

React Interview Questions and Answers

Intern to senior level

React interview questions and answers from beginner to senior: virtual DOM, props vs state, hooks, useEffect, keys, useMemo vs useCallback, re-rendering, SSR vs CSR and performance optimization.

12 questions Read online: <https://www.fredevtemplates.com/interview-questions/react>

1. What is the virtual DOM in React and how does it work?

Level: intern React

The virtual DOM is a lightweight JavaScript copy of the UI that React compares with the previous version after each update, so it only changes the parts of the real DOM that actually differ.

When state or props change, React re-runs your component to produce a new tree of React elements (the virtual DOM). It then compares that tree with the previous one in a process called reconciliation, works out the minimal set of changes, and applies only those to the real DOM.

Touching the real DOM is comparatively expensive, and batching updates lets React do it once per update instead of once per change. The virtual DOM is not automatically faster than hand-written DOM code, but it gives you a declarative model: you describe what the UI should look like for the current state, and React figures out how to get there.

Key points

Virtual DOM = tree of plain objects describing the UI

Reconciliation compares old and new trees (diffing)

Only the differences are applied to the real DOM

2. What is the difference between props and state in React?

Level: intern React

Props are read-only inputs passed from a parent component, while state is data a component owns and can change over time, which triggers a re-render when updated.

Props make a component reusable: the parent decides the values, and the child must not modify them. State is private memory for a component, created with `useState` or `useReducer`. Updating state tells React to re-render that component with the new value.

A good rule is to keep state as low as possible in the tree, and lift it up to the closest common parent only when two components need to share it. Data that can be calculated from props or state should be computed during render and not stored in extra state.

Key points

Props flow down, are read-only and come from the parent

State is owned by the component and changes cause re-renders

Derived values should be calculated, not stored in state

```
function Counter({ step }) { // step is a prop (read-only)
  const [count, setCount] = useState(0); // count is state (owned here)
  return <button onClick={() => setCount(count + step)}>{count}</button>;
}
```

```
<Counter step={5} />
```

3. What are React hooks and what are the rules of hooks?

Level: junior React

Hooks are functions such as `useState` and `useEffect` that let function components use state, lifecycle behaviour and other React features; they must be called at the top level of a component or custom hook, never inside loops, conditions or nested functions.

Before hooks, state and lifecycle methods required class components. Hooks let you do the same in function components and, importantly, let you extract and reuse stateful logic as custom hooks without wrapper components or render props.

React tracks hooks by the order in which they are called during each render. If a hook is skipped in one render because of an if statement, every hook after it would read the wrong state. That is why the two rules exist: only call hooks at the top level, and only call them from React function components or other custom hooks (whose names start with use). The `eslint-plugin-react-hooks` package enforces both.

Key points

Call hooks at the top level only

Call hooks only from components or custom hooks

React relies on call order to match state to hooks

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

4. How does `useEffect` work in React and what is the dependency array?

Level: junior React

`useEffect` runs a function after the component renders; the dependency array tells React when to re-run it: with no array it runs after every render, with an empty array only after the first, and with values only when those values change.

Effects are for synchronising your component with something outside React, such as subscriptions, timers, network requests or the document title. The function you return from an effect is the cleanup, and React runs it before the next run of the effect and when the component unmounts.

Every value from the component that the effect uses should be listed in the dependency array, otherwise the effect works with stale values. In development with Strict Mode, React deliberately mounts, unmounts and re-mounts components once to help you find missing cleanup. Do not use effects for things you can calculate during render or handle in an event handler.

Key points

No array: every render. `[]` : once. `[a, b]`: when a or b change

Return a function to clean up

Do not use effects for derived data or user-triggered actions

```
useEffect(() => {
  const controller = new AbortController();
  fetch("/api/users/" + userId, { signal: controller.signal })
    .then(r => r.json())
    .then(setUser)
    .catch(err => { if (err.name !== "AbortError") setError(err); });

  return () => controller.abort(); // cleanup when userId changes or unmounts
}, [userId]);
```

5. Why do we need keys in React lists?

Level: junior React

Keys give each list item a stable identity so React can match items between renders and correctly add, remove or reorder them without recreating or mixing up their state.

