Advanced Hooks & State Management

Q1

What happens when you call setState inside useEffect without a dependency array?

Expert

useEffect | Infinite Loops

✓ ANSWER

This creates an infinite loop. Without a dependency array, useEffect runs after every render. When setState is called inside, it triggers a re-render, which runs useEffect again, creating an endless cycle. React may eventually crash with "Maximum update depth exceeded" error. Always include proper dependencies or use an empty array [] if the effect should run only once.

Q2

Explain the difference between useCallback, useMemo, and React.memo.

Expert

Performance Memoization

✓ **ANSWER**

- **useCallback:** Memoizes function references, returns the same function instance between renders if dependencies haven't changed.
- **useMemo:** Memoizes the result of a computation, recalculates only when dependencies change.
- **React.memo:** HOC that prevents component re-renders if props haven't changed (shallow comparison).

When they fail: If dependencies contain objects/arrays created inline, memoization breaks due to new references. React.memo fails with callback props that aren't wrapped in useCallback.

Q3

Why does useReducer accept a third argument (init function)?

Advanced

useReducer Lazy Initialization

✓ **ANSWER**

The init function enables **lazy initialization** of state. It's called only once during component mount, unlike passing initialState directly which is evaluated on every render. Useful when initial state requires expensive computation (reading from localStorage, complex calculations). Example: useReducer(reducer, initialArg, init) where init(initialArg) returns initial state.

Q4

What's the difference between useState(() => initialValue) and useState(initialValue)?

Hard

useState Lazy Initialization

✓ ANSWER

useState(initialValue): Evaluates initialValue on every render (even though only used once).

useState(() => initialValue): Function is called only during initial render.

Matters when initialValue is expensive (e.g., JSON.parse(localStorage.getItem('data')) or heavy calculations). Use lazy initialization to avoid unnecessary computations on re-renders.

Q5

How would you implement a custom hook that tracks the previous value of a prop or state?

Advanced

Custom Hooks useRef

✓ ANSWER

```
function usePrevious(value) { const ref = useRef(); useEffect(() => { ref.current = value; }); return ref.current; }
```

Edge cases: Returns undefined on first render since ref isn't set yet. useEffect runs after render, so previous value is from last completed render. For synchronous access to previous value during render, you'd need more complex logic tracking both current and previous in refs.

Q6

Explain the cleanup function in useEffect. What happens if the cleanup function is async?

Expert

useEffect Cleanup

✓ **ANSWER**

Cleanup runs before re-executing effect (when dependencies change) and on component unmount. It's for cleaning up subscriptions, timers, event listeners, etc.

> **Async cleanup is NOT supported.** If you return an async function, React ignores the Promise. Cleanup must be synchronous. For async cleanup, handle it manually inside the cleanup function or use .then() to chain async operations, but don't return the Promise.

Q7

Why can't you conditionally call hooks? Explain what happens internally.

Expert

Hooks Rules React Internals

✓ ANSWER

React stores hooks in a **linked list** and relies on call order to match hooks between renders. Each hook call advances to the next position in the list. Conditional hooks break this order - React might match useState with useEffect from previous render, causing state corruption and crashes. The order must be identical across renders, which is why hooks can't be in conditions, loops, or nested functions.

Q8

What's the difference between useLayoutEffect and useEffect?

Advanced

useLayoutEffect Rendering

✓ ANSWER

useEffect: Runs asynchronously after paint (non-blocking).

useLayoutEffect: Runs synchronously after DOM mutations but before paint (blocking).

Visual bug scenario: Measuring DOM elements and updating state based on measurements. With useEffect, you see a flicker (initial render → paint → measure → update → repaint). useLayoutEffect prevents flicker by measuring before paint.

Q9

How would you implement useImperativeHandle? When is it necessary?

