



Frontend · most asked

React interview questions

27 questions · 10 easy · 15 medium · 2 hard. Open one to see the answer, a code example and the likely follow-up, then mark it as known or to review.

1

What is the virtual DOM and how does reconciliation work?

Easy



#rendering #virtual dom

The **virtual DOM** is a lightweight tree of plain JavaScript objects describing what the UI should look like. When state changes, React calls your components again to build a new tree.

Reconciliation is the diff between the old and new trees. React then applies only the minimal set of real DOM changes, because touching the real DOM is the expensive part.

The diff uses two shortcuts to stay fast (roughly $O(n)$):

- a different element **type** at the same position means "throw away the subtree and rebuild"
- in lists, **keys** tell React which child is which across renders

The point is not that the virtual DOM is "faster than the DOM", it is that you write declarative UI and React works out the updates.

? Likely follow-up: What changed with the Fiber architecture?

2

Why do list items need a **key**, and why is the array index a bad key?

Easy



#lists #rendering

A **key** gives each list item a **stable identity** so React can match items between renders. It decides which DOM nodes (and which component state) to keep, move, create or delete.

Using the **index** breaks when the list is reordered, filtered or has items inserted at the top: the indexes shift, so React reuses the wrong component for the wrong data. Symptoms are inputs keeping the wrong text, wrong checkboxes ticked, or broken animations.

- Use a real id from the data (`todo.id`)
- The index is fine only for static lists that never reorder
- Changing a component's `key` on purpose forces a full remount, a handy way to reset state

Example

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

// reset a form when the selected user changes
<ProfileForm key={userId} userId={userId} />
```

3

What is the difference between props and state?

Easy



#props #state

Props are inputs passed from a parent. They are read-only inside the child; the parent owns them.

State is data a component owns and changes over time with `useState` or `useReducer`. Updating state **triggers a re-render** of that component and its children.

- Data flows one way: parents pass state down as props
- A child changes parent data by calling a callback prop (`onChange`)
- If a value can be computed from props or other state, do not store it in state; calculate it during render

Likely follow-up: What is derived state and why is copying props into state a bug?

4

Controlled vs uncontrolled components?

Easy



#forms

A **controlled** input gets its `value` from React state and updates it through `onChange`. React is the single source of truth, so validation, formatting and conditional UI are easy.

An **uncontrolled** input keeps its own value in the DOM. You set a `defaultValue` and read it when needed, through a `ref` or `FormData` on submit.

- Controlled: instant validation, dependent fields, live previews
- Uncontrolled: simple forms, file inputs (always uncontrolled), fewer re-renders
- Libraries like React Hook Form are mostly uncontrolled for performance

A common warning is switching from uncontrolled to controlled because `value` started as `undefined`; initialise with `''`.

Example

```
function Controlled() {
  const [email, setEmail] = useState('')
  return <input value={email} onChange={(e) => setEmail(e.target.value)} />
}

function Uncontrolled() {
  const ref = useRef<HTMLInputElement>(null)
  return <input ref={ref} defaultValue="" onBlur={() =>
    console.log(ref.current?.value)} />
}
```

5

What does "lifting state up" mean?

Easy



#state #architecture

When two sibling components need the same data, you move that state to their **closest common parent** and pass it down as props, with callbacks to change it.

This keeps one source of truth instead of two copies that drift out of sync. Example: a filter input and a results list both need `query`, so `query` lives in the parent page.

The trade-off is that the parent re-renders on every change and deep trees lead to prop drilling. That is when you reach for composition, context or a store.

6

Why does calling `setCount(count + 1)` three times only add 1? What is batching?

Medium



#hooks #state

State updates are not applied immediately. Inside one render, `count` is a fixed snapshot, so all three calls set it to the same value (`0 + 1`).

Use the **functional update** form when the new value depends on the old one:

`setCount(c => c + 1)`. React queues the updater functions and runs them in order, so three calls add 3.

Batching means React groups several state updates into **a single re-render**. Since React 18 this is automatic everywhere: event handlers, promises, timeouts and native listeners.

Example

