

1.2.3: Creating Props

Step 1. Import Necessary Libraries:

Ensure you're importing React and ReactDOM correctly to use JSX syntax and render components.

```
import React from 'react';
import ReactDOM from 'reactdom/client';
```

Step 2. Create a Functional Component that Accepts Props

Let's take an example of a Farm.js component, the functional component Farm accepts props as an argument. Props are passed down from the parent component and can be accessed via props object. In the example below, we shall use the properties animal, crops, and location.



Welcome

JS index.js

JS Farm.js

X

src > JS Farm.js > [o] Farm

```
1
2  const Farm = (props) =>{
3    return(<div>
4      <h2>Farm deals with:</h2>
5      Type of animals:{props.animal}<br/>
6      Crops:{props.crops}<br/>
7      Location:{props.location}
8
9    </div>)
10 }
11 export default Farm
```

Alternatively, you could destructure the props directly within the function signature to make it cleaner:



The screenshot shows a code editor interface with tabs for 'Welcome', 'index.js', and 'Farm.js'. The 'Farm.js' tab is active. The file structure is shown as 'src > Farm.js > Farm'. The code in 'Farm.js' is as follows:

```
1
2  const Farm = ({ animal, crops, location }) =>{
3      return(<div>
4          <h2>Farm deals with:</h2>
5          Type of animals:{animal}<br/>
6          Crops:{crops}<br/>
7          Location:{location}
8
9      </div>)
10 }
11 export default Farm
```

The code uses destructuring assignment to extract 'animal', 'crops', and 'location' from the props object passed to the component.

Step 3. Pass Props from the Parent Component:

Example: In index.js, you pass the props (animal, crops, location) from the parent component (Farm) when rendering it within ReactDOM.render.



Welcome

JS index.js



JS Farm.js



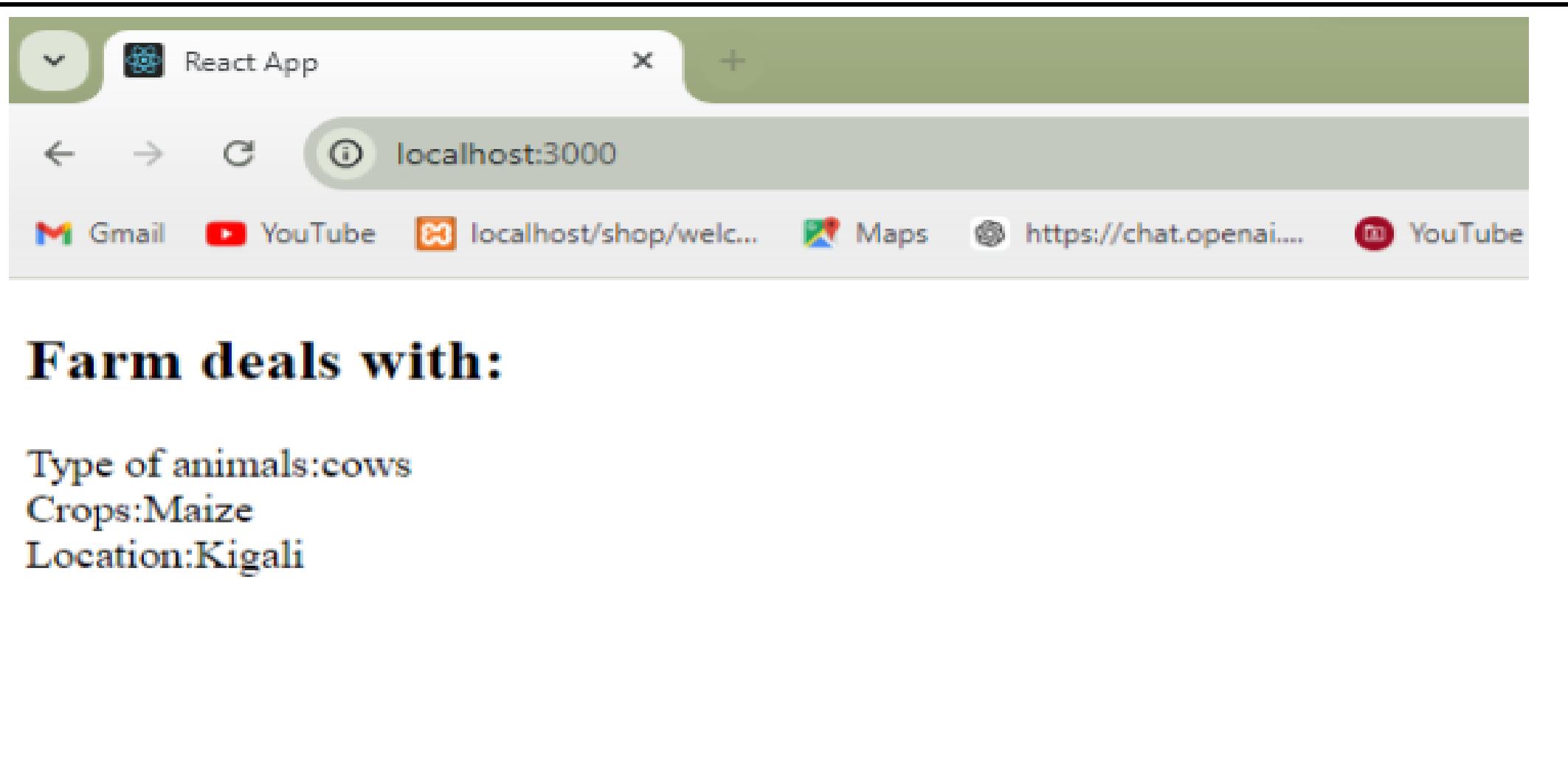
src > JS index.js > ...

```
● 1 ✘ import React from 'react';
  2 import ReactDOM from 'react-dom/client';
  3 import Farm from './Farm';
  4 const root = ReactDOM.createRoot(document.getElementById('root'));
  5 ✘ root.render(
  6   |   <Farm animal="cows" crops="Maize" location="Kigali"/>
  7 );
  8
  9
```

In this example, we are passing the values "cows", "Maize", and "Kigali" as props, which will be passed down to the Farm component and accessed inside it.

Step 4. Render the Component to the DOM:

The Farm component will receive the props and display them as per the JSX template inside Farm.js. When this code runs, it will output the following content:



1.2.4: Description of Lifecycle methods

componentDidMount:

This method is called once, immediately after the component is added (mounted) to the DOM. It's commonly used for:

- ✓ Fetching data from APIs.
- ✓ Setting up subscriptions (e.g., WebSockets)
- ✓ Initializing timers.

componentDidUpdate:

This method is called after the component updates due to changes in state or props. You can perform side effects here based on the previous props or state, such as making API requests if certain data has changed.

componentWillUnmount:

This method is called just before the component is removed (unmounted) from the DOM. It's typically used for cleanup tasks like:

- Clearing timers.
- Canceling API requests.
- Unsubscribing from services (e.g., WebSocket).

Note: In modern React, these lifecycle methods can be replaced using the `useEffect` hook in functional components.

1.3: Application Of UI Navigation

1.3.1: Applying Basic React navigation

Step 1: Install React Route

1. Open your command prompt or terminal in your project directory.
2. Run the command to install React Router as: `run npm install react-routerdom`

Step 2: Configure Routes

 1. First, ensure you have your React components, such as Home and About, created. These will be displayed based on the route.
 2. Configure your routes in the main App.js or another component responsible for routing.

Sample code

The screenshot shows a code editor interface with a yellow header bar containing the title "Sample code". The top navigation bar includes "File", "Edit", "Selection", "...", "route-app" (search bar), and various window control icons. On the left, there's an "EXPLORER" sidebar with a lock icon, showing a file tree for a project named "ROUTE-APP". The tree includes "src", "css", "includes", "HomeFiles" (with "Content.js", "Footer.js", "Header.js", "About.js", "Contact.js", "Home.js", "NavBar.js"), "App.js", "index.js", and ".gitignore". Below this is a "TIMELINE" and "NPM SCRIPTS" section. The main workspace shows a file named "App.js" with the following code:

```
src > App.js > App
1 import { Routes, Route } from "react-router-dom";
2 import NavBar from "./includes/NavBar";
3 import Home from './includes/Home'
4 import About from './includes/About'
5 import Contact from './includes/Contact'
6 export default function App(){
7   return(
8     <div>
9       <h1>Hello World</h1>
10      <NavBar />
11      <Routes>
12        <Route path="/" element={<Home />} />
13        <Route path="/About" element={<About />} />
14        <Route path="/Contact" element={<Contact />} />
15      </Routes>
16    </div>
17  )
18}
```

Step 3: Apply basic React Navigation

1. To navigate between different routes, you can create a simple navigation bar using the Link component from react-router-dom
2. Replace Link component with traditional anchor tag () in React Router, allowing seamless navigation between pages without reloading the entire application.