Expert

useImperativeHandle Refs

✓ ANSWER

Used with forwardRef to customize the ref value exposed to parent components. Instead of exposing the entire DOM element, you expose only specific methods.

```
const Input = forwardRef((props, ref) => { const inputRef = useRef(); useImperativeHandle(ref, () => ({ focus: () => inputRef.current.focus() })); return <input ref={inputRef} />; });
```

When necessary: Form controls, media players, third-party library wrappers.
Alternatives: Lifting state up, callback props (preferred declarative approach).

Q10

Explain React 18's automatic batching vs React 17.

Expert

React 18 Batching

✓ ANSWER

React 17: Only batches updates in event handlers. Updates in promises, setTimeout, async functions trigger separate renders.

React 18: Automatic batching everywhere - promises, timeouts, native events all batched. Multiple setState calls result in single render regardless of context. Use flushSync() to opt-out if synchronous updates needed. This significantly reduces unnecessary re-renders in async code.

Q11

How do you handle stale closures in useEffect?

Expert

Closures useEffect

✓ ANSWER

Stale closures occur when an effect captures old values. Solution: include all dependencies or use refs.

```
// Problem: count is stale useEffect(() => { const timer =
setInterval(() => { setCount(count + 1); // Always uses
initial count }, 1000); }, []); // Empty deps = stale
closure // Solution 1: Functional update setCount(c => c +
1); // Solution 2: Use ref const countRef = useRef(count);
countRef.current = count;
```

Always include dependencies or use functional updates to avoid stale values.

Q12

What is useId and when should you use it?

Advanced

useId React 18

✓ ANSWER

```
function FormField() { const id = useId(); return ( <>
<label htmlFor={id}>Name</label> <input id={id} /> </> ); }
```

useId generates unique IDs stable across server/client (SSR safe). Use for accessibility (form labels, ARIA), not for keys in lists. Solves hydration mismatch when using Math.random() or Date.now() for IDs.

Q13

How do you synchronize state with external systems using useSyncExternalStore?

Expert

useSyncExternalStore React 18

✓ ANSWER

```
function useWindowWidth() { return useSyncExternalStore(
(callback) => { window.addEventListener('resize', callback);
return () => window.removeEventListener('resize', callback);
}, () => window.innerWidth, () => 0 // Server snapshot ); }
```

For subscribing to external stores (Redux, browser APIs, custom observables). Ensures consistency with concurrent rendering. Takes subscribe, getSnapshot, getServerSnapshot functions.

Q14

Explain useInsertionEffect and its use cases.

Expert

useInsertionEffect CSS-in-JS

✓ ANSWER

useInsertionEffect fires before DOM mutations, earlier than useLayoutEffect. Designed for CSS-in-JS libraries to inject styles before layout calculations.

```
useInsertionEffect(() => { // Inject critical CSS const style = document.createElement('style'); style.textContent = '.btn { color: red; }'; document.head.appendChild(style); }, []);
```

Use only for: CSS-in-JS libraries, inserting global styles dynamically. Not for general side effects - use useEffect/useLayoutEffect instead.

Q15

How do you prevent unnecessary re-renders when using Context?

Expert

Context Performance

✓ ANSWER

```
// Split contexts const StateContext = createContext();
const DispatchContext = createContext(); function
Provider({children}) { const [state, dispatch] =
useReducer(reducer, initial); // Memoize dispatch (never
changes) const dispatchMemo = useMemo(() => dispatch, []);
return ( <StateContext.Provider value={state}>
<DispatchContext.Provider value={dispatchMemo}> {children}
</DispatchContext.Provider> </StateContext.Provider> ); }
```

Strategies: Split state/dispatch contexts, memoize context values, use React.memo on consumers, create multiple focused contexts instead of one large context.

Component Patterns & Architecture

Q16

Implement a Compound Component pattern. What are its advantages?

