default App;

**example 2:**

import React, { useState, useCallback, useMemo } from 'react';

const ItemList = ({ items, onIncrement }) => {

const itemCount = useMemo(() => items.length, [items]);

const handleButtonClick = useCallback(() => {

onIncrement();

}, [onIncrement]);

return (

<div>

<p>Item Count: {itemCount}</p>

<button onClick={handleButtonClick}>Increment Count</button>

</div>

);

};

const App = () => {

const [items, setItems] = useState(['Item 1', 'Item 2', 'Item 3']);

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

const incrementCount = useCallback(() => {

setCount((prevCount) => prevCount + 1);

}, []);

const addItem = useCallback(() => {

setItems((prevItems) => [...prevItems, `Item ${prevItems.length + 1}`]);

}, []);

return (

<div>

<ItemList items={items} onIncrement={incrementCount} />

<button onClick={addItem}>Add Item</button>

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

</div>

);

};

export default App;

In this example:

1. **ItemList** is a component that receives a list of items (**items**) and a callback function (**onIncrement**). It uses **useMemo** to memoize the count of items, preventing unnecessary recalculations if the **items** array doesn't change.
2. **handleButtonClick** is a callback function created with **useCallback** that calls the **onIncrement** function. It is memoized so that it doesn't get recreated on each render, improving performance by avoiding unnecessary re-renders of child components.
3. The **App** component maintains the state for **items** and a global **count**. It provides a callback function (**incrementCount**) to the **ItemList** component for incrementing the global count. The **addItem** function is also memoized using **useCallback** to avoid unnecessary recreations.
4. When you click the "Increment Count" button in the **ItemList** component, it calls the **onIncrement** callback, which triggers the **incrementCount** function in the **App** component, updating the global count.
5. Clicking the "Add Item" button in the **App** component adds a new item to the list, triggering a re-render of the **ItemList** component, which efficiently calculates and displays the updated item count without unnecessary recalculations.

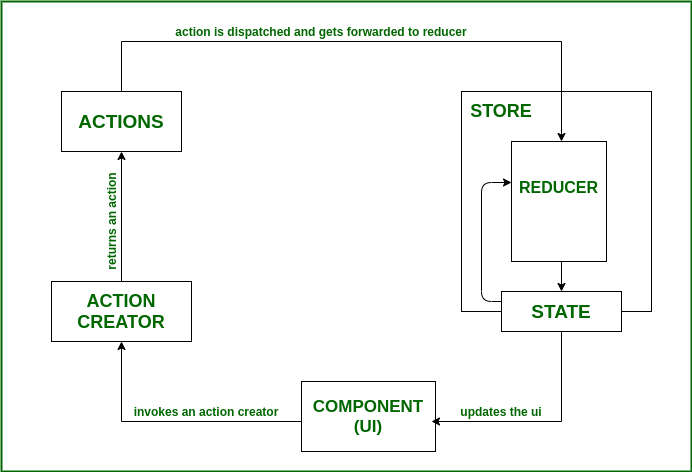
In summary, **useCallback** is used to memoize callback functions, and **useMemo** is used to memoize the result of expensive computations, both contributing to performance optimization in the React application.

**Redux:**

Redux is a state management library commonly used with React applications.

It helps manage the state of your application in a predictable and centralized way.

Imagine we have a counter component that increments and decrements a value. We'll use Redux to manage the state of the counter.



**1. Setting Up Redux:**

First, you'll need to install Redux and its related packages. In your project directory, run the following command:

*npm install redux react-redux*

**2. Creating Actions:**

In Redux, actions are plain JavaScript objects that describe an intention to change the state of your application. Actions provide **a way to communicate with the Redux store and trigger state updates**. They are dispatched to the store using the dispatch() function, which sends the action object to the reducers to handle the state modification.

Actions typically have a **type** property that indicates the **type of action being performed.** Apart from the type, actions can also contain additional data or payload that provides information about the action.

Actions are payloads of information that send data from your application to the Redux store. Create a file called counterActions.js and define two actions: increment and decrement.

Actions can also include payload data to provide additional information for the reducers.

*// counterActions.js*

export const increment = () => {

return {

type: "INCREMENT"

};

};

export const decrement = () => {

return {

type: "DECREMENT"

};

};

**3. Creating a Reducer:**

A reducer is a function that specifies how the application's state changes in response to actions.