Sample code

The screenshot shows a code editor interface with the following details:

- EXPLORER:** Shows the project structure under "IRROUTE-APP". The files listed are: src, CSS, includes, HomeFiles, Content.js, Footer.js, Header.js, About.js, Contact.js, Home.js, and NavBar.js. The file "NavBar.js" is currently selected.
- FILES:** Shows the open files: NavBar.js, App.js, and style.css. The NavBar.js file is the active tab.
- Content of NavBar.js:**

```
src > includes > JS NavBar.js > NavBar
1 import { Link } from "react-router-dom"
2
3 function NavBar(){
4   return(
5     <nav>
6       <Link to="/"><label>Home</label></Link>
7       <Link to="/About"><label>About Us</label></Link>
8       <Link to="/Contact"><label>Contact Us</label></Link>
9     </nav>
10  )
11
12 }
13 export default NavBar
```

1.3.2: Handling 404 Pages, Redirects and URL Parameter

A 404 page is displayed when no routes match the requested URL. You can achieve this by using the `<Route>` component without a path and placing it at the end of your `<Routes>` block.

Steps to Handle 404 Pages:

Step 1. Ensure react-router-dom is installed.

```
npm install react-router-dom
```

Step 2. Set up the routing in your component as shown below

The screenshot shows a code editor interface with the following details:

- File Bar:** File, Edit, Selection, View, ...
- Search Bar:** route-app
- Toolbars:** Standard file operations (New, Open, Save, etc.)
- Left Sidebar:** Explorer view showing the project structure:
 - ROUTE-... (highlighted)
 - src
 - includes
 - HomeFiles
 - Content.js
 - Footer.js
 - Header.js
 - About.js
 - Contact.js
 - Home.js
 - NavBar.js
 - NotFound.js
 - App.js (highlighted)
 - index.js
 - .gitignore
 - package-lock.json
 - package.json
 - README.md
- Bottom Navigation:** TIMELINE and NPM SCRIPTS
- Code Editor:** The App.js file is open, showing the following code:

```
src > JS App.js > ...
1 import { Routes, Route } from "react-router-dom";
2 import NavBar from "./includes/NavBar";
3 import Home from './includes/Home'
4 import About from './includes/About'
5 import Contact from './includes/Contact'
6 import NotFound from "./includes/NotFound";
7 export default function App(){
8     return(
9         <div>
10            <h1>Hello World</h1>
11            <NavBar />
12            <Routes>
13                <Route path="/" element={<Home />} />
14                <Route path="/About" element={<About />} />
15                <Route path="/Contact" element={<Contact />} />
16                <Route element={<NotFound />} />
17            </Routes>
18        </div>
19    )
20 }
```

Redirects

You can use the <Redirect> component to automatically redirect users from an old path to a new one.

Steps to use Redirects:

In your routing component, add a route that uses <Redirect> to point from the old path to the new path.

```
import { Switch, Route, Redirect } from 'react-router-dom';
import Home from './Home';
import NewComponent from './NewComponent'; // Component for the new path
const App = () => (
<Switch>
<Route path="/" exact component={Home} />
{/ Redirect from /old-path to /new-path /}
<Route path="/old-path">
<Redirect to="/new-path" />
</Route>
<Route path="/new-path" component={NewComponent} />
<Route component={NotFound} />
</Switch> );
export default App;
```

Steps to use URL Parameters

React Router allows you to capture URL parameters and use them inside components. This is useful for dynamic pages, such as a user profile.

Step 1: Define a route with a parameter in your App component.

Step 2: Access the parameter via `match.params` inside the target component.

```
import { Switch, Route } from 'react-router-dom';
import Home from './Home';
import User from './User'; // Component to handle the dynamic user id
const App = () => (
<Switch>
<Route path="/" exact component={Home} />
{/ Define a route with a URL parameter /}
<Route path="/user/:id" component={User} />
<Route component={NotFound} />
</Switch>
);
export default App;
```

Step 3: In the User component, extract the id parameter from match.params:

```
import React from 'react';
const User = ({ match }) => {
  return <h1>User ID: {match.params.id}</h1>;
};
export default User;
```

1.3.3: Applying Nested Routes

To add nested routes, you'll structure your routes inside a parent component. The parent route will contain its own route but also define child routes.

Step 1: Define Parent Component

```
import { Outlet, Link } from 'react-router-dom';
function Dashboard() {
return (
<div>
<h1>Dashboard</h1>
<nav>
<ul>
<li><Link to="overview">Overview</Link></li>
<li><Link to="stats">Stats</Link></li>
</ul>
</nav>
{/ Outlet will render child routes here /}
<Outlet />
</div>
);
} export default Dashboard;
```

Step 2: Define Child Components

For the nested routes, you'll create child components:

```
function Overview() {  
  return <h2>Overview Page</h2>;  
}  
  
function Stats() {  
  return <h2>Stats Page</h2>;  
}
```

Step 3: Add Nested Routes in App.js

Now, add the nested routes under the parent route in your Routes.

```
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';  
import Home from './components/Home';  
import About from './components/About';  
import Dashboard from './components/Dashboard';  
import Overview from './components/Overview';
```

```
import Stats from './components/Stats';
function App() {
return (
<Router>
<Routes>
<Route path="/" element={<Home/>} />
<Route path="/about" element={<About/>} />
{/ Parent Route with Nested Routes /}
<Route path="/dashboard" element={<Dashboard />}>
<Route path="overview" element={<Overview/>} />
<Route path="stats" element={<Stats/>} />
</Route>
</Routes>
</Router>
); } export default App;
```

Step 4: Test the routes

Application of learning 1.3.

ABC is an online business company Located in Kayonza district in Eastern Province-Rwanda. It needs a ReactJS developer to develop an ecommerce website. Assume you have been hired as a ReactJS developer for the company, you are tasked with creating an ecommerce application using React. Begin by installing React Router to manage UI navigation effectively. Configure various routes for product listings, user profiles, and shopping carts. Implement basic navigation to ensure users can easily access different sections. Handle 404 pages for non-existent routes and set up redirects for outdated URLs. Additionally, utilize URL parameters for filtering products and apply nested routing for displaying reviews within individual product pages.

1.4: Application Of React Hooks

Definition of Key Concepts

- ✓ **Hooks:** Hooks are special functions in React that let you use state and other React features in functional components without needing a class.
- ✓ **State Hooks (useState):** State hooks allow functional components to store and manage state.
- ✓ **Effect Hooks (useEffect):** Effect hooks let you perform side effects in functional components. These effects can be data fetching, subscriptions, or manually changing the DOM.
- ✓ **Context Hooks (useContext):** Context hooks provide a way to pass data through the component tree without needing to pass props manually at every level.
- ✓ **Ref Hooks (useRef):** Ref hooks allow you to persist values between renders without causing a re-render when the value changes.

- ✓ **Callback Hooks (useCallback):** Callback hooks memorize a function so that the function does not get recreated on every render. This is particularly useful for optimizing performance when passing functions as props to child components.

Types of Hooks

useState: Used for state management.

useEffect: Used for performing side effects in components.

useContext: Used for consuming context values.

useReducer: An alternative to useState for more complex state logic, similar to Redux's reducer.

useCallback: Memorizes functions to prevent unnecessary re-creations on re-renders.

useMemo: Memorizes values to prevent expensive calculations on each render.

useLayoutEffect: Similar to useEffect, but it runs synchronously after all DOM mutations, which can affect layout.

useImperativeHandle: Customizes the instance value exposed to parent components when using ref.

WHAT IS STATE IN REACT?

While props allow data to flow from parent to child, state is used to manage data that is local to a component. State is mutable, meaning it can change over time, and these changes trigger a re-render of the component. State is primarily used to track dynamic data that can be modified by the component itself or by user interaction.

Each component in React can have its own internal state, and this state can be updated using the **useState** hook in functional components or **this.setState()** in class components. **When a component's state changes, React re-renders the component with the updated state values.**

Example of Using State in React:

Let's create a counter component that uses state to track the number of times a button has been clicked.

PROPS

In React, props (short for "properties") are a way to pass data from a parent component to a child component. They are read-only and help make components reusable and dynamic.

```
// Method 1: Using the props parameter
function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}
```

```
// Method 2: Destructuring props
function Greeting({ name, age }) {
  return <h1>Hello, {name}! You are {age} years
old.</h1>;
}
```

```
import React, { useState } from 'react';
const Counter = () => {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>Current Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
export default Counter;
```

jsx

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

function Counter() {
  // Declare a state variable "count" with initial value 0
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Multiple State Variables

jsx

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

function UserForm() {
  // Multiple state variables
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");
  const [age, setAge] = useState(0);

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log({ name, email, age });
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={name}
        onChange={(e) => setName(e.target.value)}
        placeholder="Name"
      />
    
```

```
<input
  type="email"
  value={email}
  onChange={(e) => setEmail(e.target.value)}
  placeholder="Email"
/>
<input
  type="number"
  value={age}
  onChange={(e) => setAge(Number(e.target.value))}
  placeholder="Age"
/>
<button type="submit">Submit</button>
</form>
);
}
```

Object State

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

function Profile() {
  // State as an object
  const [user, setUser] = useState({
    name: 'John',
    age: 30,
    email: 'john@example.com'
  });

  // Update specific property in object state
  const updateName = () => {
    setUser(prevUser => ({
      ...prevUser, // Spread operator to keep other properties
      name: 'Jane'
    }));
  };
}
```

```
return (
  <div>
    <h2>{user.name}</h2>
    <p>Age: {user.age}</p>
    <p>Email: {user.email}</p>
    <button onClick={updateName}>Change
      Name</button>
  </div>
);
```

Common Use Cases for State:

- 1. Form Inputs:** State is commonly used to manage form input values and track changes made by the user.
- 2. UI Changes:** State is often used to track changes in the UI, such as showing or hiding elements, toggling menus.
- 3. Tracking Data:** State can be used to track data that changes over time, such as the number of items in a cart.