When React diffs two lists it needs to know which new item corresponds to which old one. Without keys it falls back to matching by position, so inserting an item at the top makes every item look changed, and local state such as an input's text can end up attached to the wrong row.

A key must be unique among siblings and stable across renders. Use an ID from your data. Using the array index is acceptable only for static lists that are never reordered, filtered or inserted into. Never use `Math.random()`, since a new key on every render makes React remount every item.

```
{todos.map(todo => (
  <TodoItem key={todo.id} todo={todo} /> // stable ID, not the index
))}
```

6. What is the difference between controlled and uncontrolled components in React?

Level: junior React

A controlled component keeps the input's value in React state and updates it through `onChange`, while an uncontrolled component lets the DOM hold the value and you read it with a `ref` when needed.

Controlled inputs make React the single source of truth: the value comes from state and every keystroke calls a handler that updates state. This makes validation, formatting and conditional disabling easy. The cost is a re-render on each keystroke.

Uncontrolled inputs use `defaultValue` and `ref`, so the browser manages the value and you read it on submit. They are simpler for basic forms and file inputs (which are always uncontrolled). Form libraries often use uncontrolled inputs under the hood to avoid re-rendering the whole form while typing.

```
// controlled
const [email, setEmail] = useState("");
<input value={email} onChange={e => setEmail(e.target.value)} />

// uncontrolled
const ref = useRef(null);
<input defaultValue="" ref={ref} />
const onSubmit = () => console.log(ref.current.value);
```

7. What is prop drilling and how do you avoid it in React?

Level: junior React

Prop drilling is passing props through several layers of components that do not use them just to reach a deeply nested child; you avoid it with composition, the Context API or a state management library.

First try composition: pass the child component itself, or pass children, so intermediate components never need to know about the data. This is often enough and keeps components easy to test.

If many distant components need the same value, such as the current user or theme, use the Context API. Context re-renders every consumer when its value changes, so keep contexts small and focused, and split frequently changing values from stable ones. For complex client state, a store such as Redux Toolkit or Zustand lets components subscribe to just the slice they need.

```
const ThemeContext = createContext("light");

function App() {
  return (
    <ThemeContext.Provider value="dark">
      <Layout /> {/* no theme prop needed */}
    </ThemeContext.Provider>
  );
}

function Button() {
  const theme = useContext(ThemeContext);
  return <button className={theme}>Save</button>;
}
```

8. What is the difference between useMemo and useCallback in React?

Level: mid React

useMemo caches the result of a calculation, while useCallback caches a function definition; both only recompute when their dependencies change.

useMemo(() => compute(a, b), [a, b]) returns the cached value. useCallback(fn, deps) is shorthand for useMemo(() => fn, deps): it returns the same function instance between renders.

Neither makes your app faster by default; each adds its own cost. Use useMemo for genuinely expensive calculations, and use useCallback mainly when the function is passed to a child wrapped in React.memo or is a dependency of another hook. A stable function reference is useless if the child is not memoized, since the child re-renders with its parent anyway.

Key points

useMemo returns a value, useCallback returns a function

Both depend on a dependency array

Measure before optimising; memoisation has a cost

```
const visible = useMemo(
  () => todos.filter(t => t.status === filter),
  [todos, filter]
);

const handleSelect = useCallback((id) => setSelected(id), []);

const Row = React.memo(function Row({ item, onSelect }) { /* ... */ });
<Row item={item} onSelect={handleSelect} />
```

9. What is the difference between useRef and useState in React?

Level: mid React

Updating state triggers a re-render, while changing a ref's .current value does not; use state for data shown on screen and refs for DOM access or values that must persist without causing renders.

useRef returns a mutable object whose .current property lives for the whole life of the component. Typical uses are holding a DOM node (to focus an input or measure an element), storing a timer ID, or remembering the

previous value of a prop.

