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

Answer:

React Hooks are special functions that let you "hook into" React state and lifecycle features from functional components. They were introduced in React 16.8 to allow functional components to manage state and side effects, which were previously only possible in class components.

useState() Hook:

Purpose: Adds state to functional components.

How it works: It returns an array with two elements: the current state value and a function to update it. It takes the initial state as an argument.

Example:

import { useState } from 'react';

function Counter() {

const [count, setCount] = useState(0); // Initial state is 0

return (

<div>

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

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

Increment

</button>

</div>

);

}

useEffect() Hook:

Purpose: Handles side effects in functional components (equivalent to componentDidMount, componentDidUpdate, and componentWillUnmount combined).

How it works: It takes a function (the effect) and an optional dependency array. The effect runs after render.

Examples:

import { useState, useEffect } from 'react';

function UserProfile({ userId }) {

const [user, setUser] = useState(null);

// Effect with dependency - runs when userId changes

useEffect(() => {

fetch(`/api/users/${userId}`)

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

.then(userData => setUser(userData));

}, [userId]); // Dependency array

// Effect without dependencies - runs after every render

useEffect(() => {

document.title = `User: ${user?.name || 'Loading...'}`;

});

// Cleanup effect - runs when component unmounts or before re-running

useEffect(() => {

const timer = setInterval(() => {

console.log('Timer tick');

}, 1000);

return () => clearInterval(timer); // Cleanup function

}, []);

return <div>{user?.name}</div>;

}

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

Answer:

Problems Hooks Solved:

Wrapper Hell (HOC Hell): Higher-Order Components often led to deeply nested component trees that were hard to understand and debug.

Huge Components: Class components often became large and complex, with unrelated logic scattered across different lifecycle methods.

Confusing this Keyword: Class components required understanding JavaScript this binding, which was confusing for many developers.

Logic Reusability: It was difficult to reuse stateful logic between components without complex patterns like Render Props or HOCs.

Why Hooks are Important:

Simpler Code: Functional components with hooks are more concise and easier to read.

Better Logic Organization: Related code can be grouped together in custom hooks instead of being split across different lifecycle methods.

Reusable Stateful Logic: Custom hooks allow you to extract and reuse stateful logic across multiple components.

No More this: Functional components eliminate the confusion around the this keyword.

Progressive Adoption: Hooks work alongside existing code, allowing gradual migration.

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

Answer:

useReducer is a hook for state management that is an alternative to useState. It's preferable when you have complex state logic that involves multiple sub-values or when the next state depends on the previous one.

How it works: It follows the Redux pattern: (state, action) => newState

Usage:

import { useReducer } from 'react';

// Reducer function

function todoReducer(state, action) {

switch (action.type) {

case 'ADD\_TODO':

return [...state, { id: Date.now(), text: action.text, completed: false }];

case 'TOGGLE\_TODO':

return state.map(todo =>

todo.id === action.id ? { ...todo, completed: !todo.completed } : todo

);

case 'DELETE\_TODO':

return state.filter(todo => todo.id !== action.id);

default:

return state;

}

}

function TodoApp() {

const [todos, dispatch] = useReducer(todoReducer, []); // Initial state: empty array

const addTodo = (text) => {

dispatch({ type: 'ADD\_TODO', text });

};

return (

<div>

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

{todos.map(todo => (

<div key={todo.id}>

<span>{todo.text}</span>

<button onClick={() => dispatch({ type: 'TOGGLE\_TODO', id: todo.id })}>

Toggle

</button>

</div>

))}

</div>

);

}

**Question 4 & 5: What is the purpose of useCallback & useMemo Hooks? What's the Difference?**

Answer:

Both are performance optimization hooks that prevent unnecessary re-renders and computations.

useMemo - Memoizes VALUES:

Purpose: Returns a memoized value to avoid expensive calculations on every render.

Use case: When you have expensive computations that don't need to run on every render.

useCallback - Memoizes FUNCTIONS:

Purpose: Returns a memoized function to prevent unnecessary re-renders of child components.

Use case: When passing functions as props to optimized child components that rely on reference equality.

Examples and Difference:

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

function ExpensiveComponent({ items, onItemClick }) {

// useMemo: memoizes the computed value

const expensiveValue = useMemo(() => {

return items.reduce((sum, item) => sum + item.value, 0);

}, [items]); // Only recalculate when 'items' changes

// useCallback: memoizes the function itself

const handleClick = useCallback((itemId) => {

onItemClick(itemId);

}, [onItemClick]); // Only recreate when 'onItemClick' changes

return (

<div>

<p>Total: {expensiveValue}</p>

{items.map(item => (

<MemoizedChild

key={item.id}

item={item}

onClick={handleClick} // This function reference stays stable

/>

))}

</div>

);

}

// React.memo only re-renders if props change

const MemoizedChild = React.memo(({ item, onClick }) => {

return <button onClick={() => onClick(item.id)}>{item.name}</button>;

});

Key Difference:

useMemo remembers the result of a function call: useMemo(() => computeValue, deps)

useCallback remembers the function itself: useCallback(() => { doSomething() }, deps)

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

Answer:

useRef returns a mutable ref object whose .current property is initialized to the passed argument. The returned object will persist for the full lifetime of the component.

Common Use Cases:

Accessing DOM Elements:

function TextInput() {

const inputRef = useRef(null);

const focusInput = () => {

inputRef.current.focus(); // Directly access DOM element

};

return (

<div>

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

<button onClick={focusInput}>Focus Input</button>

</div>

);

}

Storing Mutable Values Without Re-renders:

function Timer() {

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

const intervalRef = useRef(); // Doesn't trigger re-render when changed

const startTimer = () => {

intervalRef.current = setInterval(() => {

setCount(prevCount => prevCount + 1);

}, 1000);

};

const stopTimer = () => {

clearInterval(intervalRef.current);

};

return (

<div>

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

<button onClick={startTimer}>Start</button>

<button onClick={stopTimer}>Stop</button>

</div>

);

}

Keeping Track of Previous State:

function Counter() {

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

const prevCountRef = useRef();

useEffect(() => {

prevCountRef.current = count; // Update ref after render

});

const prevCount = prevCountRef.current;

return (

<div>

<p>Now: {count}, Before: {prevCount}</p>

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

</div>

);

}