Expert

Compound Components Patterns

✓ ANSWER

Components that work together sharing implicit state. Example: Tabs component.

```
const TabsContext = React.createContext(); function Tabs({children}) { const [active, setActive] = useState(0); return <TabsContext.Provider value={{active, setActive}}> {children} </TabsContext.Provider>; }
```

Advantages: Flexible API, separation of concerns, implicit state sharing.
Disadvantages: Only direct children can access context (fixed with React.Children.map or context), less obvious API.

Q17

Explain Render Props pattern. Why did hooks largely replace it?

Advanced

Render Props Patterns

✓ ANSWER

Pattern where a prop is a function that returns JSX, enabling component logic reuse.

```
<Mouse render={({x, y}) => <Cat x={x} y={y} />} />
```

Why hooks replaced it: Render props cause "wrapper hell" with nested functions, harder to read, and create new function instances. Hooks provide cleaner syntax, better composition, and no nesting. **Still useful for:** Render-time decisions, when component rendering logic varies significantly.

Q18

What are Higher-Order Components (HOCs)? Explain their limitations.

Advanced

HOC Code Reuse

✓ ANSWER

Function that takes a component and returns enhanced component.

```
const withAuth = (Component) => { return (props) => { const {user} = useAuth(); if (!user) return <Login />; return <Component {...props} user={user} />; }; };
```

Limitations: Wrapper hell with multiple HOCs, prop name collisions, static methods not copied, refs don't pass through, harder debugging (component names). Hooks solve most use cases more elegantly.

Q19

Implement Provider pattern without Context API. What problems does Context solve?

Expert

Provider Pattern Context

✓ ANSWER

Without Context, you'd pass data through props at every level (prop drilling) or use global variables (breaks React's data flow).

```
// Manual Provider function Provider({children, value}) {
return React.Children.map(children, child =>
React.cloneElement(child, {sharedData: value}) ); }
```

Context solves: Prop drilling, makes data available deep in tree without passing through intermediates, maintains React's data flow principles, re-renders only consumers when value changes.

Q20

Explain Presentational vs Container components. Still relevant with hooks?

Hard

Architecture Separation of Concerns

✓ **ANSWER**

Container: Handle logic, state, side effects. Connect to data sources.
Presentational: Handle UI rendering, receive data via props, stateless.

With hooks: Less relevant as separation of concerns. Now we can have logic (custom hooks) and UI in same component. However, the principle of separating concerns still valuable - extract complex logic to custom hooks, keep components focused on rendering.

Q21

Controlled vs Uncontrolled components. When to use each?

Advanced

Forms Component Control

✓ ANSWER

Controlled: React state controls input value. Value and onChange props required. Full control over input data.

Uncontrolled: DOM controls value. Use ref to access. defaultValue prop for initial value.

Use Controlled: Form validation, conditional disabling, enforcing format, dynamic inputs.

Use Uncontrolled: Simple forms, file inputs (must be uncontrolled), integrating non-React code, performance with many inputs.

Q22

What is Component Composition and how does it differ from inheritance?

Hard

Composition | Design Principles

✓ ANSWER

Composition: Building components by combining smaller ones. Use props.children or multiple props for slots.

```
<Card> <CardHeader title="Hello" />
<CardBody>Content</CardBody> </Card>
```

Inheritance: Class extends another class.

React recommends composition over inheritance. Composition is more flexible, easier to understand, avoids tight coupling. Inheritance creates rigid hierarchies. Facebook says they haven't found use cases where inheritance is better than composition.

Q23

Explain the State Reducer pattern. When is it preferable?

Expert

State Reducer Inversion of Control

✓ ANSWER

Allows users to intercept and modify state changes in your component by providing their own reducer.

```
function useToggle({reducer = (s, a) => a}) { const [state, dispatch] = useReducer( (s, a) => reducer(s, a) || defaultReducer(s, a), false ); }
```

When preferable: Building libraries/reusable components where consumers need control over state logic. Allows customization without exposing internal implementation. Better than many props for complex components.

Q24

Explain Portals in React. What problems do they solve?

