React Hooks Interview Questions & Answers

PDF Study Pack: freshers + experienced, with scenarios, testing, and a one-screen cheat sheet.

Homepage: https://interviewquestions.guru/

Use this as a printable companion to your React hooks interview prep.

What’s inside

- Core hooks: useState, useEffect, useRef, useContext, useMemo, useCallback, useReducer
- Advanced hooks: useLayoutEffect, useId, useTransition, useDeferredValue, useSyncExternalStore
- Top pitfalls: dependency array bugs, stale closures, infinite loops, race conditions
- Includes: interview scripts, debug checklist, and testing questions

Tip: Print (Ctrl/Cmd+P) and save as PDF to share. Keep code blocks for quick recall.

Table of Contents

1) React Hooks Explained (Quick Overview)

Hooks let function components use React features like state and effects. Mental model: React renders (calculate UI), commits (update DOM), then runs effects (sync with external systems).

Rules of Hooks

- Call hooks only at the top level (no conditions or loops).
- Call hooks only from React functions (components or custom hooks).

30-second interview script

“Hooks like useState and useEffect let function components manage state and side effects. Rendering stays pure; effects run after commit to sync with external systems like fetch, timers, and subscriptions. Dependency arrays control when effects re-run and prevent stale closures.”

2-minute interview script

“React calls my component to render - that’s computing the UI. After React commits DOM changes, it runs effects for external synchronization. The dependency array is the list of reactive values the effect reads. If a value can change and is used inside, it’s a dependency. The exhaustive-deps lint rule catches missing dependencies that cause stale closures. I use useMemo for expensive derived values, useCallback to stabilize function props for memoized children, and useRef for mutable values that shouldn’t re-render.”

2) React Hooks Cheat Sheet (One Screen)

- **Functional state update**: setCount(c => c + 1) (avoids stale state)
- **Effect cleanup**: useEffect(() => { subscribe(); return unsubscribe; }, [deps])
- **Dependency array**: list reactive values used inside the callback
- **Stale closure fixes**: functional updates, correct deps, or refs for latest values
- **Memoization cues**: useMemo for expensive values, useCallback for stable function identity
- **Refs**: DOM access + instance variables without re-render
- **Fetch abort**: AbortController in cleanup to avoid race conditions
- **React.memo**: skips child render when props are referentially equal

Fetch abort pattern

```
useEffect(() => {
  const c = new AbortController();
  fetch(url, { signal: c.signal })
    .catch(e => e.name === "AbortError" ? null : Promise.reject(e));
  return () => c.abort();
}, [url]);
```

3) React Hooks Interview Questions (Freshers) - 25

Q1. What are hooks?

Functions that let function components use state, effects, refs, context, and more.

Q2. Why were hooks introduced?

To reuse stateful logic without classes and simplify component composition.

Q3. Rules of Hooks?

Call hooks at top level and only from React functions.

Q4. useState returns what?

[value, setValue]. setValue triggers re-render.

Q5. How to update state based on previous state?

Use functional updates: setCount(c => c + 1).

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

function incThrice() {
  setCount(c => c + 1);
  setCount(c => c + 1);
  setCount(c => c + 1);
}
```

Q6. What is useEffect for?

Sync with external systems after commit (fetch, timers, subscriptions).

```
useEffect(() => {
  document.title = `Count: ${count}`;
}, [count]);
```

Q7. What does [] mean in useEffect?

Run on mount; cleanup on unmount.

Q8. What if deps are omitted?

Effect runs after every render.

Q9. What is a cleanup function?

Return function from effect to unsubscribe/abort/clear timers.

```
useEffect(() => {
  const id = setInterval(tick, 1000);
  return () => clearInterval(id);
}, []);
```

Q10. What is useRef?

Mutable container that persists across renders without re-render.

Q11. useRef vs useState?

State triggers re-render; refs do not.

Q12. What is useContext?

Reads nearest Provider value; avoids prop drilling.

Q13. Controlled vs uncontrolled inputs?

Controlled: React state; Uncontrolled: DOM state via refs.