Key Differences Between Props and State

While props and state are both used to manage data in React, they serve different purposes and function in distinct ways.

Props:

- ❖ Passed from parent to child.
- ❖ Immutable (cannot be changed by the receiving component).
- ❖ Used for static or dynamic data that is passed down the component tree.
- ❖ Can be functions, arrays, objects, or any other data type.

State:

- ❖ Local to the component.
- ❖ Mutable (can be updated within the component).
- ❖ Used for dynamic data that is managed within the component itself.
- ❖ Changing the state triggers a re-render of the component.

Factors to Consider While Selecting the Right Hook

- ❖ **State complexity:** For simple state, useState is sufficient, but for complex state logic or state that depends on previous state, useReducer might be better.
- ❖ **Side effects:** If the component needs to perform a side effect like fetching data or interacting with external APIs, useEffect is appropriate.
- ❖ **Optimization:** When passing functions or values down as props, useCallback and useMemo help in optimizing performance by preventing unnecessary re-creations.
- ❖ **Context needs:** If the component requires access to globally available data without prop drilling, useContext is the best choice.
- ❖ **DOM access:** To directly interact with or store references to DOM elements or persist mutable values across renders, useRef is ideal.

Reasons for combining Hooks

- ❖ **Modularity:** Hooks encapsulate specific logic, allowing developers to compose different behaviours without code repetition.
- ❖ **Optimization:** Combining `useCallback` or `useMemo` with other hooks can optimize rendering performance by preventing unnecessary re-renders or recalculations.
- ❖ **Code organization:** Combining multiple hooks can help organize state, side effects, and event handling logic in a way that's easier to maintain and understand.
- ❖ **Reusability:** Custom hooks can be created by combining multiple built-in hooks. This enables reusing common logic across different components.

1. 4 .2: Implementation of Hooks

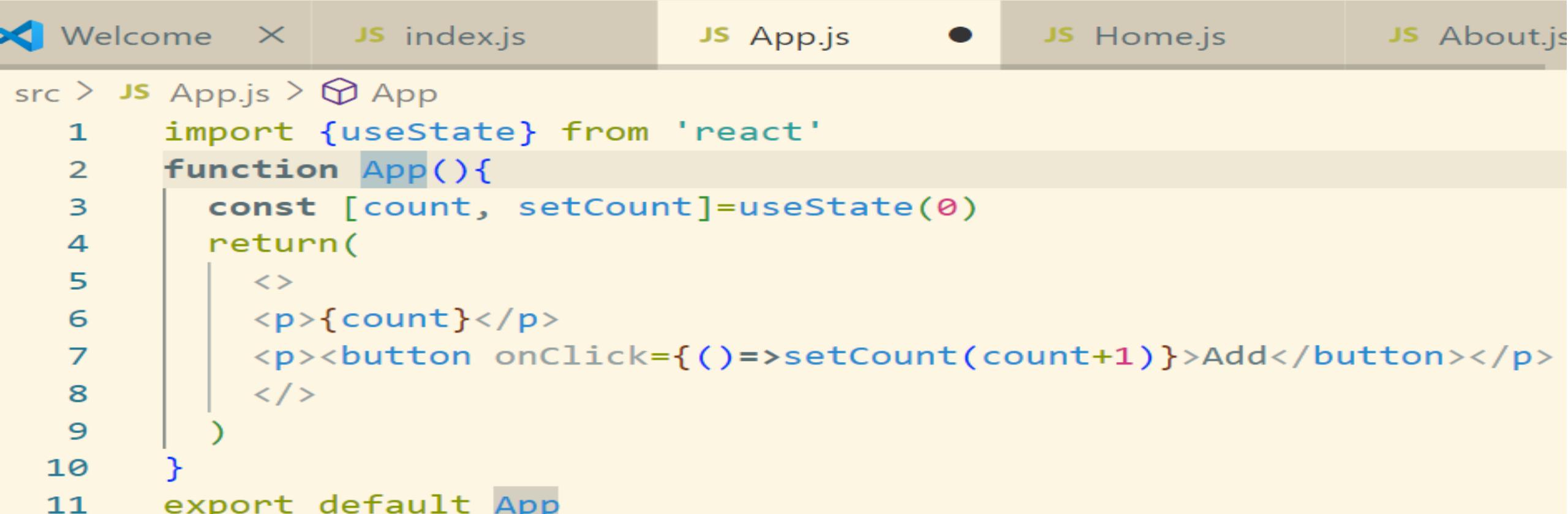
Steps of Implementing Hooks

Step 1: Import the useState hook from React.

Step 2: Use useState to define a state variable and a function to update it.

Step 3: Use the state variable in your component's render logic.

Step 4: Call the state updater function to modify the state based on an event.



```
src > JS App.js > App
1 import {useState} from 'react'
2 function App(){
3   const [count, setCount]=useState(0)
4   return(
5     <>
6     <p>{count}</p>
7     <p><button onClick={()=>setCount(count+1)}>Add</button></p>
8   )
9 }
10
11 export default App
```

Steps to implement Effect Hook

Step 1: Import the `useEffect` hook from React.

Step 2: Define the effect logic inside the `useEffect` callback.

Step 3: Optionally return a cleanup function to run when the component unmounts.

Step 4: Add dependencies in the second argument to control when the effect runs.

Steps to implement Ref Hook

Step 1: Import the `useRef` hook from React.

Step 2: Create a `ref` object using `useRef`.

Step 3: Attach the `ref` to a DOM element using the `ref` attribute.

src > JS App.js > [②] App

```
1 import { useEffect } from 'react';
2 const App = () => {
3   useEffect(() => {
4     const timerID = setInterval(() => console.log('Tick'), 1000);
5
6     return () => clearInterval(timerID); // Cleanup on unmount
7   }, []); // Empty dependency array ensures it runs only on mount
8
9   return <h1>Timer</h1>;
10 }
11 export default App
```

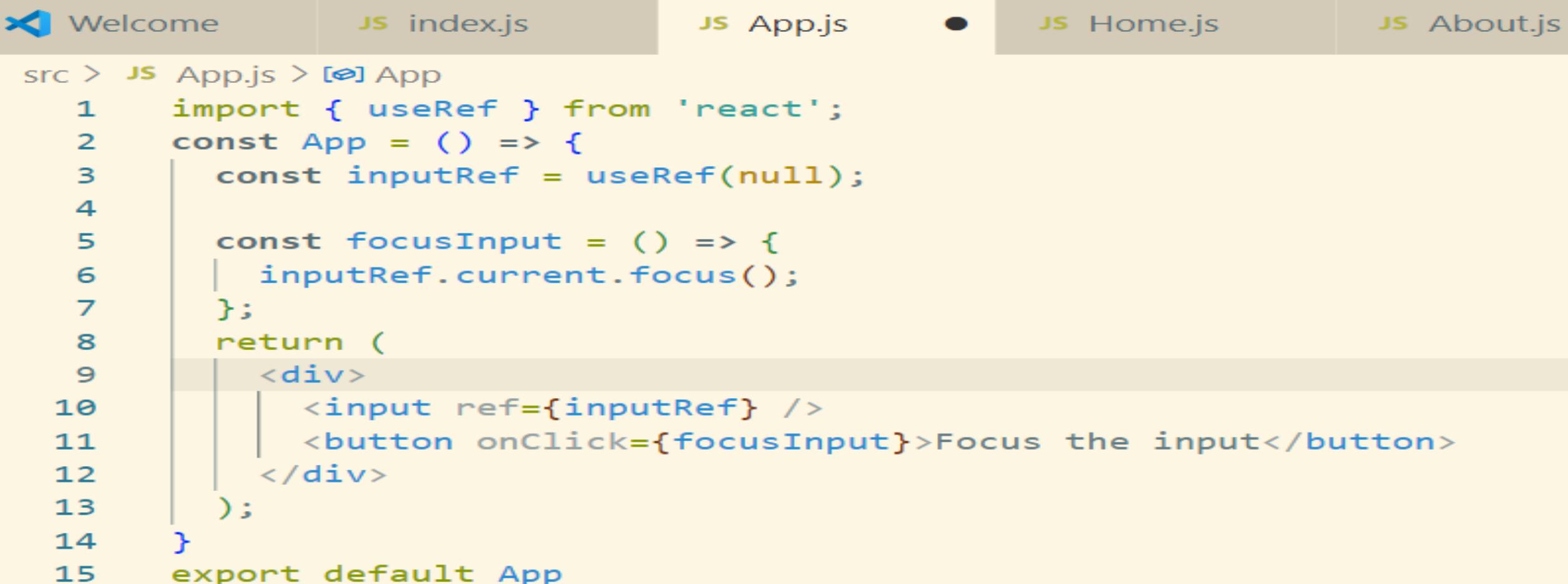
Steps to implement Ref Hook

Step 1: Import the useRef hook from React.

Step 2: Create a ref object using useRef.

Step 3: Attach the ref to a DOM element using the ref attribute.

Step 4: Use the current property of the ref to interact with the DOM element.