Advanced

Portals DOM

✓ ANSWER

Portals render children into DOM node outside parent component's hierarchy.

```
ReactDOM.createPortal(child, container)
```

Solves: Modals (need to render above everything), tooltips, popovers that break out of overflow:hidden containers, dropdowns that need to escape parent boundaries.

Limitations: Event bubbling still follows React tree (not DOM tree), styling can be complex, must manage multiple DOM roots.

Q25

How do you implement the Proxy Component pattern?

Expert

Proxy Pattern | Advanced Patterns

✓ ANSWER

```
function ProxyComponent({component: Component, ...props}) {
// Intercept and modify props const enhancedProps = {
...props, onClick: (e) => { logClick(e); props.onClick?.(e);
} }; return <Component {...enhancedProps} />; }
```

Use cases: Adding analytics, authentication checks, prop validation, logging.

Performance: Minimal overhead if props don't change frequently. Use React.memo if needed.

Q26

What is the Observer pattern in React? How do you implement it?

Expert

Observer Pattern Design Patterns

✓ ANSWER

```
class EventEmitter { constructor() { this.events = {}; }
on(event, callback) { if (!this.events[event])
this.events[event] = []; this.events[event].push(callback);
} emit(event, data) { this.events[event]?.forEach(cb =>
cb(data)); } off(event, callback) { this.events[event] =
this.events[event]?.filter(cb => cb !== callback); } }
function useEventListener(emitter, event, handler) {
useEffect(() => { emitter.on(event, handler); return () =>
emitter.off(event, handler); }, [emitter, event, handler]);
}
```

Useful for decoupled communication between components. Modern alternative: Context API, custom hooks.

Q27

Explain the Container/View pattern and its modern relevance.

Hard

Architecture Patterns

✓ **ANSWER**

Container handles data fetching and business logic. View handles pure UI rendering.

```
// Container function UserContainer() { const {data, loading} = useUser(); return <UserView user={data} loading={loading} />; } // View function UserView({user, loading}) { if (loading) return <Spinner />; return <div>{user.name}</div>; }
```

Modern approach: Extract logic to custom hooks instead of container components. More flexible and composable.

Q28

How do you implement the Strategy pattern in React?

Expert

Strategy Pattern Design Patterns

✓ ANSWER

```
const sortStrategies = { alphabetical: (items) =>
[...items].sort((a, b) => a.name.localeCompare(b.name)),
date: (items) => [...items].sort((a, b) => new Date(b.date)
- new Date(a.date)), popularity: (items) =>
[...items].sort((a, b) => b.views - a.views) }; function
List({items, sortBy}) { const sorted =
sortStrategies[sortBy](items); return sorted.map(item =>
<Item key={item.id} {...item} />); }
```

Allows selecting algorithm at runtime. Useful for different rendering, sorting, filtering, validation strategies.

Q29

What is the Factory pattern in React? Provide an example.

Advanced

Factory Pattern Component Creation

✓ ANSWER

```
const componentFactory = { text: (props) => <TextInput
{...props} />, number: (props) => <NumberInput {...props}
/>, date: (props) => <DatePicker {...props} />, select:
(props) => <Select {...props} /> }; function
DynamicField({type, ...props}) { const Component =
componentFactory[type]; if (!Component) return null; return
<Component {...props} />; } // Usage <DynamicField
type="text" name="email" />
```

Creates components based on type/config. Useful for form builders, dynamic UIs, plugin systems.

Q30

How do you implement slots pattern in React?

Advanced

Slots Composition

✓ ANSWER

```
function Layout({header, sidebar, footer, children}) {
return ( <div className="layout"> <header>{header}</header>
<div className="main"> <aside>{sidebar}</aside> <main>
{children}</main> </div> <footer>{footer}</footer> </div> );
} // Usage <Layout header={<Nav />} sidebar={<Menu />}
footer={<Footer />} > <Content /> </Layout>
```

Provides named insertion points. More flexible than single children prop. Vue-inspired pattern adapted for React.