```
function Form() {
  const [name, setName] = useState("");
  const emailRef = useRef(null);

  function submit(e) {
    e.preventDefault();
    alert({ name, email: emailRef.current.value });
  }

  return (
    <form onSubmit={submit}>
      <input value={name} onChange={e => setName(e.target.value)} />
      <input ref={emailRef} />
      <button>Submit</button>
    </form>
  );
}
```

Q14. What is useMemo?

Memoizes a computed value.

Q15. What is useCallback?

Memoizes a function identity.

Q16. When should you avoid memoization?

When it adds complexity without measurable benefit.

Q17. What is a custom hook?

A function starting with use that composes hooks to reuse logic.

Q18. Can you call hooks in conditions?

No, must be consistent order every render.

Q19. Can you set state during render?

Avoid; can cause infinite loops.

Q20. Common effect bug?

Missing deps causing stale closures.

Q21. What is dependency array?

Values that effect/memo/callback depends on.

Q22. What is stale closure?

Callback uses captured old values due to missing deps.

Q23. How to avoid memory leaks?

Cleanup subscriptions and timers in effects.

Q24. What is useReducer good for?

Complex state transitions with predictable updates.

Q25. What is React.memo?

Memoizes a component render based on props equality.

4) React Hooks Interview Questions (Experienced) - 40

Q1. Explain render -> commit -> effects.

Render calculates UI; commit updates DOM; effects sync with external systems after commit.

Q2. What does exhaustive-deps enforce?

Dependency arrays include all reactive values used inside callbacks to prevent stale closures.

Q3. Why is disabling exhaustive-deps risky?

It can hide correctness bugs; restructure code instead.

Q4. What is stale closure? Give an example.

Async callbacks can capture old state; fix with deps, refs, or functional updates.

```
// Broken interval (stale closure)
useEffect(() => {
  const id = setInterval(() => setCount(count + 1), 1000);
  return () => clearInterval(id);
}, []);

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

Q5. How do infinite loops happen in effects?

Effect sets state that changes dependencies; compute derived data in render/useMemo.

```
// Broken: derived state loop
useEffect(() => {
  setFiltered(items.filter(x => x.active));
}, [items, filtered]);

// Fixed: derive in render
const filtered = useMemo(
  () => items.filter(x => x.active),
  [items]
);
```

Q6. useMemo vs useCallback?

useMemo caches values; useCallback caches function identity.

```
const total = useMemo(() => calc(items), [items]);
const onPick = useCallback((id) => select(id), [select]);
```

Q7. When does React.memo help?

When child render is expensive and props are stable by reference.

Q8. When is memoization premature?

When it increases complexity without measurable benefit.

Q9. How to stabilize object dependencies?

Memoize objects/arrays with useMemo.

Q10. How to stabilize function dependencies?

Use useCallback or move logic inside effect.

Q11. useRef for instance variables?

Store timers, previous values, and flags without re-render.

Q12. useLayoutEffect vs useEffect?

Layout effect runs before paint for measurements; effect runs after paint.

Q13. How to handle race conditions in fetch?

Abort previous requests or track request IDs.

```
useEffect(() => {
  const c = new AbortController();
  fetch(url, { signal: c.signal })
    .then(r => r.json())
    .then(setData)
    .catch(e => e.name === "AbortError" ? null : setError(e));
  return () => c.abort();
}, [url]);
```

Q14. How to avoid setState after unmount?

Abort fetch in cleanup and ignore AbortError.

Q15. useReducer vs useState tradeoff?

Reducer is good for complex transitions; state for simple values.

Q16. How to split context to reduce renders?

Split state/actions contexts and memoize provider values.

Q17. How do you design custom hooks?

Expose stable API; ensure cleanup; accept primitives or memoize options.

Q18. Implement useDebounce.

Use a timer in effect; clearTimeout in cleanup.

Q19. Implement useFetch with AbortController.

Start fetch in effect; abort on cleanup; handle errors.

Q20. What is referential equality?

Same reference (===). Critical for memoization and React.memo.

Q21. How to avoid derived state anti-pattern?

Compute from props/state during render or useMemo.

Q22. What belongs in deps array?

All reactive values used inside: props, state, and render-created values.

Q23. How to debug effect runs too often?

Look for unstable deps and Strict Mode dev behavior.

Q24. How to prevent subscription leaks?

Return cleanup unsubscribe.

Q25. How to test effects?

Test via components; mock external APIs; assert cleanup on unmount.

Q26. How to test timers?

Use fake timers; advance time; assert state/UI changes.

Q27. How to test async + cleanup?

Unmount and ensure abort/unsubscribe; ensure no late updates.

Q28. When to use useTransition?