The screenshot shows a code editor interface with a navigation bar at the top. The navigation bar has tabs for 'Welcome', 'index.js', 'App.js' (which is currently selected), 'Home.js', and 'About.js'. Below the navigation bar, the code editor displays the contents of the 'App.js' file. The code uses the 'useRef' hook to create a reference to an input element, which can then be focused programmatically.

```
src > JS App.js > [o] App
1 import { useRef } from 'react';
2 const App = () => {
3   const inputRef = useRef(null);
4
5   const focusInput = () => {
6     inputRef.current.focus();
7   };
8   return (
9     <div>
10    <input ref={inputRef} />
11    <button onClick={focusInput}>Focus the input</button>
12  </div>
13);
14
15 export default App
```

Steps to implement Callback Hook

Step 1: Import the `useCallback` hook from React.

Step 2: Define a function and wrap it with `useCallback` to memoize it.

Step 3: Specify dependencies to determine when the function should be re-created

1.4.3: Performing Performance Optimisation

Use `useMemo`

The `useMemo` hook is useful for optimizing expensive computations by memorizing their results until dependencies change.

We identified the multiplication as an expensive computation.

The `useMemo` hook caches the result, and recalculates only if `count` or `multiplier` changes

Steps to use useMemo

- Step 1. Identify expensive computations or derived data
- Step 2. Wrap the computation inside a useMemo hook, passing the function that returns the computed value as the first argument.
- Step 3. Provide a dependency array as the second argument, which will trigger re-computation when any of the dependencies change.
- Step 4. Use the memorized value where necessary.

Steps to Use useCallback Hook

- Step 1. Identify callback functions or event handlers that are passed as props to child components or that are recreated on every render.
- Step 2. Wrap the function definition inside a useCallback hook, passing the function as the first argument.
- Step 3. Provide a dependency array as the second argument to control when the function gets recreated.

```
1 import React, { useState, useCallback } from "react";
2 const ChildComponent = React.memo(({ handleClick }) => {
3   console.log("Child component re-rendered");
4   return <button onClick={handleClick}>Click Me</button>;
5 });
6 const ParentComponent = () => {
7   const [count, setCount] = useState(0);
8
9   // Memoize the callback with useCallback
10  const handleClick = useCallback(() => {
11    console.log("Button clicked");
12  }, []);
13
14  return (
15    <div>
16      <h2>Count: {count}</h2>
17      <button onClick={() => setCount(count + 1)}>Increment Count</button>
18      <ChildComponent handleClick={handleClick} />
19    </div>
20  );
21};
22 export default ParentComponent;
23
```

Optimize rendering

Techniques like memoization of components, splitting complex components are used to avoid unnecessary state updates, and lazy loading.

- ❖ **Memorize Components:** ComplexComponent is wrapped in React.memo to avoid unnecessary re-renders.
- ❖ **Lazy Loading:** LazyLoadedComponent is loaded only when needed using **lazy** and **Suspense**, reducing the initial load time.
- ❖ **Avoid Unnecessary State Updates:** The updateCount function directly increments the state, preventing extra logic that might trigger unwanted re-renders.

Steps to optimize rendering

Step 1. Memorize components

Step 2. Split complex components

Step 3. Embed Lazy loading method

```
1 import React, { useState, lazy, Suspense } from "react";
2 // Lazy Load heavy components
3 const LazyLoadedComponent = lazy(() => import("./LazyLoadedComponent"));
4 const ComplexComponent = React.memo(({ data }) => {
5   console.log("Complex component re-rendered");
6   return (
7     <div>
8       <h3>Complex Component</h3>
9       <p>{data}</p>
10      </div>
11    );
12  });
13 const OptimizedComponent = () => {
14   const [count, setCount] = useState(0);
15   const [data, setData] = useState("Some data");
16   const updateCount = () => setCount((prev) => prev + 1);
17   return (
18     <div>
19       <h1>Optimized Rendering Example</h1>
20       <button onClick={updateCount}>Increment Count</button>
21       <Suspense fallback={<div>Loading...</div>}>
22         <LazyLoadedComponent />
23       </Suspense>
24       <ComplexComponent data={data} />
25     </div>
26   );
27 };
28
29 export default OptimizedComponent;
```

1.4.4: Handling Complex State Logic

Step 1. Import useReducer from React.

Step 1. Import useReducer from React.

This hook manages complex state logic in React components.

```
import { useReducer } from 'react';
```

Step 2. Define the Initial State:

Create an initial state object.

```
const initialState = { count: 0 };
```

Step 3. Create the Reducer Function:

The reducer function accepts the current state and an action. Based on the action type, it returns a new state.

```
function reducer(state, action) {  
  switch (action.type) {  
    case 'increment':  
      return { count: state.count + 1 };  
    case 'decrement':  
      return { count: state.count - 1 };  
    default:  
      throw new Error();  
  }  
}
```

Step 4. Use the useReducer Hook:

Inside the functional component (Hooks), invoke `useReducer`. It takes the reducer and the `initialState`, returning an array with the current state (`state`) and the dispatch function (`dispatch`).

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

Step 5. Handle Button Clicks:

Use dispatch to send actions to the reducer when the buttons are clicked.

Increment action:

```
<button onClick={() => dispatch({ type: 'increment' })}>+</button>
```

Decrement action:

```
<button onClick={() => dispatch({ type: 'decrement' })}>-</button>
```

The entire component will look as follows:

src > includes > **JS** Hooks.js >  reducer

```
1 import { useReducer } from 'react';
2 const initialState = { count: 0 };
3 function reducer(state, action) {
4   switch (action.type) {
5     case 'increment':
6       return { count: state.count + 1 };
7     case 'decrement':
8       return { count: state.count - 1 };
9     default:
10       throw new Error();
11   }
12 }
13 const Hooks = () => {
14   const [state, dispatch] = useReducer(reducer, initialState);
15   return (
16     <div>
17       <p>Count: {state.count}</p>
18       <button onClick={() => dispatch({ type: 'increment' })}>+</button>
19       <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
20     </div>
21   );
22 }
23 export default Hooks
```

1.4.5: Managing Global State

Context API

The Context API is built into React and is a relatively simple solution for managing global state without additional libraries.

Steps to use Context API

Step 1. Create a Context:

Create a new context using `React.createContext()`.

```
const GlobalStateContext = React.createContext();
```

Step 2. Provide the Context:

Wrap your app or components that need access to the global state with a `Context.Provider`.

```
const GlobalStateProvider = ({ children }) => {
  const [state, setState] = useState(initialState);
  return (
    <GlobalStateContext.Provider value={{ state, setState }}>
      {children}
    </GlobalStateContext.Provider>
  );
};
```

Step 3. Consume the Context:

Access the global state from any component using `useContext`.

```
const { state, setState } = useContext(GlobalStateContext);
```

The Context API works well for small to medium applications, but for complex state management, consider other options.

Redux

Redux is a popular state management library that allows a single store to manage global state in a predictable way, using actions and reducers.

Step 1. Install Redux:

Install the necessary packages.

```
npm install redux reactredux
```

Step 2. Create Reducers:

A reducer is a function that returns a new state based on the current state and the action dispatched.

```
const initialState = { count: 0 };
const counterReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    case 'DECREMENT':
      return { count: state.count - 1 };
    default:
      return state;
  }
};
```

Step 3. Create a Redux Store:

Create a store using createStore.

```
import { createStore } from 'redux';
const store = createStore(counterReducer);
```

Step 4. Provide the Store:

Use Provider from reactredux to wrap your app and provide the Redux store.

```
import { Provider } from 'reactredux';
<Provider store={store}>
  <App />
</Provider>;
```

Step 5. Connect Components:

Use the `useSelector` and `useDispatch` hooks to access and update the state from your components.

```
import { useSelector, useDispatch } from 'reactredux';
const Counter = () => {
  const count = useSelector((state) => state.count);
  const dispatch = useDispatch();
  return (
    <div>
      <button onClick={() => dispatch({ type: 'DECREMENT' })}></button>
      <span>{count}</span>
      <button onClick={() => dispatch({ type: 'INCREMENT' })}>+</button>
    </div>
  );
};
```

Redux is suitable for larger applications where state mutations are complex, and you need predictability.

MobX

MobX is a state management library that leverages observable state and reactions, making it more flexible and less boilerplate than Redux.

Steps to use MobX

Step 1. Install MobX:

Install MobX and its React bindings.

```
npm install mobx mobxreactlite
```

Step 2. Create Observable State:

Use `makeAutoObservable` to make your state observable.

```
import { makeAutoObservable } from 'mobx';
class CounterStore {
  count = 0;
  constructor() {
    makeAutoObservable(this);
  }
  increment() {
    this.count++;
  }
  decrement() {
    this.count--;
  }
}
const counterStore = new CounterStore();
```

Step 3. Provide Store:

Use React's context to provide the MobX store to the components.

```
const StoreContext = React.createContext(counterStore);
const StoreProvider = ({ children }) => {
  return (
    <StoreContext.Provider value={counterStore}>
      {children}
    </StoreContext.Provider>
  );
};
```

4. Use Observer in Components:

Use the observer function to make components reactively update when the store changes.

```
import { observer } from 'mobx-react-lite';
import { useContext } from 'react';
const Counter = observer(() => {
  const store = useContext(StoreContext);
  return (
    <div>
      <button onClick={() => store.decrement()}></button>
      <span>{store.count}</span>
      <button onClick={() => store.increment()}>+</button>
    </div>
  );
});
```

MobX allows more flexible and less restrictive state management, especially in applications that require more fluid, observable states.

Zustand

Zustand is a lightweight state management library that is simpler and less boilerplate than Redux or MobX.