React reducers are commonly used to manage the state of a component or a group of related components. They provide a predictable way to update and manage state based on actions dispatched within the component.

In React, a reducer is a function that determines changes to an application's state. It takes in an action and the current state as input and returns a new state based on that action. Reducers are typically used in conjunction with the **useReducer** hook to manage state in functional components.

*// counterReducer.js*

const counterReducer = (state = 0, action) => {

switch (action.type) {

case 'INCREMENT':

return state + 1;

case 'DECREMENT':

return state - 1;

default:

return state;

}

};

export default counterReducer;

**4. Creating a Redux Store:**

The store holds the state tree of your application. Create a file called store.js to configure the Redux store.

The store allows you to manage the application state globally, making it accessible from any component. This eliminates the need for **prop drilling** (passing props through intermediate components) and provides a more efficient and scalable way to share data between components.

*// store.js*

import { createStore } from 'redux';

import counterReducer from './counterReducer';

const store = createStore(counterReducer);

export default store;

**5. Creating the Counter Component:**

Create a file called Counter.js and define a React component that uses Redux to manage the counter state.

*// Counter.js*

import React from 'react';

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

import { increment, decrement } from './counterActions';

const Counter = () => {

const counter = useSelector((state) => state);

//useSelector is a hook.

const dispatch = useDispatch(); //useDispatch is a hook.

return (

<div>

<h2>Counter: {counter}</h2>

<button onClick={() => dispatch(increment())}>Increment</button>

<button onClick={() => dispatch(decrement())}>Decrement</button>

</div>

);

};

export default Counter;

**6. Integrating Redux in the App:**

Finally, integrate Redux in your main App component and render the Counter component.

**<Provider store={store}> ... </Provider>:**

This block of code wraps the Counter component inside the Provider component and passes the store object as a prop. The Provider component makes the Redux store available to all components in the application.

*// App.js*

import React from 'react';

import { Provider } from 'react-redux';

import store from './store';

import Counter from './Counter';

const App = () => {

return (

<Provider store={store}>

<Counter />

</Provider>

);

};

export default App;

**7. Render the App Component:**

In your main index.js file, render the App component to see the counter in action.

*// index.js*

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App';

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

**3 principles of Redux:**

1. Single Source of Truth[​](https://redux.js.org/tutorials/fundamentals/part-2-concepts-data-flow#single-source-of-truth)

The **global state** of your application is stored as an object inside a single **store**. Any given piece of data should only exist in one location, rather than being duplicated in many places.

This makes it easier to debug and inspect your app's state as things change, as well as centralizing logic that needs to interact with the entire application.

This does not mean that every piece of state in your app must go into the Redux store! You should decide whether a piece of state belongs in Redux or your UI components, based on where it's needed.

2. State is Read-Only[​](https://redux.js.org/tutorials/fundamentals/part-2-concepts-data-flow#state-is-read-only)

The only way to change the state is to dispatch an **action**, an object that describes what happened.

This way, the UI won't accidentally overwrite data, and it's easier to trace why a state update happened. Since actions are plain JS objects, they can be logged, serialized, stored, and later replayed for debugging or testing purposes.

3. Changes are Made with Pure Reducer Functions[​](https://redux.js.org/tutorials/fundamentals/part-2-concepts-data-flow#changes-are-made-with-pure-reducer-functions)

To specify how the state tree is updated based on actions, you write **reducer** functions. Reducers are pure functions that take the previous state and an action, and return the next state. Like any other functions, you can split reducers into smaller functions to help do the work, or write reusable reducers for common tasks.

**Reducer:**

React reducers are commonly used to manage the state of a component or a group of related components. They provide a predictable way to update and manage state based on actions dispatched within the component.

In React, a reducer is a function that determines changes to an application's state. It takes in an action and the current state as input and returns a new state based on that action. Reducers are typically used in conjunction with the useReducer hook to manage state in functional components.

**Prop Drilling:**

Prop drilling refers to the process of passing props (properties) from a parent component through one or more intermediate components down to a child component that needs access to the prop. It can occur when components need to share data or functionality that is not directly related to the parent-child relationship.