```
function Counter() {
  const [count, setCount] = useState(0)
  function addThree() {
    setCount(count + 1)      // 0 + 1
    setCount(count + 1)      // 0 + 1 again
    setCount((c) => c + 1)    // uses the queued value: 2
  }
  return <button onClick={addThree}>{count}</button> // shows 2
}
```

? Likely follow-up: How would you read the new state right after setting it?

7

How does `useEffect` work: dependencies, cleanup, and why does it run twice in development?

Medium 🔥🔥 #hooks #effects

`useEffect` runs code **after** React has painted, to sync with something outside React (subscriptions, timers, the DOM, network).

- No array: runs after every render
- `[]`: runs once after mount
- `[a, b]`: runs when `a` or `b` changed (compared with `Object.is`)

The function you return is the **cleanup**. It runs before the effect runs again and on unmount: unsubscribe, clear timers, abort fetches.

In development, **StrictMode** mounts, unmounts and remounts each component, so effects run, clean up and run again. This is deliberate: it **exposes missing cleanup**. It does not happen in production.

Example

```
useEffect(() => {
  const controller = new AbortController()
  fetch(`/api/users/${id}`, { signal: controller.signal })
    .then((r) => r.json())
    .then(setUser)
    .catch(() => {})
  return () => controller.abort() // cleanup: cancel stale request
}, [id])
```

? Likely follow-up: When do you NOT need an effect?

8

What are common `useEffect` mistakes, and when do you not need an effect?

Medium 🔥🔥 #hooks #effects

Effects are for syncing with **external systems**. Many effects in real code are unnecessary:

- **Derived data:** computing `fullName` or a filtered list belongs in render (or `useMemo`), not in an effect that sets state
- **User events:** logic caused by a click belongs in the event handler
- **Resetting state on prop change:** use a `key` instead

Common bugs:

- missing dependencies, which cause stale values (trust the ESLint rule)
- objects or functions created during render as dependencies, so the effect runs every render
- race conditions in fetches without an abort or "ignore" flag

An effect that only sets state from other state is a **code smell**: it causes an extra render.

9

`useMemo` vs `useCallback` vs `React.memo`?

Medium 🔥🔥 #performance #hooks

- `useMemo(() => value, deps)` caches a **computed value** between renders
- `useCallback(fn, deps)` caches a **function** (it is `useMemo(() => fn, deps)`)

- `React.memo(Component)` skips re-rendering a **component** when its props are shallowly equal

They work together: `React.memo` is useless if the parent passes a new object or arrow function each render, so you stabilise those props with `useMemo` / `useCallback`.

Use them for expensive calculations, props to memoised children, or effect dependencies. Do not wrap everything: memoisation has its own cost. With the **React Compiler** (stable in 2025), much of this is done automatically at build time.

Example

```
const Row = React.memo(function Row({ item, onSelect }: RowProps) {
  return <li onClick={() => onSelect(item.id)}>{item.name}</li>
})

function List({ items, query }: ListProps) {
  const visible = useMemo(() => items.filter((i) =>
i.name.includes(query)), [items, query])
  const onSelect = useCallback((id: string) => console.log(id), [])
  return <ul>{visible.map((i) => <Row key={i.id} item={i} onSelect=
{onSelect} />)}</ul>
}
```

? Likely follow-up: How do you find out which components re-render too often?

10

What is `useRef` used for?

Easy



#hooks #refs

`useRef` returns a mutable object `{ current }` that persists across renders without causing a re-render when it changes.

Two main uses:

- **DOM access:** focus an input, measure size, scroll into view, integrate a non-React library
- **Instance values:** timer ids, the previous value of a prop, an "is mounted" flag, the latest callback

Rule of thumb: if changing the value should update the screen, it belongs in state; if not, a ref. Do not read or write `ref.current` during render, only in effects and handlers.

Example

```
function Search() {
  const inputRef = useRef<HTMLInputElement>(null)
  const timer = useRef<number | undefined>(undefined)
  useEffect(() => inputRef.current?.focus(), [])
  function onChange() {
    clearTimeout(timer.current)
    timer.current = window.setTimeout(() => console.log('search'), 300)
  }
  return <input ref={inputRef} onChange={onChange} />
}
```

11

What are the rules of hooks, and why do they exist?

Easy



#hooks

1. Call hooks only at the **top level** of a component or custom hook: not in conditions, loops or after an early `return`
2. Call them only from **React function components** or other **custom hooks**

React does not store hooks by name. It keeps a list per component and matches them by **call order** on every render. If a condition skips a hook, every hook after it reads the wrong state.

The `eslint-plugin-react-hooks` rule enforces this. The new `use` API is the exception: it can be called inside conditions.

12

What is a custom hook? Write one.

Medium



#hooks #reuse

A custom hook is a function whose name starts with `use` and that calls other hooks. It lets you **reuse stateful logic** (not state itself) between components: each component calling it gets its own independent state.

Good candidates: fetching, debouncing, media queries, local storage, form fields, subscriptions. Custom hooks replaced most HOCs and render props because they compose without wrapper components.

Example

