React useEffect Hook - Complete Notes

1️⃣ What is useEffect?

Definition: `useEffect` is a React hook that allows functional components to perform side effects.

Side effects include: - API calls - Timers / intervals - Event listeners / subscriptions - DOM updates outside JSX - Local storage / cookies

Syntax:

```
useEffect(() => {
  // effect logic
  return () => {
    // optional cleanup
  };
}, [dependencies]);
```

- Optional cleanup function is returned. - Effects run **after render**, never during render.

2️⃣ Understanding Side Effects

Definition: Any operation that affects something outside the function's return value or interacts with the external world.

Why separate side effects from render: - Render must be **pure** and deterministic (input → output) - Side effects can cause: - Multiple API calls - Duplicate timers - Memory leaks - Inconsistent UI

Solution: `useEffect` runs after render (commit phase)

Examples:

```
// Updating DOM outside JSX
document.title = `Count: ${count}`;

// Fetching data
fetch("/api/data").then(res => res.json());

// Event listeners
window.addEventListener("resize", handleResize);

// Timers
setInterval(() => setCount(prev => prev + 1), 1000);
```

3️⃣ Dependency Array

Pattern	Behavior
No array	Runs after every render
[]	Runs once on mount, cleanup runs on unmount
[dep1, dep2]	Runs when any dependency changes, cleanup runs before next effect

Example:

```
useEffect(() => {
  console.log("Effect runs");
  return () => console.log("Cleanup runs");
}, [count]);
```

- First render → effect runs - Next render where `count` changes → cleanup runs → effect runs again

---

4️⃣ Cleanup Function

Runs: - Before the next effect (when dependencies change) - On component unmount

Importance: - Prevents memory leaks - Avoids duplicate subscriptions, timers, or intervals - Keeps state consistent

Example:

```
useEffect(() => {
  const interval = setInterval(() => setCount(prev => prev + 1), 1000);
  return () => clearInterval(interval);
}, []);
```

5️⃣ Lifecycle Simulation

Dependency Array	React Class Equivalent
[]	componentDidMount + componentWillUnmount (cleanup)
[dep]	componentDidUpdate (specific prop/state) + cleanup
No deps	componentDidMount + componentDidUpdate on all renders

---

6️⃣ Functional Updates and Stale Closures

`useEffect` captures **render-scoped values**, which may result in stale state.

Problem Example:

```
useEffect(() => {
  const interval = setInterval(() => {
    console.log(count); // may log stale count
  }, 1000);
}, []); // count captured from first render
```

✅ **Fix with functional update:**

```
useEffect(() => {
  const interval = setInterval(() => {
    setCount(prev => prev + 1); // always latest value
  }, 1000);
  return () => clearInterval(interval);
}, []);
```

7️⃣ Common Gotchas

- **Missing dependencies:**

```
useEffect(() => console.log(count), []); // ❌ 'count' missing → stale
```

- **Infinite loops:**

```
useEffect(() => setCount(count + 1), [count]); // ❌ endless re-renders
```

- **Over-fetching:** due to missing dependency array
- **Stale closures in async functions** → use functional updates or refs

---

8️⃣ Advanced Patterns

1️⃣ Split effects by concern:

```
useEffect(() => { /* fetch data */ }, []);
useEffect(() => { /* update title */ }, [count]);
```

2️⃣ Custom hooks:

```
function useWindowWidth() {
  const [width, setWidth] = useState(window.innerWidth);
  useEffect(() => {
    const handleResize = () => setWidth(window.innerWidth);
    window.addEventListener("resize", handleResize);
    return () => window.removeEventListener("resize", handleResize);
  }, []);
  return width;
}
```

3️⃣ Stable refs for dynamic intervals / callbacks:

```
const savedCallback = useRef();
useEffect(() => { savedCallback.current = callback; }, [callback]);
useEffect(() => {
  const tick = () => savedCallback.current();
  const id = setInterval(tick, delay);
  return () => clearInterval(id);
}, [delay]);
```

9️⃣ Re-render & useEffect Flow

1. Component renders → DOM is updated (commit phase)
2. `useEffect` runs → side effects executed
3. If effect updates state → React schedules another render
4. Cleanup runs before next effect or on unmount

Visual Model:

```
Render → Commit DOM → useEffect runs → (state update?) → next render →
cleanup → useEffect runs again
```

1️⃣0️⃣ Production Use Cases

- **API Calls:** fetch data on mount
- **Timers / Intervals:** countdowns, clocks, counters

- **Event Listeners:** resize, scroll, keypress
- **Subscriptions:** WebSocket, Redux, Context
- **DOM Updates:** document.title, focus, animation triggers
- **Local storage / caching:** save/load data

1️⃣1️⃣ Senior-Level Best Practices

- Always use **dependency array** properly → avoid stale closures
- Split effects by concern → maintainable code
- Use **cleanup** → prevent memory leaks, duplicates
- **Functional updates** for state inside effects → latest value
- Avoid infinite loops → careful with state updates inside effect
- Use **refs** for stable callbacks → dynamic intervals or timers
- **Custom hooks** → encapsulate reusable side effect logic

1️⃣2️⃣ Summary Table

Pattern	Runs When	Cleanup
useEffect(() => {}, [])	Mount only	On unmount
useEffect(() => {}, [dep])	Mount + dep changes	Before next effect or unmount
useEffect(() => {})	Every render	Before next render

✅**Senior-Level Mental Model:** - useEffect = side effects **after render**, never during render - Cleanup ensures no overlapping subscriptions/timers - Dependency array = controls when effect runs - Functional updates = avoid stale closures - Proper effect management = predictable, efficient, memory-safe UI