To keep urgent input responsive while deferring heavy updates.

```
const [isPending, startTransition] = useTransition();

function onChange(e) {
  const next = e.target.value;
  setQuery(next); // urgent
  startTransition(() => setFiltered(expensiveFilter(next))); // non-urgent
}
```

Q29. When to use useDeferredValue?

To defer expensive derived rendering based on a fast-changing input.

Q30. What does useSyncExternalStore do?

Subscribes safely to external stores for concurrent rendering.

```
const value = useSyncExternalStore(store.subscribe, store.getSnapshot);
```

Q31. Why not use useId for list keys?

Keys must be per-item identity; useId is per-component instance.

Q32. How do you avoid callback identity rerenders?

useCallback + memoized child, or move handler into child.

Q33. Error handling pattern in effects?

Catch errors; ignore AbortError; set error state.

Q34. Cleanup best practices?

Abort, unsubscribe, clear timers, and reset external state.

Q35. Custom hook pitfalls?

Unstable deps causing re-subscribe; missing cleanup; leaking abstraction.

Q36. How to avoid over-rendering with context?

Split contexts; memoize provider values; selectors.

Q37. Effect vs event handler?

Handlers respond to user actions; effects sync after commit.

Q38. How to fix effect uses inline object?

Memoize the object or move inside effect.

Q39. What is controlled vs uncontrolled tradeoff?

Controlled easier validation; uncontrolled fewer renders but harder live validation.

5) Scenario-Based Debug Questions - 15

Format: buggy code -> fixed code -> quick explanation.

Scenario 1. Missing deps causes stale props

Buggy

```
// Bug
useEffect(() => { console.log(userId); }, []);
```

Fixed

```
// Fix
useEffect(() => { console.log(userId); }, [userId]);
```

Why: Include reactive values in deps.

Scenario 2. Effect loop due to derived state

Buggy

```
// Bug
useEffect(() => setFiltered(filter(items)), [items, filtered]);
```

Fixed

```
// Fix
const filtered = useMemo(() => filter(items), [items]);
```

Why: Derive in render or useMemo.

Scenario 3. Race condition in fetch

Buggy

```
// Bug
useEffect(() => { fetch(url).then(r => r.json()).then(setData); }, [url]);
```

Fixed

```
// Fix (abort)
useEffect(() => {
  const c = new AbortController();
  fetch(url, { signal: c.signal }).then(r => r.json()).then(setData);
  return () => c.abort();
}, [url]);
```

Why: Abort previous requests in cleanup.

Scenario 4. Inline object breaks deps

Buggy

```
// Bug
const opts = { headers: { a: 1 } };
useEffect(() => doThing(opts), [opts]);
```

Fixed

```
// Fix
const opts = useMemo(() => ({ headers: { a: 1 } }), []);
```

```
useEffect(() => doThing(opts), [opts]);
```

Why: Memoize objects/arrays used as deps.

Scenario 5. Child rerenders due to callback identity

Buggy

```
// Bug
<Child onClick={() => setOpen(true)} />
```

Fixed

```
// Fix
const onClick = useCallback(() => setOpen(true), []);
<Child onClick={onClick} />
```

Why: Stabilize callback props when child is memoized.

Scenario 6. Ref used for UI state (no re-render)

Buggy

```
// Bug
const count = useRef(0);
return <button onClick={() => count.current++}>{count.current}</button>;
```

Fixed

```
// Fix
const [count, setCount] = useState(0);
return <button onClick={() => setCount(c => c + 1)}>{count}</button>;
```

Why: Use state for UI; refs for mutable non-UI.

Scenario 7. Flicker when measuring layout

Buggy

```
// Bug
useEffect(() => setW(ref.current.getBoundingClientRect().width), []);
```

Fixed

```
// Fix
useLayoutEffect(() => setW(ref.current.getBoundingClientRect().width), []);
```

Why: Measure before paint with layout effect.

Scenario 8. Subscription leak (no cleanup)

Buggy

```
// Bug
useEffect(() => {
  window.addEventListener("scroll", onScroll);
}, []);
```

Fixed

```
// Fix
useEffect(() => {
```

```
  window.addEventListener("scroll", onScroll);
  return () => window.removeEventListener("scroll", onScroll);
}, []);
```

Why: Always remove listeners in cleanup.