Steps to use Zustand:

Step 1. Install Zustand:

Install the library.

npm install zustand

Step 2. Create a Store:

Create a Zustand store using `create`.

```
import create from 'zustand';
const useStore = create((set) => ({
  count: 0,
  increment: () => set((state) => ({ count: state.count + 1 })),
  decrement: () => set((state) => ({ count: state.count - 1 })),
}))
```

Step 3. Use the Store in Components:

Use the Zustand store in your React components with hooks.

```
const Counter = () => {
  const { count, increment, decrement } = useStore();
  return (
    <div>
      <button onClick={decrement}></button>
      <span>{count}</span>
      <button onClick={increment}>+</button>
    </div>
  );
};
```

Zustand is ideal for small to medium applications where you want simple and fast state management with minimal boilerplate.

Comparison:

- ❖ Context API: Best for small to medium apps with simple state.
- ❖ Redux: Ideal for large, complex apps where state management needs to be predictable.
- ❖ MobX: Great for apps with reactive state and complex workflows, offering flexibility.
- ❖ Zustand: A simple and lightweight solution for smaller apps with fewer state management requirements.

Points to Remember

Types of Hooks

1. State Hooks (`useState`)
2. Effect Hooks (`useEffect`)
3. Context Hooks (`useContext`)
4. Ref Hooks (`useRef`)
5. Callback Hooks (`useCallback`)

Additional Hooks:

1. `useReducer`
2. `useMemo`
3. `useLayoutEffect`
4. `useImperativeHandle`

Factors for Selecting the Right Hook

1. State Complexity
2. Side Effects
3. Optimization
4. Context Needs
5. DOM Access

Reasons for Combining Hooks

1. Modularity
2. Optimization
3. Code Organization
4. Reusability

Steps to implement State Hooks

1. Import the useState hook from React.
2. Use useState to define a state variable and a function to update it.
3. Use the state variable in your component's render logic.
4. Call the state updater function to modify the state based on an event.

Steps to implement Effect Hook

1. Import the useEffect hook from React.
2. Define the effect logic inside the useEffect callback.
3. Optionally return a cleanup function to run when the component unmounts.
4. Add dependencies in the second argument to control when the effect runs.

Steps to implement Ref Hook

1. Import the useRef hook from React.
2. Create a ref object using useRef.
3. Attach the ref to a DOM element using the ref attribute.
4. Use the current property of the ref to interact with the DOM element.

Steps to implement Callback Hook

1. Import the `useCallback` hook from React.
2. Define a function and wrap it with `useCallback` to memorize it.
3. Specify dependencies to determine when the function should be re-created

To optimize performance, you have to use following hooks:

1. `useMemo`
2. `useCallback`
3. `useReducer`

Global state:

- ❖ Context API
- ❖ Create a Context
- ❖ Provide the Context
- ❖ Consume the Context

Redux

1. Install Redux
2. Create Reducers
3. Create a Redux Store
4. Provide the Store
5. Connect Components

MobX

1. Install MobX
2. Create Observable State
3. Provide Store
4. Use Observer in Components

Zustand:

1. Install Zustand
2. Create a Store
3. Use the Store in Components

Application of learning 1.4.

ABC is a web Application development company located in Kimihurura – Kigali city, it's planning to create a customer's shopping site using React. As a React developer, the company has hired you to build a dashboard application using React. Utilize State Hooks to manage local component states and Effect Hooks for side effects like data fetching. To optimize performance, you have to strategically combine hooks and implement Context API for managing global state, this will allow seamless data access across components. As the app grows, they decide to integrate Redux for more complex state management, ensuring that components re-render efficiently and maintain synchronization with the global state. Throughout the process, employ Ref Hooks to access DOM elements directly and improve performance.

1.5: Implementation of Events Handling

1.5.1. Description of ReactJS Events

Description of key concepts

Events

Events are actions that occur as a result of user interactions or browser actions, such as clicks, typing, submitting a form, or scrolling. React uses a system of event handling similar to DOM event handling but with a unified approach through Synthetic Events. These events help manage user inputs and trigger specific functions within a component.

Types of Events

React supports many event types, similar to native HTML/DOM events, but they are implemented as synthetic events. Some common types include:

A Synthetic Event is a cross-browser wrapper around the browser's native event system, used primarily in React. It provides a consistent API for event handling regardless of the browser being used.

Why Synthetic Events Exist

Cross-browser compatibility: Different browsers handle events differently (especially older ones like IE). Synthetic Events normalize this behavior.

Performance: React uses event delegation - instead of attaching event handlers to individual DOM nodes, it attaches a single handler at the root and dispatches events as needed.

Key Characteristics

/ Example usage

```
function Button() {  
  const handleClick = (event) => {  
    // `event` is a SyntheticEvent  
    event.preventDefault(); // Works in all browsers
```

```
event.stopPropagation(); // Consistent across browsers  
  console.log(event.type); // 'click'  
  console.log(event.target); // The DOM element  
};
```

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

Important Properties

Synthetic Events have the same interface as native events, including:

- ❖ **event.target** - The element that triggered the event
- ❖ **event.currentTarget** - The element the handler is attached to
- ❖ **event.preventDefault()** - Prevent default behavior
- ❖ **event.stopPropagation()** - Stop event bubbling
- ❖ **event.nativeEvent** - Access to the underlying native event

Mouse Events:

- ❖ **onClick**: Fires when an element is clicked.
- ❖ **onDoubleClick**: Fires when an element is double clicked.
- ❖ **onMouseEnter**: Fires when the mouse pointer enters an element.
- ❖ **onMouseLeave**: Fires when the mouse pointer leaves an element.

Keyboard Events:

- ❖ **onKeyDown**: Fires when a key is pressed.
- ❖ **onKeyUp**: Fires when a key is released.
- ❖ **onKeyPress**: Fires when a key is pressed, but this event is considered deprecated in favor of **onKeyDown** and **onKeyUp**.

Supported Events

Common Synthetic Events in React include:

`onClick, onDoubleClick`

`onChange, onInput, onSubmit`

`onKeyDown, onKeyUp`

`onMouseEnter, onMouseLeave`

`onFocus, onBlur`

`onScroll, onWheel`

When to Use Native Events

You can access the native event if needed:

```
handleClick = (syntheticEvent) => {  
  const nativeEvent = syntheticEvent.nativeEvent;  
  // Use native event for specific browser APIs  
};
```

Benefits

- **Consistency:** Same behavior across all browsers
- **Automatic cleanup:** React handles event listener cleanup
- **Performance optimization:** More efficient event handling
- **Future-proof:** Browser inconsistencies are abstracted away

Synthetic Events are a key part of React's abstraction that makes building cross-browser applications easier and more reliable.

what is native event?

A Native Event is the browser's built-in event object that is created by the DOM API when an event occurs. It's the raw, browser-specific event that you would work with in vanilla JavaScript without any frameworks or libraries.

Native Event in Vanilla JavaScript

```
// Direct DOM event handling
const button = document.getElementById('myButton');
button.addEventListener('click', function(event) {
  // `event` is a native event
  console.log(event instanceof Event); // true
  console.log(event.type); // 'click'
  console.log(event.target); // The button element
});
```

Key Characteristics

- Browser-specific implementation: Different browsers may have slightly different APIs or behaviors
- Direct DOM access: No abstraction layer between your code and the browser
- Standard Web API: Part of the official DOM specification

Common Native Event Types

/ Mouse events

```
element.addEventListener('click', (e) => {});  
element.addEventListener('mousedown', (e) => {});  
element.addEventListener('mouseup', (e) => {});  
element.addEventListener('mousemove', (e) => {});
```

// Keyboard events

```
element.addEventListener('keydown', (e) => {});  
element.addEventListener('keyup', (e) => {});  
element.addEventListener('keypress', (e) => {});
```

```
// Form events
element.addEventListener('submit', (e) => {});
element.addEventListener('change', (e) => {});
element.addEventListener('input', (e) => {});

// UI events
element.addEventListener('scroll', (e) => {});
element.addEventListener('resize', (e) => {});
element.addEventListener('load', (e) => {});
```

When to Use Native Events Over Synthetic

- ❖ **Outside React's virtual DOM:** When working with third-party libraries
- ❖ **Performance-critical scenarios:** Direct DOM manipulation
- ❖ **Advanced browser APIs:** Features not abstracted by React
- ❖ **Global event listeners:** window, document events

```
// Global resize listener (not in React's scope)  
window.addEventListener('resize', handleResize);
```

```
// Direct DOM manipulation
```

```
const canvas = document.getElementById('myCanvas');  
canvas.addEventListener('mousemove', drawOnCanvas);
```

Native events are the foundation of browser interactivity, while synthetic events (like in React) provide a consistent, optimized abstraction layer on top of them.

Form Events:

- ❖ **onChange**: Fires when the value of an input element changes.
- ❖ **onSubmit**: Fires when a form is submitted.

Focus Events:

- ❖ **onFocus**: Fires when an element receives focus.
- ❖ **onBlur**: Fires when an element loses focus.

Touch Events (for touch devices):

- ❖ **onTouchStart**: Fires when a touch event begins.
- ❖ **onTouchMove**: Fires when a touch event moves.
- ❖ **onTouchEnd**: Fires when a touch event ends.

Scroll Events:

onScroll: Fires when an element's scroll position changes.

Clipboard Events:

- ❖ onCopy: Fires when content is copied.
- ❖ onPaste: Fires when content is pasted.