Here's an example to illustrate prop drilling:

*// Parent component*

function ParentComponent() {

const data = "Some data";

return (

<div>

<IntermediateComponent data={data} />

</div>

);

}

*// Intermediate component*

function IntermediateComponent({ data }) {

return (

<div>

<ChildComponent data={data} />

</div>

);

}

*// Child component*

function ChildComponent({ data }) {

return <div>{data}</div>;

}

In the example above, the ParentComponent has some data that needs to be accessed by the ChildComponent. However, since IntermediateComponent is in between, the data prop needs to be passed through it to reach ChildComponent. This is known as prop drilling.

Prop drilling can lead to several issues:

* This reduces component reusability and increases dependencies.
* When props change at the top level, all intermediate components in the prop drilling chain receive the updated props, potentially causing unnecessary re-renders and impacting performance.
* To mitigate the issues associated with prop drilling, various state management solutions like Redux, React Context, or custom hooks can be used. These solutions allow for a more centralized and accessible state without the need to pass props through every intermediate component. They provide a way to share data or functionality across components efficiently and decouple components from the specific prop requirements of their parents.

**State Lifting:**

State lifting in React refers to the practice of moving the state of a component higher up the component tree so that it can be shared and managed by multiple components. This is often used when multiple components need to share and synchronize the same piece of state.

Let's say you have a parent component that renders two child components. These child components need to share the same piece of data or state. Instead of managing this state separately within each child component, you can lift the state up to the parent component and pass it down to the children as props.

import React, { useState } from "react";

*// Child component that displays a counter*

function Counter(props) {

return <div>Counter: {props.value}</div>;

}

*// Parent component that manages the counter state and passes it to the child components*

function App() {

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

const increment = () => {

setCount (count + 1);

};

return (

<div>

<Counter value={ count } />

<button onClick={increment}>Increment</button>

</div>

);

}

export default App;

**Fetch Data From API: (Using Fetch Function)**

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

const EmployeeList = () => {

// State variable to store the list of employees

const [employees, setEmployees] = useState([]);

// Fetching data from the API using useEffect

useEffect(() => {

const fetchData = async () => {

try {

const response = await fetch(

"https://dummy.restapiexample.com/api/v1/employees"

);

const data = await response.json();

setEmployees(data.data);

} catch (error) {

console.log("Error fetching data:", error);

}

};

fetchData();

}, []);

return (

<div>

{employees.map((employee) => (

<div key={employee.id}>

<p>Id: {employee.id}</p>

<p>Name: {employee.employee\_name}</p>

<p>Salary: {employee.employee\_salary}</p>

<p>Age: {employee.employee\_age}</p>

</div>

))}

</div>

);

};

export default EmployeeList;

**Api Call using Axios:**

Axios is a popular JavaScript library used for making HTTP requests from the browser or Node.js. It provides an easy-to-use API for sending asynchronous HTTP requests and handling responses. You can use it to fetch data from APIs, send data to servers, and perform various HTTP operations.

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

import axios from "axios";

function MyComponent() {

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

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

  useEffect(() => {

    const fetchData = async () => {

      try {

        const response = await axios.get(

          "https://jsonplaceholder.typicode.com/users"

        );

        setData(response.data);

      } catch (error) {

        setError(error);

      }

    };

    fetchData();

  }, []);

  if (error) {

    return <div>Error: {error.message}</div>;

  }

  if (!data) {

    return <div>Loading...</div>;

  }

  return (

    <div>

      <h2>User List</h2>

      <ul>

        {data.map((user) => (

          <li key={user.id}>

            <strong>ID:</strong> {user.id}, <strong>Name:</strong> {user.name},{" "}

            <strong>Email:</strong> {user.email}

          </li>

        ))}

      </ul>

    </div>

  );

}

export default MyComponent;

**Output in Table:**

<table>

        <thead>

          <tr>

            <th>ID</th>

            <th>Name</th>

            <th>Email</th>

          </tr>

        </thead>

        <tbody>

          {data.map((user) => (

            <tr key={user.id}>

              <td>{user.id}</td>

              <td>{user.name}</td>

              <td>{user.email}</td>

            </tr>

          ))}

        </tbody>

      </table>

**Router:**

A Router is a library or component that helps you manage the navigation and routing of your application. It enables you to create different views or pages and switch between them based on the URL or user interactions.

