**Module-5) React - Styling &Amp; Advance React**

**Q:-1: What is conditional rendering in React? How can you conditionally render elements in a React component?**

* Conditional rendering in React means displaying elements or components based on certain conditions (like if, true/false, status, etc.).
* It works just like conditions in JavaScript, but inside JSX.
* React evaluates expressions and renders elements conditionally instead of always rendering everything.
* **Ways to conditionally render:**

1. Using if-else statements.
2. Using ternary operators (condition ? true : false).
3. Using logical AND (&&) for short conditions.
4. By storing conditional results in variables and returning them.

**Q:-2: Explain how if-else, ternary operators, and && (logical AND) are used in JSX for conditional rendering.**

* **If-Else Statement**
* Used when you need to handle multiple conditions clearly.
* You usually write the condition outside the JSX (inside the component function) and then return different elements based on the condition.
* Example: Show "Login" if isLoggedIn is false, otherwise show "Logout".
* **Ternary Operator (condition ? true : false)**
* A compact way of writing conditions directly inside JSX.
* It evaluates the condition and chooses between two elements.
* Best used for short and simple conditional rendering.
* Example: {isLoggedIn ? "Welcome Back!" : "Please Login"}.
* **Logical AND (&&) Operator**
* Used when you want to render something only if a condition is true.
* If the condition is false, React ignores the expression and renders nothing.
* Best for rendering optional elements.
* Example: {isLoggedIn && <h1>Welcome Back!</h1>}.

**7. Lists and Keys**

**Q:-1How do you render a list of items in React? Why is it important to use keys when rendering lists?**

React, lists are usually rendered using JavaScript’s map() function.

* You store data in an array.
* Then you use map() to loop over the array and return JSX for each item.

**Why is it important to use keys when rendering lists?**

* **Keys** help React **identify which items have changed, been added, or removed**.
* Without keys, React re-renders the whole list every time, which is inefficient.
* With keys, React can update only the changed elements, improving performance.
* Keys should be **unique and stable**, usually an **id** from your data (not array index unless you have no unique value).

**Q:-2 : What are keys in React, and what happens if you do not provide a unique key?**

**What are keys in React?**

* Keys are special attributes you add to elements when rendering lists in React.
* They help React identify each element uniquely.
* With keys, React can efficiently track, update, or remove list items without re-rendering the entire list

**What happens if you do not provide a unique key?**

* **Performance issues**

React will re-render all list items unnecessarily because it can’t tell which ones changed.

* **Incorrect UI updates**

Without unique keys, React may reuse old elements incorrectly.

* **Console warning**

React will show a warning:

**8. Forms in React**

**Q:-1: How do you handle forms in React? Explain the concept of controlled components.**

* **Handling Forms in React:**
* In React, forms are handled using **state**.
* Each form input (like text, email, checkbox) is tied to a **state variable**.
* When the user types or changes input, React updates the state using an **onChange handler**.
* The updated state reflects the current input value, making the form data always available in React.
* **Controlled Components:**
* A **controlled component** is an input form element (like <input>, <textarea>, <select>) that is **controlled by React state**.
* The value of the form element comes from React state (value={stateVariable}), not directly from the DOM.
* This means React has **full control** over the form’s data

**Q:-2: What is the difference between controlled and uncontrolled components in React?**

|  |  |  |
| --- | --- | --- |
| Aspect | Controlled Components | Uncontrolled Components |
| Definition | Form elements controlled by React state. | Form elements store data in the DOM itself. |
| Data Source | Value comes from React state (value={state}). | Value comes from the DOM (e.g., document.getElementById). |
| Accessing Value | Read the value directly from React state. | Use refs to access the DOM value. |
| Use Case | Preferred in most cases (predictable, easy validation). | Useful for simple forms where you don’t need full React control. |
| Example | <input value={state} onChange={...} /> | <input defaultValue="hello" ref={inputRef} /> |

**Q:-1: What are lifecycle methods in React class components? Describe the phases of a component’s lifecycle.**

**What are lifecycle methods?**

* In **React class components**, lifecycle methods are **special functions** that run at specific points during a component’s life.
* They let you control what happens when a component is **created (mounted)**, **updated (re-rendered)**, and **removed (unmounted)** from the DOM.