Synthetic Events

- ❖ Synthetic events are **React's cross-browser wrapper around the native browser events**. This system ensures that the events behave consistently across different browsers, simplifying event handling for developers. Synthetic events are normalized, meaning you can write event-handling code that works reliably across all browsers without worrying about discrepancies in how browsers implement events.

Event Bubbling

Event bubbling refers to the process where an event triggered on an inner element propagates up through its parent elements. In React, when an event occurs on a child element, it bubbles up to its parent and can trigger any event handlers defined on the parent for that same event type.

❖ **Debouncing and Throttling**

Both debouncing and throttling are techniques used to control how often a function is executed in response to an event, improving performance when dealing with events like scrolling, typing, or resizing.

Debouncing:

Debouncing ensures that a function is called only after a certain period of inactivity. This is useful for events that can fire in rapid succession, such as keystrokes or window resizing. **The function will only execute after the user has stopped triggering the event for a specified delay.**

Throttling:

Throttling ensures that a function is executed at most once every specified interval, even if the event is triggered continuously. It limits the number of times the function is executed over time.

Example use case: Handling window scrolling events, where you want to limit how often a handler is called while the user is scrolling.

1.5.2: USING CONTROLLED COMPONENTS

STEPS TO USE CONTROLLED COMPONENTS

Step 1. Create a functional or class component where you will manage your form state.

Step 2. Use the useState hook (for functional components) or this.state (for class components) to create state variables that will hold the values of your form inputs.

```
1 import React, { useState } from 'react';
2
3 const Hooks = () => {
4     // Step 2: Create state variables for form inputs
5     const [formData, setFormData] = useState({
6         username: '',
7         email: '',
8         password: '',
9         agreeToTerms: false,
10        gender: ''
11    });
12
```

Step 3. Add input fields to your component and set their value prop to the corresponding state variable. This binds the input field's value to the state.

```
29   return (
30     <form onSubmit={handleSubmit}>
31       /* Step 3: Bind state to input fields */
```

Step 4. Implement an onChange event handler for each input element that updates the state. This ensures the component re-renders with the new value.

```
// Step 4: Event handler to update state when input changes
const handleInputChange = (e) => {
  const { name, value, type, checked } = e.target;
  setFormData({
    ...formData,
    [name]: type === 'checkbox' ? checked : value // Handle checkbox
  });
};
```

Step 5. Create a function to handle form submission. This function can prevent the default form submission behaviour and access the state values.

```
// Step 5: Form submission handler
const handleSubmit = (e) => {
  e.preventDefault(); // Prevent default form submission
  // You can now access formData values here for form submission or validation
  console.log('Form submitted with data:', formData);
};
```

Step 6. Use the state values in your component logic, such as for form validation, submission, or displaying the values elsewhere.

Step 7. You can repeat these steps for other input types like checkboxes, radio buttons, or selects.

Full Sample source code

```
import React, { useState } from 'react';
const Hooks = () => {
  // Step 2: Create state variables for form inputs
  const [formData, setFormData] = useState({
    username: '',
    email: '',
    password: '',
    agreeToTerms: false,
    gender: ''
  });
  // Step 4: Event handler to update state when input changes
  const handleInputChange = (e) => {
    const { name, value, type, checked } = e.target;
    setFormData({
      ...formData,
      [name]: type === 'checkbox' ? checked : value // Handle checkbox
    });
  };
};
```

```
// Step 5: Form submission handler
const handleSubmit = (e) => {
  e.preventDefault(); // Prevent default form submission
  // You can now access formData values here for form submission or validation
  console.log('Form submitted with data:', formData);
};

return (
  <form onSubmit={handleSubmit}>
    {/* Step 3: Bind state to input fields */}
    <div>
      <label>Username: </label>
      <input
        type="text"
        name="username"
        value={formData.username}
        onChange={handleInputChange}
      />
    </div>
)
```

```
<div>
<label>Email: </label>
<input
type="email"
name="email"
value={formData.email}
onChange={handleInputChange}
/>
</div>
<div>
<label>Password: </label>
<input
type="password"
name="password"
value={formData.password}
onChange={handleInputChange}
/>
</div>
<div>
<label>
```

```
<input  
type="checkbox"  
name="agreeToTerms"  
checked={formData.agreeToTerms}  
onChange={handleInputChange}  
/>
```

I agree to the terms and conditions

```
</label>  
</div>
```

```
<div>
```

```
<label>Gender: </label>
```

```
<select  
name="gender"  
value={formData.gender}  
onChange={handleInputChange}>  
<option value="">Select</option>  
<option value="male">Male</option>
```

```
<option value="female">Female</option>
</select>
</div>
<button type="submit">Submit</button>
</form>
);
};
export default Hooks;
```

Output of the code:

Hello World

[**Home**](#) [**About Us**](#) [**Contact Us**](#) [**Hooks**](#)

Username:

Email:

Password:

I agree to the terms and conditions

Gender:

1.5.3 : Passing Arguments to Event Handlers

Steps to Pass Arguments to Event Handlers

Step 1. Create a functional or class component where you will manage your form state.

step 2. Use the useState hook (for functional components) or this.state (for class components) to create state variables that will hold the values of your form inputs.

src > includes > `JS HOOKS.JS` > `HOOKS`

```
1 import React, { useState } from 'react';
2
3 const Hooks = () => {
4   // 1. Initialize state variables for each form field
5   const [name, setName] = useState('');
6   const [email, setEmail] = useState('');
7   const [password, setPassword] = useState('');
8   const [gender, setGender] = useState('');
9   const [isSubscribed, setIsSubscribed] = useState(false);
10
```

Step 3. Add input fields to your component and set their value prop to the corresponding state variable. This binds the input field's value to the state.

Step 4. Implement an onChange event handler for each input element that updates the state. This ensures the component re-renders with the new value.

```
// 2. Event handler for input changes
const handleInputChange = (event) => {
  const { name, value, type, checked } = event.target;

  // Update the corresponding state based on input type
  if (type === 'checkbox') {
    setIsSubscribed(checked);
  } else if (name === 'name') {
    setName(value);
  } else if (name === 'email') {
    setEmail(value);
  } else if (name === 'password') {
    setPassword(value);
  } else if (name === 'gender') {
    setGender(value);
  }
};
```

Step 5. Create a function to handle form submission. This function can prevent the default form submission behaviour and access the state values.

Step 6. Use the state values in your component logic, such as for form validation, submission, or displaying the values elsewhere.

Step 7. You can repeat these steps for other input types like checkboxes, radio buttons, or selects.

Sample code

```
import React, { useState } from 'react';
const Hooks = () => {
  // 1. Initialize state variables for each form field
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");
  const [gender, setGender] = useState("");
  const [isSubscribed, setIsSubscribed] = useState(false);
```

```
// 2. Event handler for input changes
const handleInputChange = (event) => {
  const { name, value, type, checked } = event.target;
  // Update the corresponding state based on input type
  if (type === 'checkbox') {
    setIsSubscribed(checked);
  } else if (name === 'name') {
    setName(value);
  } else if (name === 'email') {
    setEmail(value);
  } else if (name === 'password') {
    setPassword(value);
  } else if (name === 'gender') {
    setGender(value);
  }
};
```

```
// 3. Form submit handler
const handleSubmit = (event) => {
  event.preventDefault();
  // Access state values
  console.log('Form submitted with the following values:');
  console.log('Name:', name);
  console.log('Email:', email);
  console.log('Password:', password);
  console.log('Gender:', gender);
  console.log('Subscribed:', isSubscribed);
  // You can add form validation or submit logic here
};
return (
  <form onSubmit={handleSubmit}>
    <div>
      <label>Name:</label>
```

```
<input  
  type="text"  
  name="name"  
  value={name}  
  onChange={handleInputChange}  
/>  
</div>  
<div>  
  <label>Email:</label>  
  <input  
    type="email"  
    name="email"  
    value={email}  
    onChange={handleInputChange}  
  />  
</div>
```

```
<div>
  <label>Password:</label>
  <input type="password"
    name="password"
    value={password} onChange={handleInputChange}
  />
</div>
<div>
  <label>Gender:</label>
  <select name="gender" value={gender} onChange={handleInputChange}>
    <option value="">Select Gender</option>
    <option value="male">Male</option>
    <option value="female">Female</option>
  </select>
```

```
</div>
<div>
<label>Subscribe to newsletter:</label>
<input
  type="checkbox"
  name="isSubscribed"
  checked={isSubscribed}
  onChange={handleInputChange}
/>
</div>
<button type="submit">Submit</button>
</form>
);
};
export default Hooks;
```

Output of the code

[Home](#) [About Us](#) [Contact Us](#) [Hooks](#)

Name:

Email:

Password:

Gender:

Subscribe to newsletter:

1.5.4: Using Custom Hooks for Event Listeners

Steps to **use Custom Hooks for Event Listeners**

```
import { useEffect, useRef } from 'react';

function useEventListener(eventName, handler, element = window) {
  // Create a ref that stores the handler
  const savedHandler = useRef();
  // Update ref.current value if handler changes
  useEffect(() => {
    savedHandler.current = handler;
  }, [handler]);
  useEffect(() => {
    // Make sure the element supports addEventListener
    const isSupported = element && element.addEventListener;
    if (!isSupported) return;
    // Create an event listener that calls the saved handler
    const eventListener = (event) => savedHandler.current(event);
    element.addEventListener(eventName, eventListener);
    return () => element.removeEventListener(eventName, eventListener);
  }, [element, eventName]);
}
```

```
// Add event listener
element.addEventListener(eventName, eventListener);
// Remove event listener on cleanup
return () => {
  element.removeEventListener(eventName, eventListener);
};
}, [eventName, element]); // Rerun if eventName or element changes
}
```

```
import React, { useState, useRef } from 'react';
import useEventListener from './useEventListener'; // Import your custom hook
function ExampleComponent() {
  const [coords, setCoords] = useState({ x: 0, y: 0 });
  const buttonRef = useRef(null);
  // Handler for window mousemove event
```

Step 2. Use the useEventListener Hook in a Component:

```
import React, { useState, useRef } from 'react';
import useEventListener from './useEventListener'; // Import your custom hook
function ExampleComponent() {
  const [coords, setCoords] = useState({ x: 0, y: 0 });
  const buttonRef = useRef(null);
  // Handler for window mousemove event
  const handleMouseMove = (event) => {
    setCoords({ x: event.clientX, y: event.clientY });
  };
  // Attach mousemove event to the window
  useEventListener('mousemove', handleMouseMove);
  // Handle button click
  const handleClick = () => {
    console.log('Button clicked!');
  };
}
```

```
// Attach mousemove event to the window
useEventListener('mousemove', handleMouseMove);
// Handle button click
const handleButtonClick = () => {
  console.log('Button clicked!');
};
// Attach click event to the button
useEventListener('click', handleButtonClick, buttonRef.current);
return (
  <div>
    <p>Mouse position: {coords.x}, {coords.y}</p>
    <button ref={buttonRef}>Click me!</button>
  </div>
);
}
export default ExampleComponent;
```

Note: Handling Dynamic Elements:

If the event listener is attached to dynamic or conditionally rendered elements, the hook should gracefully handle null or undefined values (which it does by checking `isSupported`). For instance, in the above case, `buttonRef.current` will initially be null until the button is mounted.

The hook will only add the event listener once the element exists, and will clean it up when the component is unmounted or the element is removed.

1.5.5: Handling Events on Dynamic Lists

Steps to Handle Events on Dynamic Lists

Step 1. Set Up State for the List

Create a state variable to manage the dynamic list. This is usually done using the `useState` hook.

```
const [list, setList] = useState([]);
```

Step 2. Attach Event Handlers to List Items

Each list item should have an event handler attached (e.g., onClick, onMouseOver). You can pass a handler function to each item in the list.

```
list.map((item, index) => (  
<li key={index} onClick={() => handleItemClick(item)}>  
{item.name}  
</li>  
));
```

Step 3. Handle Dynamic Updates (Add/Delete Items)

Use the setList function to add or delete items from the list dynamically.

```
const addItem = (newItem) => {  
setList((prevList) => [...prevList, newItem]);  
};  
const removeItem = (id) => {  
setList((prevList) => prevList.filter(item => item.id !== id));  
};
```

Step 4. Pass Data to Event Handlers

Pass the appropriate data (such as the clicked item's data) into the event handler.

```
const handleItemClick = (item) => {  
  console.log(item clicked: ${item.name});  
};
```

Step 5. Conditional Rendering for Dynamic Behavior

Depending on certain conditions, you might want to render different content or apply different styles.

```
{list.length > 0 ? (  
  <ul>{/* Render list */}</ul>  
) : (  
  <p>No items available</p>  
)}
```

Step 6. Update the State on Event Trigger

When an event is triggered (like clicking an item), ensure it updates the state appropriately.

```
const handleItemClick = (item) => {  
  setSelectedItem(item);  
};
```

Step 7. Use Proper key Props in List Rendering

When rendering lists, ensure each item has a unique key for optimal performance.

```
list.map((item) => (  
  <li key={item.id}>{item.name}</li>  
));
```

Step 8. Use useCallback or memo to Prevent Unnecessary Re-renders

If the functions passed to components are recreated unnecessarily, it can lead to performance issues. Use useCallback to memorize functions.

```
const handleItemClick = useCallback((item) => {  
  // Handle click event  
}, []);
```

Note: You can also use React.memo to optimize components that do not need to re-render on every state change.

```
const ListItem = React.memo(({ item, onClick }) => (  
<li onClick={() => onClick(item)}>{item.name}</li>  
));
```

Points to Remember

Types of Events include User Interface Events, Focus and Blur Events, Form Events, Mouse Events, Keyboard Events, Synthetic Events

Event Bubbling

Debouncing and throttling

Steps to use Controlled Components

- Create a functional or class component where you will manage your form state. Use the useState hook (for functional components) or this.state (for class components) to create state variables that will hold the values of your form inputs.
- Add input fields to your component and set their value prop to the corresponding state variable. This binds the input field's value to the state.
- Implement an onChange event handler for each input element that updates the state. This ensures the component re-renders with the new value.
- Create a function to handle form submission. This function can prevent the default form submission behaviour and access the state values.

- Use the state values in your component logic, such as for form validation, submission, or displaying the values elsewhere.
- You can repeat these steps for other input types like checkboxes, radio buttons, or selects.

- Steps to Pass Arguments to Event Handlers

- Create a functional or class component where you will manage your form state.
- Use the `useState` hook (for functional components) or `this.state` (for class components) to create state variables that will hold the values of your form inputs.
- Add input fields to your component and set their `value prop` to the corresponding state variable. This binds the input field's value to the state.
- Implement an `onChange` event handler for each input element that updates the state. This ensures the component re-renders with the new value.
- Create a function to handle form submission. This function can prevent the default form submission behaviour and access the state values.

- Use the state values in your component logic, such as for form validation, submission, or displaying the values elsewhere.
- You can repeat these steps for other input types like checkboxes, radio buttons, or selects.

Steps to use Custom Hooks for Event Listeners

- Set Up State for the List
- Attach Event Handlers to List Items
- Handle Dynamic Updates (Add/Delete Items)
- Pass Data to Event Handlers
- Conditional Rendering for Dynamic Behavior
- Update the State on Event Trigger
- Use Proper key Props in List Rendering
- Use useCallback or memo to Prevent Unnecessary Re-renders

Application of learning 1.5.

ABC company is a company located in Niboye- Sector, Kicukiro District in Kigali City, It has many employs who are assigned daily task, Assume you have been hired as their web developer to develop them a Todo List app in React that will help the employee manager to add, remove, and mark tasks as complete. Implement event handling for user interactions, such as clicking buttons or pressing keys. Use debouncing for search input to improve performance and throttling for scrolling events to manage resource usage. Utilize controlled components to maintain the state of inputs. Pass arguments to event handlers using arrow functions and the bind method. Create custom hooks for managing event listeners, and ensure event handling works seamlessly on a dynamically generated list of tasks.

1.6: Implementation of API Integration

1.6.1. Description of API Integration

API in React.js

API (Application Programming Interface) allows your application to interact with external services, servers, or databases. APIs enable the communication between the front-end (React) and the back-end systems. This is commonly used to fetch or send data from/to external sources like databases, other web services, or cloud platforms. For instance, if you're building a weather app in React, you would call a weather API to fetch data about the weather and display it to the user.

Importance of API

- **Dynamic Data:** APIs allow React apps to retrieve dynamic data, such as user information, product details, or live weather, enabling the app to provide real-time updates without reloading.
- **Scalability:** React apps can connect to various external APIs to scale functionality, like integrating third-party services (e.g., payment gateways, social logins).
- **Separation of Concerns:** API integration ensures a separation between front-end (UI) and back-end (data/services). This makes the application architecture more modular and maintainable.
- **Interactivity:** By integrating APIs, react apps can make the user experience more interactive, such as updating data without refreshing the entire page (using AJAX or Fetch API).
- **Reusability:** API-based architectures allow you to reuse data and functionality across different platforms, such as web, mobile, or desktop applications.

Methods used for fetching data from APIs

Fetch API

The Fetch API is a built-in JavaScript feature used to make HTTP requests. It returns a promise, which you can resolve to get the response data.

Axios

Axios is a popular third-party library for making HTTP requests, known for its simplicity and ease of use. Unlike Fetch, it automatically converts response data to JSON and has better support for handling errors.

Async/Await:

Often used in combination with Fetch or Axios for cleaner and more readable asynchronous code.

React Query:

A powerful library for managing server-side state and handling API requests with caching, background syncing, and many other features.

1.6.2: Installing dependencies (Axios)

Steps to Install dependencies

Step 1. Ensure Node.js and npm are Installed

Run the following commands to verify if node or npm are installed

node -v

npm -v

Step 2. Create a React Project

Use npx to create a new React project.

npx create-react-app my-api-project

This will generate a React project called my-api-project.

Step 3. Install Project Dependencies for API Integration

After your project has been created, navigate to the project directory:

```
cd my-api-project
```

Install necessary dependencies for making API requests, such as axios. Axios is a popular library for making HTTP requests from React.

```
npm install axios
```

Step 4. Add New Dependencies for API Integration

At this point, axios is installed, and you can use it to perform API integration. There might be other packages depending on your specific requirements, such as react-router if you're working with routes, but for now, we are focusing on basic API calls.

Step 5. Create a Functional Component for API Integration

Create a simple functional component that fetches data from an API.

