**Shubham Kulkarni**
@shubhamkulkarni_insta

# 💡 [Top 10 Advanced React Interview Questions with Answers]

## ✅ 1. What is React Fiber and how does it differ from the old reconciliation algorithm?

React Fiber is a complete rewrite of React's core rendering engine introduced around React 16. Its main job is to **break rendering work into small chunks** and **prioritize tasks** based on urgency, which allows smoother rendering — especially in large apps.

In older versions of React, the rendering process was synchronous and blocking. So if the UI update was heavy, the browser could freeze.

But Fiber makes rendering **interruptible** — React can pause work, do something more urgent like handling user input, then come back and continue where it left off.

👉 In short: Fiber brings **asynchronous rendering, prioritization, and better user experience**.

---

## ✅ 2. How does React determine when to re-render a component?

React re-renders a component when:

- Its **state or props change**
- Its **parent component re-renders**, unless it's memoized

But here's the twist: React doesn't do **deep comparisons** by default. So even if an object or array looks "the same", if the reference changes, React treats it as different.

That's why techniques like:

- `React.memo()`
- `useMemo()`
- `useCallback()` are important in performance tuning — they help you avoid unnecessary renders when data hasn't really changed.

🧠 Bonus: The **key prop** also plays a role in determining if elements inside a list need to be re-rendered or not.

---

## ✅ 3. What are concurrent features in React and how do they help?

Concurrent features — introduced in React 18 — allow React to prepare multiple versions of the UI at the same time.

Let's say a user is typing a search query and you're fetching filtered results. Without concurrency, the UI could feel laggy. But with concurrent rendering, React can **keep the UI responsive** while preparing the new result list **in the background**.

📦 Features like `startTransition`, `useDeferredValue`, and automatic batching allow this.

So, concurrent features are like giving React **multitasking superpowers** — it can prioritize urgent updates like user input over non-urgent tasks like data fetching.

---

## ✅ 4. Explain React's batching behavior and what changed in React 18.

Batching means React **groups multiple state updates into one render** to improve performance.

Before React 18, batching only worked **inside React event handlers**. If you updated state inside a `setTimeout` or `fetch().then()`, those updates triggered multiple renders.

React 18 introduced **automatic batching** — so now, even updates inside promises, `setTimeout`, or native events are batched by default.

This means **fewer renders**, better performance, and more predictable behavior across async code.

---

## ✅ 5. What is the difference between useMemo and useCallback, and when not to use them?

- `useMemo` is used to memoize **values**
- `useCallback` is used to memoize **functions**

They both prevent re-creation on every render, but people often **overuse them**.

Here's the deal: if the memoization is **more expensive** than recalculating the value, you're actually hurting performance. Only use them if:

- The value/function is heavy to compute
- It causes unnecessary re-renders in children

Think of them as **performance tools**, not default habits.

---

## ✅ 6. How does Suspense work, and what are some real use cases beyond lazy loading?

`React.Suspense` lets you **wait for something** — like a lazy-loaded component or even **data fetching** (when used with libraries like React Query, Relay, or SWR).

You wrap a component with `<Suspense fallback={<Loading />}>`, and while the content is loading, React shows the fallback.

Beyond just lazy loading:

- You can use it with streaming SSR (in Next.js)
- You can coordinate parallel data loading
- You can delay transitions to avoid layout shifts

It's about **better UX during loading states**, not just code splitting.

---

## ✅ 7. What is useImperativeHandle and when should you use it?

This one's niche, but powerful.

`useImperativeHandle` is used with `forwardRef` to **expose custom methods** from a child component to a parent, instead of exposing the full DOM ref.

For example: You have a custom `Input` component and you want to expose a `focus()` method to the parent. Instead of giving full DOM access, you expose just that method through `useImperativeHandle`.

🛡️ It's a way to keep encapsulation, while still giving the parent controlled access.

---

## ✅ 8. How do you optimize large lists in React?

Rendering thousands of items can crash the browser.

To solve this:

- Use **windowing libraries** like `react-window` or `react-virtualized`
- Render only the visible portion of the list
- Use `key` props properly
- Combine with memoization (e.g., `React.memo`) to avoid re-rendering unchanged items

Think of it like a Netflix carousel — you only render what the user sees. This saves memory, time, and power.

---

## ✅ 9. How does useRef differ from useState?

`useRef` holds a **mutable value** that doesn't cause re-renders when it changes. `useState` holds values that **do** trigger re-renders.

Use `useRef` for:

- DOM refs ( `ref={myRef}` )
- Storing timers, counters, or previous values
- Avoiding re-renders on updates

So:

- `useState` = React cares when it changes
- `useRef` = React ignores the change

---

## ✅ 10. How does React handle hydration in SSR, and what problems can arise?

Hydration is the process where React **attaches event listeners** to server-rendered HTML on the client.

The problem is — if the HTML rendered on the server doesn't match what the client renders, you get a **hydration mismatch warning**.

Common causes:

- Using random values (like `Math.random()` ) during render
- Accessing `window` or `localStorage` on the server
- Not wrapping async components in `<Suspense>`

Hydration issues are subtle but can break your UI in production — so always test SSR apps thoroughly.

---