import React from "react";

import { BrowserRouter as Router, Route, Routes, Link } from "react-router-dom";

function Home() {

return <h1>Home Page</h1>;

}

function About() {

return <h1>About Page</h1>;

}

function Contact() {

return <h1>Contact Page</h1>;

}

function App() {

return (

<Router>

<div>

<nav>

<ul>

<li>

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

</li>

<li>

<Link to="/about">About</Link>

</li>

<li>

<Link to="/contact">Contact</Link>

</li>

</ul>

</nav>

<Routes>

<Route path="/" exact component={Home} />

<Route path="/about" component={About} />

<Route path="/contact" component={Contact} />

</Routes>

</div>

</Router>

);

}

export default App;

**Display Text and Check Box:**

import React, { useState } from "react";

const Form = () => {

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

const [isChecked, setIsChecked] = useState(false);

const handleNameChange = (event) => {

setName(event.target.value);

};

const handleCheckboxChange = (event) => {

setIsChecked(event.target.checked);

};

const handleSubmit = (event) => {

event.preventDefault();

console.log("Name:", name);

console.log("Checkbox:", isChecked);

};

return (

<form onSubmit={handleSubmit}>

<div>

<label>

Name:

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

</label>

</div>

<div>

<label>

<input

type="checkbox"

checked={isChecked}

onChange={handleCheckboxChange}

/>

Checkbox

</label>

</div>

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

</form>

);

};

export default Form;

**Change Button Color:**

import React from "react";

import { useState } from "react";

function App() {

const [color, setColor] = useState("crimson");

function changecolor() {

setColor("blue");

}

return (

<div className="App">

<button style={{ background: color }} onClick={changecolor}>

Click here

</button>

</div>

);

}

export default App;

**React program that randomly changes the background color / 2 colors of the page when a button is clicked:**

import React, { useState } from "react";

function App() {

const [backgroundColor, setBackgroundColor] = useState("white"); *// Initial background color is white*

const changetoBlue = () => {

*// Change background color to blue*

setBackgroundColor("blue");

};

const changetoGreen = () => {

*// Change background color to green*

setBackgroundColor("green");

};

const changeRandomColor = () => {

*// Generate a random color*

const randomColor = "#" + Math.floor(Math.random() \* 16777215).toString(16);

setBackgroundColor(randomColor); *// Fix: Use setBackgroundColor to update state*

};

return (

<div className="App" style={{ backgroundColor, height: "200px" }}>

<button onClick={changetoBlue}>Change to Blue</button>

<button onClick={changetoGreen}>Change to Green</button>

<button onClick={changeRandomColor}>Change Background Color</button>

</div>

);

}

export default App;

**Generate a table, the cells clickable and changes the bg color to red while we click any cell.**

import React, { useState } from "react";

const DynamicTable = () => {

const [rows, setRows] = useState(3);

const [columns, setColumns] = useState(3);

const [lastClickedCell, setLastClickedCell] = useState(null);

const generateTable = () => {

const table = [];

let counter = 1;

for (let i = 0; i < rows; i++) {

const row = [];

for (let j = 0; j < columns; j++) {

row.push(counter++);

}

table.push(row);

}

return table;

};

const cellClickHandler = (cellValue) => {

setLastClickedCell(cellValue);

};

const isCellClicked = (cellValue) => {

return lastClickedCell === cellValue;

};

return (

<div>

<label htmlFor="rows">Rows: </label>

<input

type="number"

id="rows"

value={rows}

onChange={(e) => setRows(parseInt(e.target.value, 10))}

/>

<label htmlFor="columns">Columns: </label>

<input

type="number"

id="columns"

value={columns}

onChange={(e) => setColumns(parseInt(e.target.value, 10))}

/>

<button onClick={generateTable}>Generate Table</button>

<table style={{ borderCollapse: "collapse", margin: "20px" }}>

<tbody>

{generateTable().map((row, rowIndex) => (

<tr key={rowIndex}>

{row.map((cell, cellIndex) => (

<td

key={cellIndex}

style={{

border: "1px solid #ddd",

width: "50px",

height: "50px",

cursor: "pointer",

backgroundColor: isCellClicked(cell) ? "red" : "white"