**Phases of a Component’s Lifecycle:**

1. **Mounting Phase (Component Creation)**
   * Happens when the component is first inserted into the DOM.
   * Common methods:
     + constructor() → Initialize state and bind methods.
     + render() → Returns JSX to display UI.
     + componentDidMount() → Runs **once after initial render**, good for API calls.
2. **Updating Phase (Component Re-rendering)**
   * Happens when props or state change.
   * Common methods:
     + render() → Re-renders UI when data changes.
     + componentDidUpdate(prevProps, prevState) → Runs after re-render, useful for side effects like fetching new data.
3. **Unmounting Phase (Component Removal)**
   * Happens when the component is removed from the DOM.
   * Common method:
     + componentWillUnmount() → Runs just before removal, used for cleanup (e.g., removing event listeners, clearing timers).

**Q:-2: Explain the purpose of componentDidMount(), componentDidUpdate(), and componentWillUnmount().**

1. **componentDidMount()**
   * Runs **only once**, immediately after the component is mounted (inserted into the DOM).
   * Common uses:
     + Fetching data from an API.
     + Starting timers.
     + Setting up subscriptions (e.g., WebSocket, events).
2. **componentDidUpdate(prevProps, prevState)**
   * Runs **after every update/re-render** (when state or props change).
   * Receives previous props and state as arguments.
   * Common uses:
     + Fetching new data when props change.
     + Running side effects after updates.
3. **componentWillUnmount()**
   * Runs **just before the component is removed** from the DOM.
   * Common uses:
     + Clearing timers or intervals.
     + Removing event listeners.
     + Canceling network requests.

**10. Hooks (useState, useEffect, useReducer, useMemo, useRef, useCallback)**

**Q:-1 What are React hooks? How do useState() and useEffect() hooks work in functional components?**

**React hooks: revolutionizing functional components**

React Hooks are special functions introduced in React 16.8 that allow you to "hook into" React features like state and lifecycle methods directly from functional components. This eliminates the need for class components, simplifying your codebase and promoting a more functional programming style.

**Why use hooks?**

**Hooks address some limitations of traditional class components, including:**

* **Simplifying code:**Hooks enable cleaner, more intuitive code by avoiding the complexities of this and class component structure.
* **Reusability:**Hooks facilitate the reuse of stateful logic and side effects across different components, which was often cumbersome with class components.
* **Lifecycle Management:**Hooks provide a unified approach to managing component lifecycles, replacing the separate componentDidMount, componentDidUpdate, and componentWillUnmount methods found in class components.

**How useState() and useEffect() work in functional components**

**useState():** managing component state

The useState hook allows you to declare state variables within a functional component. When called, it returns an array containing the current state value and a function to update that state. The component re-renders when the setter function is called.

**Here's an example:**

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

**function Counter() {**

**const [count, setCount] = useState(0); *// Initialize count to 0***

**return (**

**<div>**

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

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

**</div>**

**);**

**}**

**export default Counter;**

**Q:-2 What problems did hooks solve in React development? Why are hooks considered an important addition to React?**

**Problems Before Hooks**

Before React 16.8, developers mostly used class components to manage state and lifecycle. This caused a few problems:

**1. Complexity with State & Lifecycle**

* In class components, logic related to one feature was often split across multiple lifecycle methods.  
  Example:
  + componentDidMount → fetch data.
  + componentDidUpdate → update data.
  + componentWillUnmount → cleanup.
* This made components harder to read and maintain.

**2. Code Reuse Issues**

* Sharing stateful logic (like fetching data, form handling, subscriptions) was difficult.
* Solutions like Higher-Order Components (HOCs) or Render Props worked but often led to:
  + “Wrapper hell” (deeply nested components).
  + Hard-to-follow code.

**3. Class Component Drawbacks**

* Developers had to understand this, binding event handlers, and lifecycle quirks.
* Beginners found classes harder to learn.
* Functional components were simpler, but they couldn’t handle state or side effects.

**How Hooks Solve These Problems**

1. **Simpler State Management**
   * useState lets functional components manage state without classes.
2. **Side Effects in One Place**
   * useEffect handles all lifecycle needs (mount, update, unmount) in a single API.
   * Keeps related logic together instead of scattering it across methods.
3. **Code Reusability**
   * With custom hooks, you can extract and reuse logic easily.
   * Example: useFetch, useAuth, etc.
4. **No More Classes**
   * Hooks let you write everything in functional components.
   * No this binding, fewer bugs, easier for beginners.
5. **Cleaner and Smaller Components**
   * Hooks encourage splitting big components into smaller, reusable logic pieces.