- ❖ Open the src folder.
- ❖ Inside the src folder, create a new file called ApiComponent.js.

```
import React, { useEffect, useState } from 'react';
import axios from 'axios';
const ApiComponent = () => {
  const [data, setData] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);
  useEffect(() => {
    // Replace with your API endpoint
    const fetchData = async () => {
      try {
        const response = await axios.get('https://jsonplaceholder.typicode.com/posts');
```

```
setData(response.data);
setLoading(false);
} catch (error) {
setError('Failed to fetch data');
setLoading(false);
}
};

fetchData();
}, []));
if (loading) return <p>Loading...</p>;
if (error) return <p>{error}</p>;
return (
<div>
<h1>API Data</h1>
<ul>
```

```
{data.map(item => (
  <li key={item.id}>{item.title}</li>
))
</ul>
</div>
);
};

export default ApiComponent;
```

In this example:

- **useState** is used to manage the state of the data, loading status, and errors.
- **useEffect** triggers the API call after the component is rendered.
- **axios.get()** is used to make a GET request to the API endpoint.

Step 6. Update App.js to Render the API Component

Open the src/App.js file and replace its content with the following to render ApiComponent:

```
import React from 'react';
import ApiComponent from './ApiComponent';
function App() {
  return (
    <div className="App">
      <ApiComponent />
    </div>
  );
}
export default App;
```

Step 7. Run the Project

Now you can run the project to test the API integration.

`npm start`

1.6.3 Steps to Defining and Grouping API Calls

Step 1. Install Axios (or use Fetch API) for API Calls

If using Axios, install it with the following command:

`npm install axios`

Step 2. Create a Separate API Utility File

Create a dedicated file (e.g., `api.js`) to handle all API interactions. This keeps the codebase organized and ensures API logic is reusable.

`src/utils/api.js`

Step 3. Define API Base URL and Common Configuration

Set a base URL for your API and create common configurations, like headers and timeouts, to avoid repeating this setup in every call.

```
import axios from 'axios';
const apiClient = axios.create({
  baseURL: 'https://api.example.com', // Replace with your API URL
  timeout: 10000,
  headers: {
    'Content-Type': 'application/json',
  },
});
export default apiClient; 88
```

Step 4. Define API Endpoints

Define functions for each API endpoint in the utility file. For example, you can create functions for GET, POST, PUT, DELETE requests, and map them to specific URLs.

```
export const getUsers = () => apiClient.get('/users');
export const createUser = (data) => apiClient.post('/users', data);
export const updateUser = (id, data) => apiClient.put('/users/${id}', data);
export const deleteUser = (id) => apiClient.delete('/users/${id}');
```

Step 5. Use API Calls in React Components

```
import React, { useEffect, useState } from 'react';
import { getUsers } from './utils/api';
const UsersList = () => {
  const [users, setUsers] = useState([]);
  useEffect(() => {
```

```
const fetchUsers = async () => {
  try {
    const response = await getUsers();
    setUsers(response.data);
  } catch (error) {
    console.error('Error fetching users:', error);
  }
};

fetchUsers();
}, []));
return (
<ul>
{users.map(user => (
<li key={user.id}>{user.name}</li>
))}</ul> );
}; export default UsersList;
```

Step 6. Perform Error Handling and Response Interception

Implement global error handling and intercept responses to manage authentication tokens or retry failed requests.

```
ApiClient.interceptors.response.use(  
  (response) => response,  
  (error) => {  
    if (error.response && error.response.status === 401) {  
      // Handle unauthorized access (e.g., redirect to login)  
    }  
    return Promise.reject(error);  
  }  
);
```

Step 7. Handle Caching and Optimizations

Implement caching strategies like memoizing API responses (e.g., using React Query or Redux). This minimizes unnecessary API calls and improves performance.

Example with React Query:

```
npm install react-query
```

```
import { useQuery } from 'react-query';
import { getUsers } from './utils/api';
const UsersList = () => {
  const { data: users, isLoading, error } = useQuery('users', getUsers);
  if (isLoading) return <div>Loading...</div>;
  if (error) return <div>Error loading users</div>;
```

```
return (
<ul>
{users.map(user => (
<li key={user.id}>{user.name}</li>
))}
</ul>
);
};

export default UsersList;
```

1.6.4: Handling Data Fetching and Responses

Steps to Handle Data Fetching and Responses

Step 1. Set Up State to Manage Data and Loading Status

```
import React, { useState, useEffect } from 'react';
const ExampleComponent = () => {
  const [data, setData] = useState(null); // State to store fetched data
  const [loading, setLoading] = useState(true); // State to manage loading status
  const [error, setError] = useState(null); // State to manage errors
};
```

Step 2. Use useEffect to Fetch Data When Component Loads

Use the `useEffect` hook to perform side effects like fetching data when the component is mounted. You can make use of `async/await` for cleaner code and error handling.

```
useEffect(() => {
  const fetchData = async () => {
    try {
      setLoading(true);
      const response = await fetch('https://api.example.com/data');
      if (!response.ok) {
        throw new Error('Failed to fetch');
      }
      const result = await response.json();
      setData(result);
    } catch (err) {
      setError(err.message);
    } finally {
      setLoading(false);
    }
  };
  fetchData();
}, []);
```

Step 3. Render Based on State (Loading, Error, or Data)

```
return (  
<div>  
{loading && <p>Loading...</p>}  
{error && <p>Error: {error}</p>}  
{data && <div>{JSON.stringify(data)}</div>}  
</div>  
);
```

Step 4. Handle POST or Other Methods

For handling other HTTP methods like POST, you can create a function that sends data using fetch. You might need additional state to manage the post request status.

```
const postData = async (newData) => {
```

```
try {
const response = await fetch('https://api.example.com/data', {
method: 'POST',
headers: {
'Content-Type': 'application/json',
},
body: JSON.stringify(newData),
});
if (!response.ok) {
throw new Error('Failed to post data');
}
const result = await response.json();
console.log('Post successful:', result);
} catch (err) {
console.error('Error posting data:', err.message);
} };
```

```
// Example of using postData when a button is clicked  
return (  
<div>  
<button onClick={() => postData({ name: 'New Data' })}>Submit Data</button>  
</div>  
);
```

1.6.5: Error Handling

Steps to perform Error Handling

Step 1. Choose an API Request Method

You can choose between native fetch or third-party libraries like axios. Both are widely used for making HTTP requests in React apps.

Fetch: Built into modern browsers.

Axios: Offers more features like automatic JSON parsing, request/response interceptors, etc.

```
// Using Fetch
fetch('https://api.example.com/data')
.then(response => response.json())
.catch(error => console.error('Error:', error));
// Using Axios
import axios from 'axios';
axios.get('https://api.example.com/data')
.then(response => console.log(response))
.catch(error => console.error('Error:', error));
```

Step 2. Set Up State for Loading, Error, and Data

In your component, set up state to handle loading, error, and data responses.

```
import { useState, useEffect } from 'react';
const MyComponent = () => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);
};
```

Step 3. Make the API Call with Error Handling

When making the API call, make sure to handle errors properly and update the state accordingly. You can do this inside a `useEffect` hook for fetching data when the component mounts.

```
useEffect(() => {
  const fetchData = async () => {
    setLoading(true); // Start loading
    setError(null); // Clear previous errors
    try {
      const response = await fetch('https://api.example.com/data');
      if (!response.ok) { // Handle non-2xx status codes
        throw new Error(`HTTP error! Status: ${response.status}`);
      }
      const result = await response.json();
      setData(result); // Set fetched data
    } catch (err) {
      setError(err.message); // Set error message
    } finally {
      setLoading(false); // End loading
    }
  }
})
```

```
};  
};  
fetchData();  
, []);
```

Step 4. Handle Errors Based on Status Code

You can also handle specific HTTP status codes differently.

```
try {  
const response = await fetch('https://api.example.com/data');  
if (response.status === 404) {  
throw new Error('Resource not found');  
} else if (response.status === 500) {  
throw new Error('Server error');  
}
```

```
const result = await response.json();
setData(result);
} catch (err) {
setError(err.message);
}
```

For Axios:

```
axios.get('https://api.example.com/data')
.then(response => setData(response.data))
.catch(error => {
if (error.response) {
if (error.response.status === 404) {
setError('Resource not found');
} else if (error.response.status === 500) {
setError('Server error');
```

```
}

} else {
  setError(error.message);
}
})
.finally(() => setLoading(false));
```

Step 5. Show Loading, Error, and Data States in the UI

Display different states based on the values of loading, error, and data.

```
return (
<div>
{loading && <p>Loading...</p>} {/ Show loading state /}
{error && <p>Error: {error}</p>} {/ Show error message /}
{data && (
<div>
{/ Render data /} <p>Data: {JSON.stringify(data)}</p></div>)}
```

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

Example with Axios:

```
import React, { useState, useEffect } from 'react';  
import axios from 'axios';  
const MyComponent = () => {  
  const [data, setData] = useState(null);  
  const [loading, setLoading] = useState(true);  
  const [error, setError] = useState(null);  
  useEffect(() => {  
    axios.get('https://api.example.com/data')  
      .then(response => setData(response.data))  
      .catch(error => setError(error.message))  
      .finally(() => setLoading(false));  
  }, []);
```

```
return (
<div>
{loading && <p>Loading...</p>}
{error && <p>Error: {error}</p>}
{data && <div>{JSON.stringify(data)}</div>}
</div>
);
};

export default MyComponent;
```

1.6.6:Handling Asynchronous and Concurrency