}}

onClick={() => cellClickHandler(cell)}

>

{cell}

</td>

))}

</tr>

))}

</tbody>

</table>

</div>

);

};

export default DynamicTable;

**Payment options:**

import React, { useState } from "react";

const PaymentOptions = () => {

const [selectedOption, setSelectedOption] = useState(""); *// 'card' or 'upi'*

const [amount, setAmount] = useState("");

const [upiId, setUpiId] = useState("");

const handleOptionChange = (option) => {

setSelectedOption(option);

};

const handlePaymentSubmit = (e) => {

e.preventDefault();

*// Add validation logic*

if (

amount &&

((selectedOption === "card" && */\* validate card details \*/* true) ||

(selectedOption === "upi" && upiId))

) {

*// Add logic for handling the payment submission based on the selected option, amount, and UPI ID*

console.log(

`Payment submitted via ${selectedOption} for amount ${amount} with UPI ID ${upiId}:`

);

*// Display success alert*

window.alert("Payment successful!");

} else {

*// Display error alert if validation fails*

window.alert("Invalid input. Please check your details and try again.");

}

};

return (

<div>

<h2>Select Payment Option</h2>

<form onSubmit={handlePaymentSubmit}>

<label>

Enter Amount:

<input

type="number"

value={amount}

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

required

/>

</label>

<br />

<label>

<input

type="radio"

value="card"

checked={selectedOption === "card"}

onChange={() => handleOptionChange("card")}

/>

Credit Card

</label>

<label>

<input

type="radio"

value="upi"

checked={selectedOption === "upi"}

onChange={() => handleOptionChange("upi")}

/>

UPI

</label>

{selectedOption === "card" && (

<div>

<h3>Credit Card Details</h3>

{*/\* Include credit card form component here \*/*}

</div>

)}

{selectedOption === "upi" && (

<div>

<h3>UPI Details</h3>

<label>

Enter UPI ID:

<input

type="text"

value={upiId}

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

required

/>

</label>

</div>

)}

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

</form>

</div>

);

};

export default PaymentOptions;

**React Phone Book application:**

import React, { useState } from 'react';

import { createRoot } from 'react-dom/client';

const style = {

  table: {

    borderCollapse: 'collapse'

  },

  tableCell: {

    border: '1px solid gray',

    margin: 0,

    padding: '5px 10px',

    width: 'max-content',

    minWidth: '150px'

  },

  form: {

    container: {

      padding: '20px',

      border: '1px solid #F0F8FF',

      borderRadius: '15px',

      width: 'max-content',

      marginBottom: '40px'

    },

    inputs: {

      marginBottom: '5px'

    },

    submitBtn: {

      marginTop: '10px',

      padding: '10px 15px',

      border: 'none',

      backgroundColor: 'lightseagreen',

      fontSize: '14px',

      borderRadius: '5px',

      cursor: 'pointer'

    }

  }

}

function PhoneBookForm({ addEntryToPhoneBook }) {

  const [userFirstname, setUserFirstname] = useState('');

  const [userLastname, setUserLastname] = useState('');

  const [userPhone, setUserPhone] = useState('');

  const handleFormSubmit = (e) => {

    e.preventDefault();

    // Pass the entered information to the parent component

    addEntryToPhoneBook({

      userFirstname,

      userLastname,

      userPhone

    });

    // Clear the form after submitting

    setUserFirstname('');

    setUserLastname('');

    setUserPhone('');

  };

  return (

    <form onSubmit={handleFormSubmit} style={style.form.container}>

      <label>First name:</label>

      <br />

      <input

        style={style.form.inputs}

        className='userFirstname'

        name='userFirstname'

        type='text'

        value={userFirstname}

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

      />

      <br/>

      <label>Last name:</label>

      <br />

      <input

        style={style.form.inputs}

        className='userLastname'

        name='userLastname'

        type='text'

        value={userLastname}

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

      />

      <br />

      <label>Phone:</label>

      <br />

      <input

        style={style.form.inputs}

        className='userPhone'

        name='userPhone'

        type='text'

        value={userPhone}

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

      />

      <br/>

      <input

        style={style.form.submitBtn}

        className='submitButton'

        type='submit'

        value='Add User'

      />

    </form>

  );