**Why Hooks Are Important**

* Unify functional and class components :- now everything can be functional.
* Make React simpler & more powerful:- less boilerplate, more reusable logic.
* Better developer experience :- easier to read, test, and reuse.
* Future-friendly:- many new React features (like concurrent rendering) work better with hooks.

**Q:-3 What is useReducer ? How we use in react app?**

useReducer is a React Hook that provides an alternative to useState for managing state in functional components, particularly useful for more complex state logic or when state transitions depend on the previous state. It operates on principles similar to Redux, using a reducer function to manage state updates.

How to use useReducer in a React app:

Import useReducer.

import React, { useReducer } from 'react';

* **Define a Reducer Function:** This function takes the current state and an action object as arguments and returns the new state. It typically uses a switch statement to handle different action types.

const reducer = (state, action) => {  
 switch (action.type) {  
 case 'increment':  
 return { count: state.count + 1 };  
 case 'decrement':  
 return { count: state.count - 1 };  
 default:  
 return state;  
 }  
 };

Initialize useReducer in your component.

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

* reducer: The reducer function defined in the previous step.
* initialState: The initial value of your state.
* state: The current state value, similar to the state variable from useState.
* dispatch: A function used to trigger state updates by dispatching actions to the reducer.
* **Dispatch Actions to Update State:** When you need to update the state, call the dispatch function, passing an action object. The action object typically has a type property to specify the action, and can also include a payload for additional data.

JavaScript

<button onClick={() => dispatch({ type: 'increment' })}>Increment</button>  
 <button onClick={() => dispatch({ type: 'decrement' })}>Decrement</button>

This setup centralizes the state update logic within the reducer function, making it easier to manage and test complex state transitions.

**Q:-4 What is the purpose of useCallback & useMemo Hooks?**

The purpose of the useCallback and useMemo hooks in React is to optimize performance by memoizing values and functions, preventing unnecessary re-renders. useMemo caches the result of a function, while useCallback caches the function definition itself.

**useCallback:**

* **Purpose**: Caches a provided function, returning a memoized version. This memoized function is only updated when one of its dependencies changes.
* **Use Case:**Primarily used when passing functions as props to child components. Without useCallback, the function would be recreated on every parent render, potentially causing unnecessary re-renders in the child.

**Example:**

**const MyComponent = React.memo(({ onClick }) => {  
 console.log('Child rendered');  
 return <button onClick={onClick}>Click me</button>;  
 });  
   
 const ParentComponent = () => {  
 const [count, setCount] = React.useState(0);  
   
 const handleClick = React.useCallback(() => {  
 setCount(count + 1);  
 }, [count]); // Dependency on 'count'  
   
 return (  
 <div>  
 <p>Count: {count}</p>  
 <MyComponent onClick={handleClick} />  
 </div>  
 );  
 };**

In this example, handleClick is memoized. If count changes, the handleClick function is updated. Otherwise, the same handleClick function is passed to MyComponent, preventing its re-render unless count actually changes.

**useMemo:**

* **Purpose:**Caches the result of a function, returning the memoized value. The function is only re-executed when one of its dependencies changes.
* **Use Case:**Useful for expensive calculations or operations within a component. By memoizing the result, you avoid recomputing it on every render, especially when the input values haven't changed.

**Example:**

**const MyComponent = ({ data }) => {  
 const processedData = React.useMemo(() => {  
 // Expensive calculation using 'data'  
 return data.map(item => item \* 2);  
 }, [data]); // Dependency on 'data'  
   
 return (  
 <ul>  
 {processedData.map((item, index) => (  
 <li key={index}>{item}</li>  
 ))}  
 </ul>  
 );  
 };**

Here, processedData is memoized. If data remains the same between renders, the mapping calculation is skipped, and the previous result is used.

**In essence:**

* useCallback is for memoizing functions, especially when passing them as props.
* useMemo is for memoizing the result of computations, especially when those computations are expensive.

Both hooks are crucial for optimizing React applications by preventing unnecessary re-renders and expensive calculations, leading to improved performance.

**Q:-5 What’s the Difference between the useCallback & useMemo Hooks?**

The useCallback and useMemo hooks in React are both designed for performance optimization through memoization, but they differ in what they memoize:

* **useMemo:**This hook memoizes a value. It takes a function and a dependency array. The function is executed, and its returned value is cached. This cached value is then returned on subsequent renders unless one of the dependencies in the array changes, in which case the function is re-executed and the new value is cached. useMemo is typically used for expensive computations or data transformations to avoid recalculating them unnecessarily.

