Lists , Hooks

• Lists and Keys

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

ANS 1 : In React, rendering a list of items is typically done using the JavaScript `.map()` function. You take an array of data and map over it to create a new array of React elements, usually returning a component or JSX for each item. For example, if you have an array of names, you can use `names.map(name => <li>{name}</li>)` to render each name inside a `<li>` element. This approach makes React components dynamic and helps display data-driven content efficiently.

Using **keys** when rendering lists is crucial because keys help React identify which items have changed, been added, or removed. When a component re-renders, React uses these keys to match old and new elements, optimizing the update process for better performance. Without keys, React might re-render the entire list unnecessarily, causing performance issues and potential bugs, especially when items are reordered or removed.

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

ANS 2: Keys in React are special string attributes used to uniquely identify elements in a list. They are used internally by React to efficiently manage and update the user interface when changes occur in a collection of elements. Typically, keys are assigned using a unique property like an `id` or index (though using an index is not recommended when list order can change).

If you do not provide a unique key, React cannot properly track which items have been modified, moved, or deleted. This can lead to unexpected behavior, such as components reusing the wrong data or losing their internal state during updates. Additionally, React will display a warning in the console, reminding developers to provide unique keys to improve rendering efficiency and maintain predictable UI behavior.

Lists , Hooks

- Hooks (useState, useEffect, useReducer, useMemo, useRef, useCallback)

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

  ANS 1: React Hooks are special functions that allow developers to use state and other React features in functional components without needing class components. The `useState()` hook is used to add state to a functional component—it returns a state variable and a function to update it. For example, `const [count, setCount] = useState(0)` initializes a counter. The `useEffect()` hook, on the other hand, lets you perform side effects like fetching data, setting up subscriptions, or manually changing the DOM. It runs after the component renders, and you can control when it runs by passing dependencies in an array.

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

ANS 2 : Before hooks, developers relied on class components to manage state and lifecycle methods, which often led to complex, hard-to-maintain code. Hooks solved this by enabling functional components to handle state and side effects more cleanly and with less boilerplate. They also improved code reusability through custom hooks, making logic sharing between components easier. Hooks are considered an important addition to React because they simplified component design, reduced code duplication, and made it easier to understand how data and effects flow within an application.

Question 3: What is useReducer? How do we use it in a React app?

ANS 3: The `useReducer()` hook is used to manage complex state logic in React components, especially when the state depends on multiple sub-values or complex actions. It works similarly to reducers in Redux—taking a reducer function and an initial state, then returning the current state and a dispatch function. For example, `const [state, dispatch] = useReducer(reducer, initialState)` allows you to update the state by dispatching actions like `dispatch({ type: 'INCREMENT' })`. It's particularly useful for larger apps where managing multiple `useState()` hooks becomes cumbersome.

Question 4: What is the purpose of useCallback and useMemo Hooks?

ANS 4: The `useCallback()` and `useMemo()` hooks are performance optimization tools in React. The `useCallback()` hook is used to memoize functions so that they are not re-created on every render unless their dependencies change. This helps prevent unnecessary re-renders of child components that rely on those functions. The `useMemo()` hook, on the other hand, memoizes the result of a computation or expression, caching expensive calculations and recomputing only when dependencies change. Both hooks improve performance by reducing redundant re-renders and recalculations.

Question 5: What’s the difference between the useCallback and useMemo Hooks?

ANS 5: The main difference between `useCallback()` and `useMemo()` lies in what they return. `useCallback()` returns a memoized function, while `useMemo()` returns a memoized value. You use `useCallback()` when you want to prevent function recreation on every render, which is useful when passing callback props to child components. In contrast, you use `useMemo()` when you want to optimize heavy computations or derived data, ensuring the calculation only runs when dependencies change. Essentially, `useCallback(fn, deps)` is equivalent to `useMemo(() => fn, deps)`.

Question 6: What is useRef? How does it work in a React app?

ANS 6: The `useRef()` hook is used to create a mutable reference object that persists across renders without causing re-renders when its value changes. It's commonly used to directly access or manipulate DOM elements—for example, focusing an input field using `ref.current.focus()`. It can also store previous values or mutable variables that don't trigger a re-render when updated. In a React app, you create a ref using `const inputRef = useRef(null)` and attach it to a JSX element with `ref={inputRef}`, allowing you to interact with that element programmatically.