}

function InformationTable({ entries }) {

  return (

    <table style={style.table} className='informationTable'>

      <thead>

        <tr>

          <th style={style.tableCell}>First name</th>

          <th style={style.tableCell}>Last name</th>

          <th style={style.tableCell}>Phone</th>

        </tr>

      </thead>

      <tbody>

        {entries.map((entry, index) => (

          <tr key={index}>

            <td style={style.tableCell}>{entry.userFirstname}</td>

            <td style={style.tableCell}>{entry.userLastname}</td>

            <td style={style.tableCell}>{entry.userPhone}</td>

          </tr>

        ))}

      </tbody>

    </table>

  );

}

function Application() {

  const [entries, setEntries] = useState([]);

  const addEntryToPhoneBook = (entry) => {

    // Update the state with the new entry

    setEntries([...entries, entry]);

  };

  return (

    <section>

      {/\* Pass the addEntryToPhoneBook function to the PhoneBookForm \*/}

      <PhoneBookForm addEntryToPhoneBook={addEntryToPhoneBook} />

      {/\* Pass the entries to the InformationTable \*/}

      <InformationTable entries={entries} />

    </section>

  );

}

const container = document.getElementById('root');

const root = createRoot(container);

root.render(<Application />);

**React Quiz App:**

import React, { useState } from 'react';

import { createRoot } from 'react-dom/client';

const style = {

  container: {

    padding: '20px',

    border: '1px solid #E0E0E0',

    borderRadius: '15px',

    width: 'max-content',

    marginBottom: '40px',

  },

  question: {

    fontWeight: 'bold',

    marginBottom: '10px',

  },

  options: {

    marginBottom: '5px',

  },

  button: {

    marginTop: '10px',

    padding: '10px 15px',

    border: 'none',

    backgroundColor: '#007BFF',

    color: '#FFF',

    fontSize: '14px',

    borderRadius: '5px',

    cursor: 'pointer',

  },

  feedback: {

    marginTop: '10px',

    fontSize: '14px',

  },

};

// Define the questions array before the component function

const questions = [

  {

    question: 'What is the capital of France?',

    options: ['London', 'Paris', 'Berlin', 'Madrid'],

    correct: 'Paris',

  },

  {

    question: 'What is the capital of Germany?',

    options: ['Berlin', 'Munich', 'Frankfurt', 'Hamburg'],

    correct: 'Berlin',

  },

];

function QuizApp() {

  const [currentQuestion, setCurrentQuestion] = useState(0);

  const [userAnswers, setUserAnswers] = useState(Array(questions.length).fill(''));

  const [feedback, setFeedback] = useState('');

  const handleOptionChange = (event) => {

    const selectedOption = event.target.value;

    const updatedUserAnswers = [...userAnswers];

    updatedUserAnswers[currentQuestion] = selectedOption;

    setUserAnswers(updatedUserAnswers);

  };

  const handleQuizSubmit = () => {

    const currentAnswer = userAnswers[currentQuestion];

    const correctAnswer = questions[currentQuestion].correct;

    setFeedback(currentAnswer === correctAnswer ? 'Correct!' : 'Incorrect!');

    setCurrentQuestion(currentQuestion + 1);

  };

  return (

    <div style={style.container}>

      {currentQuestion < questions.length ? (

        <div>

          <div id="question" style={style.question}>

            {questions[currentQuestion].question}

          </div>

          <div style={style.options}>

            {questions[currentQuestion].options.map((option, index) => (

              <div key={index} className="option">

                <input

                  type="radio"

                  id={`option${index + 1}`}

                  name="quizOptions"

                  value={option}

                  checked={userAnswers[currentQuestion] === option}

                  onChange={handleOptionChange}

                />

                <label htmlFor={`option${index + 1}`}>{option}</label>

              </div>

            ))}

          </div>

          <button style={style.button} id="submitBtn" onClick={handleQuizSubmit}>

            Submit

          </button>

          {feedback && <div id="feedback" style={style.feedback}>{feedback}</div>}

        </div>

      ) : (

        <div>

          <p id="quizComplete" style={style.feedback}>

            Quiz Complete! You have finished all questions.

          </p>

        </div>

      )}

    </div>

  );

}

const container = document.getElementById('root');

const root = createRoot(container);

root.render(<QuizApp />);