**Example:**-

**const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);**

* **useCallback:**This hook memoizes a function. It takes a function and a dependency array. The function itself is cached and returned on subsequent renders, preventing its re-creation, as long as its dependencies remain unchanged. useCallback is particularly useful when passing functions as props to child components, especially those wrapped with React.memo, to prevent unnecessary re-renders of the child component due to a new function reference being passed down.

**Example:-**

**const memoizedCallback = useCallback(() => {  
 doSomething(a, b);  
 }, [a, b]);**

**Q:-6 What is useRef ? How to work in react app?**

useRef is a React Hook that provides a way to create a mutable reference to a value that persists across component renders without causing re-renders when the value changes. It is commonly used for:

* Directly accessing and interacting with DOM elements:

This is the most common use case, allowing you to manipulate elements directly, such as focusing an input field, triggering animations, or measuring element dimensions.

* Storing mutable values that do not trigger re-renders:

Unlike state variables, changes to a ref's .current property do not cause the component to re-render. This is useful for storing values that need to persist but do not directly impact the UI, such as timer IDs or previous state values.

How to use useRef in a React app:

**Import useRef.**

**import React, { useRef } from 'react';**

* Create a ref: Call useRef() inside your functional component. It returns an object with a single property called current, initialized with the value you pass as an argument.

**const myRef = useRef(null);   
 const myValueRef = useRef(0);** Attach the ref to a DOM element (if applicable): Use the ref attribute on the JSX element you want to reference.

**<input type="text" ref={myRef} />**

* Access or modify the ref's current property: You can access the referenced value or DOM element through myRef.current.

***// To focus an input field:*  
 myRef.current.focus();  
  
 *// To update a persistent value:*  
 myValueRef.current = myValueRef.current + 1;**

**11. Routing in React (React Router)**

**Q:-1: What is React Router? How does it handle routing in single-page applications?**

React Router is a standard library for routing in React applications. It enables navigation between different views or components in a React app, allowing you to build a Single Page Application (SPA) with multiple "pages."

In a normal multi-page website, every time a user clicks a link, the browser makes a new request to the server and reloads the entire page. But in an SPA with React Router:

* Only one HTML page (index.html) is loaded initially.
* When the user navigates, React Router changes the URL in the browser without refreshing the whole page.
* It dynamically renders the required React component for that route.

This creates a smooth user experience with faster navigation, since only the content changes while the page structure remains the same.

**Q:-2: Explain the difference between BrowserRouter, Route, Link, and Switch components in React Router.**

**1. BrowserRouter**

* It is the parent component that enables routing in a React app.
* It uses the HTML5 history API (pushState, replaceState) to keep the UI in sync with the URL.
* Must wrap around all components that use routing.

**<BrowserRouter>**

**<App />**

**</BrowserRouter>**

**2. Route**

* Defines the path-to-component mapping.
* When the URL matches the path, the corresponding component is rendered.

**<Route path="/about" element={<About />} />**

**3. Link**

* Used for navigation between routes.
* Works like an <a> tag but without reloading the page.
* Updates the URL and lets React Router render the right component.

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

**4. Switch (React Router v5) / Routes (React Router v6)**

* Ensures that only one matching route is rendered.
* Without Switch, multiple routes may match and render simultaneously.
* In React Router v6, Switch is replaced with Routes.

**<Switch>**

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

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

**</Switch>**

**12. React – JSON-server and Firebase Real Time Database**

**Q:-1: What do you mean by RESTful web services?**

* REST stands for Representational State Transfer.
* RESTful web services follow REST architecture principles.
* They allow communication between client and server using standard HTTP methods:
  + GET → Retrieve data
  + POST → Create new data
  + PUT/PATCH → Update existing data
  + DELETE → Remove data
* Data is usually exchanged in JSON format.  
   Example: GET /users → returns all users

**Q:-2: What is JSON-Server? How we use in React?**

* JSON-Server is a simple tool to create a fake REST API using a JSON file.
* It helps developers mock APIs during development without needing a backend.
* **How to use it in React:**

Step 1: Install JSON-Server

Step 2: Create a db.json file

Step 3: Run JSON-Server

**Q:-3: How do you fetch data from a JSON-Server API in React? Explain fetch() or axios().**

* In React, we use **fetch()** or **axios()** to call APIs.
* Example using fetch():