Because React does not know when you change a ref, the UI will not update. If a value affects what the user sees, it belongs in state. Avoid reading or writing `ref.current` during rendering, except for lazy initialisation.

```
function SearchBox() {
  const inputRef = useRef(null);
  const [query, setQuery] = useState(""); // changes the UI

  return (
    <>
      <input ref={inputRef} value={query} onChange={e => setQuery(e.target.value)} />
      <button onClick={() => inputRef.current.focus()}>Focus</button>
    </>
  );
}
```

10. What is the difference between SSR, SSG and CSR in React?

Level: mid React

CSR renders in the browser after JavaScript loads, SSR renders the HTML on the server for each request, and SSG renders the HTML once at build time; SSR and SSG give crawlers and users content faster, which helps SEO.

With client-side rendering the server sends an almost empty HTML file and the browser builds the page after downloading and running JavaScript. It is simple to host but the first paint is slower and search engines must run JavaScript to see the content.

Server-side rendering generates HTML per request, so content is visible immediately and is always fresh, at the cost of server work and a slower time to first byte. Static site generation builds HTML ahead of time and serves it from a CDN, which is the fastest option for content that does not change per user. Incremental static regeneration (as used by Next.js) rebuilds static pages in the background on a schedule.

In every case React then hydrates the HTML in the browser, attaching event handlers so the page becomes interactive. Mismatches between server and client output cause hydration errors.

Key points

CSR: render in browser. SSR: render per request. SSG: render at build time

SSR and SSG improve first paint and SEO

Hydration makes server-rendered HTML interactive

11. Why does a React component re-render and how do you prevent unnecessary re-renders?

Level: senior React

A component re-renders when its state changes, its parent re-renders, or a context it consumes changes; you prevent wasted renders by moving state down, using `React.memo`, and keeping props referentially stable.

By default, when a component renders, all of its children render too, even if their props did not change. React then compares the output and only updates the DOM where something is different, so a re-render is not necessarily a DOM update, but the work of calling your component still costs time.

Start by fixing the structure: keep state close to where it is used, and split a component so a frequently changing part (such as an input) does not drag a heavy sibling along. Passing children as props also helps, because the same element object does not re-render. If that is not enough, wrap pure components in `React.memo`, stabilise object and function props with `useMemo` and `useCallback`, and split large contexts. Use the React DevTools Profiler to confirm what actually rendered and why before optimising.

Key points

Triggers: own state change, parent render, context change
Move state down and compose with children first
React.memo only helps if props are stable

12. How do you optimize performance in a React application?

Level: senior React

Measure first with the Profiler and Lighthouse, then reduce unnecessary renders, split code with lazy loading, virtualise long lists, and optimise images and bundle size.

Rendering: avoid wasted re-renders by colocating state, using React.memo, useMemo and useCallback where profiling shows a benefit, and avoiding creating new objects in props for memoised children. For slow updates triggered by typing or filtering, useTransition and useDeferredValue keep the interface responsive.

Loading: split the bundle by route with React.lazy and Suspense (or Next.js dynamic imports), remove unused dependencies, and analyse the bundle with a tool such as webpack-bundle-analyzer. Serve properly sized, modern-format images and lazy-load below-the-fold content.

Data and lists: virtualise long lists with a library such as react-window so only visible rows are in the DOM, cache server data with React Query or SWR, and debounce expensive handlers. Track real-user metrics such as Largest Contentful Paint and Interaction to Next Paint, not just lab scores.

Key points

Profile before optimising

Reduce renders, reduce bundle size, reduce DOM size

Use real-user metrics to verify improvements

```
const Reports = React.lazy(() => import("./Reports"));

<Suspense fallback={<Spinner />}>
  <Reports />
</Suspense>
```

These answers are written for real interview conversations. Prefer explaining a trade-off out loud over reciting a definition. Full explanations and related questions: <https://www.fredevtemplates.com/interview-questions/react>