Scenario 9. useMemo wrong deps (stale cached value)

Buggy

```
// Bug
const total = useMemo(() => calc(items), []);
```

Fixed

```
// Fix
const total = useMemo(() => calc(items), [items]);
```

Why: Memoization must be correct first.

Scenario 10. Async sets state after unmount

Buggy

```
// Bug
useEffect(() => { fetch(url).then(setData); }, [url]);
```

Fixed

```
// Fix
useEffect(() => {
  const c = new AbortController();
  fetch(url, { signal: c.signal }).then(setData);
  return () => c.abort();
}, [url]);
```

Why: Abort or ignore results after unmount.

Scenario 11. Derived state stored in state

Buggy

```
// Bug
const [total, setTotal] = useState(price * qty);
useEffect(() => setTotal(price * qty), [price, qty]);
```

Fixed

```
// Fix
const total = price * qty;
```

Why: Compute derived values during render.

Scenario 12. Effect uses render-created function

Buggy

```
// Bug
const run = () => fetch(`/api?q=${term}`);
useEffect(() => { run(); }, [run]);
```

Fixed

```
// Fix
useEffect(() => { fetch(`/api?q=${term}`); }, [term]);
```

Why: Move logic inside effect or stabilize with useCallback.

Scenario 13. Context provider value unstable

Buggy

```
// Bug
const value = { count, inc: () => setCount(c => c + 1) };
```

Fixed

```
// Fix
const inc = useCallback(() => setCount(c => c + 1), []);
const value = useMemo(() => ({ count, inc }), [count, inc]);
```

Why: Memoize provider values to reduce rerenders.

Scenario 14. Effect runs too often due to options object

Buggy

```
// Bug
useEffect(() => fetch(url, options), [url, options]);
```

Fixed

```
// Fix
const options = useMemo(() => buildOptions(token), [token]);
useEffect(() => fetch(url, options), [url, options]);
```

Why: Stabilize options.

Scenario 15. Stale closure in event listener

Buggy

```
// Bug
useEffect(() => {
  window.addEventListener("click", () => console.log(count));
  return () => window.removeEventListener("click", () => {});
}, []);
```

Fixed

```
// Fix
useEffect(() => {
  const onClick = () => console.log(count);
  window.addEventListener("click", onClick);
  return () => window.removeEventListener("click", onClick);
}, [count]);
```

Why: Use stable handler variable and correct deps.

6) Testing Hooks & Components - 8 Interview Questions

T1. How do you test a component that fetches in useEffect?

Mock fetch, render, await UI updates, and ensure cleanup or abort on unmount.

T2. How do you test effect cleanup?

Unmount and assert unsubscribe/abort/clearTimeout was called.

T3. How do you test timers in effects?

Use fake timers, advance time, then assert state/UI changes.

T4. How do you test race conditions?

Control promise resolution order and assert the latest request wins.

T5. How do you test stale closures?

Trigger async callback after state changes; ensure functional update or ref uses latest value.

T6. How do you test memoization behavior?

Measure renders of memoized children; ensure stable props prevent rerender.

T7. How do you test async cancellation?

Unmount or change deps; assert AbortController.abort called or old results ignored.

T8. How do you test transitions/deferred values?

Assert urgent input updates immediately and pending/deferred UI updates later.

Pseudo-test patterns (adapt to your runner)

```
// Async fetch
global.fetch = async () => ({ ok: true, json: async () => ({ name: "Ada" }) });
const ui = render(<UserGreeting userId="1" />);
await ui.findByText(/Ada/);

// Cleanup on unmount
const ui2 = render(<SomeComponent />);
ui2.unmount();
expect(unsubscribe).toHaveBeenCalled();

// Fake timers
useFakeTimers();
render(<DebouncedSearch />);
type(input, "react");
advanceTimersByTime(300);
```

7) References (Official Docs)

- Homepage: https://interviewquestions.guru/
- React Hooks API Reference: https://react.dev/reference/react/hooks
- useEffect: https://react.dev/reference/react/useEffect
- You Might Not Need an Effect: https://react.dev/learn/you-might-not-need-an-effect
- eslint-plugin-react-hooks (README): https://github.com/facebook/react/blob/main/packages/eslint-plugin-react-hooks/README.md
- exhaustive-deps rule: https://react.dev/reference/eslint-plugin-react-hooks/lints/exhaustive-deps
- React Hook Form docs: https://react-hook-form.com/get-started