1. Difference Between an Empty Dependency Array ([]) and Omitting the Dependency Array in useEffect

The behavior of useEffect changes depending on the dependency array:

Dependency Array	Execution Behavior
useEffect(() => {...}, [])	Runs **only once** when the component mounts.
useEffect(() => {...}) *(No dependency array)*	Runs **on every render**, causing potential performance issues.

 Example of useEffect with [] (Runs Once on Mount)

useEffect(() => { console.log('Component mounted'); }, []); // Runs only once

 Example of useEffect Without [] (Runs on Every Render)

```
useEffect(() => { console.log('Runs on every render'); });
```

When to Use?

- **Use []** when you only need the effect to run once (e.g., fetching initial data).
- **Omit []** only if you intentionally want the effect to run on every render (rare case).

---

2. Handling Cleanup in useEffect and Why It Is Necessary

What is Cleanup in useEffect?

When an effect creates side effects (like timers, event listeners, or subscriptions), React provides a cleanup function to **remove them when the component unmounts** or when dependencies change.

Why is Cleanup Important?

- **Prevents Memory Leaks** (e.g., removing event listeners).
- **Ensures Correct Behavior** when dependencies update.
- **Avoids Duplicate Effects** when re-rendering.

Example of Cleanup in useEffect

useEffect(() => { const handleResize = () => console.log('Resized'); window.addEventListener('resize', handleResize); return () => { window.removeEventListener('resize', handleResize); // Cleanup }; }, []);

Here, return () => {...} **removes the event listener when the component unmounts**.

---

3. Common Pitfalls or Mistakes to Avoid When Using useEffect

⚠️ Mistake 1: Forgetting the Dependency Array ([])

✅ Fix: Always include dependencies to control re-runs.

useEffect(() => { console.log('Runs unnecessarily on every render'); }); // Incorrect useEffect(() => { console.log('Runs only once'); }, []); // Correct

⚠️ Mistake 2: Incorrect Dependency Array Handling

✅ Fix: Ensure **all variables used inside useEffect are in the dependency array**.

useEffect(() => { console.log(user); // Using 'user', but not adding it to dependencies }, []); // Incorrect useEffect(() => { console.log(user); }, [user]); // Correct

⚠️ Mistake 3: Not Cleaning Up Side Effects

✅ Fix: Always clean up timers, event listeners, or subscriptions.

useEffect(() => { const interval = setInterval(() => { console.log('Interval running'); }, 1000); return () => clearInterval(interval); // Cleanup }, []);

⚠️ Mistake 4: Infinite Re-render Loops

✅ Fix: Ensure useEffect doesn't update the state variable inside itself **without proper condition checks**.

useEffect(() => { setCount(count + 1); // Causes infinite loop }, [count]); // Incorrect useEffect(() => { if (count < 5) setCount(count + 1); // Prevents infinite loop }, [count]); // Correct

---

4. Conditionally Running Effects in useEffect

Sometimes, you may want useEffect to run **only under specific conditions**.

✅ Using an if Condition Inside useEffect

useEffect(() => { if (userLoggedIn) { console.log('User is logged in'); } }, [userLoggedIn]); // Runs only when `userLoggedIn` changes

⚠️ Using a State Variable to Control Execution

const [shouldFetch, setShouldFetch] = useState(false); useEffect(() => { if (!shouldFetch) return; // Effect runs only when shouldFetch is true fetchData(); }, [shouldFetch]); <button onClick={() => setShouldFetch(true)}>Fetch Data</button>