**Q:-4: What is Firebase? What features does Firebase offer?**

* **Firebase** is a **Backend-as-a-Service (BaaS)** platform by Google.
* It helps developers build apps quickly without managing servers.
* **Features of Firebase:**
  1. **Authentication** → Google, Facebook, Email/Password login
  2. **Realtime Database** → NoSQL database with real-time updates
  3. **Firestore** → Cloud-hosted NoSQL database
  4. **Hosting** → Deploy web apps easily
  5. **Cloud Functions** → Run serverless backend code
  6. **Push Notifications** → Send messages via Firebase Cloud Messaging
  7. **Analytics & Crashlytics** → Track user behavior and errors

**Q:-5: Importance of handling errors and loading states when working with APIs in React**

* When fetching data, the app goes through **3 main states**:
  1. **Loading** → While waiting for the response
  2. **Success** → Data received
  3. **Error** → API failed (network issue, server error, wrong URL)
* Why important?
  1. Prevents showing blank screens
  2. Improves **user experience**
  3. Provides feedback (e.g., “Loading…” or “Something went wrong”)

**13. Context API**

**Q:-1: What is the Context API in React? How is it used to manage global state across multiple components?**

Context API is a React feature that allows you to share state or data globally across multiple components without the need to pass props down through every intermediate component (also called “prop drilling”).

It is especially useful when multiple components in different parts of your component tree need access to the same state, for example:

* Theme settings (light/dark mode)
* Authentication status (logged in or not)
* User preferences
* Language settings (i18n)

How it manages global state:

1. Create a Context: You define a context object that will hold the shared state.
2. Provide the Context: Wrap the components that need access in a <Provider> component and pass the shared state as its value.
3. Consume the Context: Any nested component can access the context using useContext() without receiving it as a prop.

**Q:-2: Explain how createContext() and useContext() are used in React for sharing state**

1. **createContext()**
   * It **creates a context object**.
   * The object contains:
     + **Provider**: Component to wrap around children to share data.
     + **Consumer** (optional, mostly used in class components).
2. **useContext()**

* A **React hook** that allows a functional component to **consume the nearest context value** from the Provider.
* Simplifies access to global state without props.

**14. State Management (Redux, Redux-Toolkit or Recoil)**

**Q:-1: What is Redux, and why is it used in React applications? Explain the core concepts of actions, reducers, and the store.**

Redux is a predictable state management library for JavaScript applications, commonly used with React. It helps manage application state in a single global store, making it easier to track, debug, and maintain. Redux is particularly useful in large-scale applications where multiple components need access to shared state.

**Why Redux is used in React applications:**

* Centralizes state in a single store, avoiding prop drilling.
* Makes state changes predictable and traceable.
* Improves maintainability of complex applications.
* Enables powerful debugging with tools like Redux DevTools.

**Core Concepts:**

1. **Store**
   * The single source of truth for application state.
   * Holds the complete state object of the app.
   * Allows components to subscribe to state changes.
2. **Actions**
   * Plain JavaScript objects that describe what happened.
   * Must have a type property, which describes the action.
3. **Reducers**
   * Pure functions that take the current state and an action and return a new state.
   * Do not mutate the state directly.

**Flow in Redux:**

1. Component dispatches an action.
2. Reducer receives current state + action → returns new state.
3. Store updates and notifies subscribed components.
4. Components re-render with the new state.

**Question 2: How does Recoil simplify state management in React compared to Redux?**

Recoil is a state management library developed by Facebook for React that provides a simpler and more React-friendly way to manage state**.**

**Benefits and simplifications compared to Redux:**

1. **Less boilerplate**
   * Unlike Redux, Recoil does not require actions, reducers, or constants. You define atoms (state units) and selectors (derived state).
2. **Direct component subscription**
   * Components subscribe directly to atoms, making state updates automatic and simple.
   * No need to connect components manually like Redux’s connect().
3. **Scoped state**
   * Atoms can be local to certain parts of the component tree, reducing global clutter.
   * Useful for modular components**.**
4. **Derived state with selectors**
   * Computed or filtered values can be obtained using selectors without creating additional reducers.
5. **Better React integration**

* Recoil uses React hooks (useRecoilState, useRecoilValue, useSetRecoilState) which feels natural in functional components.