```
function useDebounce<T>(value: T, delay = 300): T {
  const [debounced, setDebounced] = useState(value)
  useEffect(() => {
    const id = setTimeout(() => setDebounced(value), delay)
    return () => clearTimeout(id)
  }, [value, delay])
  return debounced
}

// usage
const debouncedQuery = useDebounce(query, 500)
```

? Likely follow-up: How would you test a custom hook?

13

What is Context, and what is its performance cost?

Medium



#context #state #performance

Context passes a value deep into the tree without threading props through every level. You create it with `createContext`, provide it near the top and read it with `useContext` (or `use`).

Good for values that change rarely: theme, current user, locale, feature flags.

The cost: when the provider's `value` changes, **every consumer re-renders**, even if it only uses one field. Passing a new object literal each render makes this happen on every parent render.

- Memoise the value with `useMemo`
- Split contexts (state and dispatch, or by concern)
- For fast-changing global state, use a store with selectors

Example

```
const ThemeContext = createContext<'light' | 'dark'>('light')

function App() {
  const [theme, setTheme] = useState<'light' | 'dark'>('light')
  return (
    <ThemeContext value={theme}>                                {/* React 19: no .Provider needed */}
    <Page />
    </ThemeContext>
  )
}

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

14

What is prop drilling and how do you avoid it?

Easy



#state #architecture

Prop drilling is passing props through several components that do not use them, only to reach a deeply nested child. It makes refactoring painful and couples the middle layers to data they do not care about.

Solutions, roughly in order:

- **Composition:** pass components as `children` or slot props, so the parent renders the deep child directly (often overlooked and the `simplest fix`)
- **Context** for widely shared, rarely changing data
- A **state library** (Zustand, Redux Toolkit, Jotai) for app-wide, frequently updated state
- **Server state** libraries (TanStack Query) so components fetch the data they need

15

When would you use Redux or Zustand instead of Context?

Medium



#state #redux

Context is a **transport mechanism**, not a state manager: it has no selectors, so every consumer re-renders on any change.

A store like **Zustand** or **Redux Toolkit** lets components subscribe to a slice of state, so only the components using that slice re-render. They also give devtools, middleware and patterns for complex updates.

- Context: theme, auth user, small apps
- Zustand: simple global client state with minimal boilerplate
- Redux Toolkit: large teams, complex flows, time-travel debugging, strict conventions
- TanStack Query / SWR: **server** state (caching, refetching), which is most "global state" in practice

Example

```
const useCart = create<CartState>((set) => ({
  items: [],
  add: (item) => set((s) => ({ items: [...s.items, item] })),
}))

function CartCount() {
  const count = useCart((s) => s.items.length) // re-renders only when
length changes
  return <span>{count}</span>
}
```

16

What causes a component to re-render, and how do you optimise it?

Medium



#performance #rendering

A component re-renders when:

- its **state** changes
- its **parent** re-renders (props changing or not, unless it is memoised)
- a **context** it reads changes
- a store hook it uses returns a new value

Re-rendering is not the same as DOM updates: React still diffs and may change nothing. Optimise only when you measure a real problem with the React DevTools Profiler.

Fixes:

- move state down closer to where it is used
- pass expensive children as `children` so they are not re-created
- `React.memo` plus stable props
- virtualise long lists (TanStack Virtual)

- `useTransition` / `useDeferredValue` for heavy updates

🔗 Likely follow-up: Does a child re-render if its props did not change?

17

How do you do code splitting with `React.lazy` and `Suspense`?

Easy



#performance #suspense

Code splitting breaks the bundle into chunks that load on demand, so the first page load ships less JavaScript.

`React.lazy(() => import('./Chart'))` creates a component whose code is fetched the first time it renders. While it loads, the nearest `<Suspense fallback={...}>` shows a fallback.

- Split by **route** first, then by heavy widgets (editors, charts, modals)
- `Suspense` also handles data: Server Components, `use(promise)` and libraries that support it
- Wrap with an **error boundary** in case the chunk fails to load

Example

```
const Chart = lazy(() => import('./Chart'))

function Dashboard() {
  return (
    <Suspense fallback={<Spinner />}>
      <Chart />
    </Suspense>
  )
}
```

18

What are error boundaries and what do they not catch?

Medium



#errors

An **error boundary** is a component that catches errors thrown while rendering its children and shows a fallback UI instead of unmounting the whole app.

It must still be a **class component** with `static getDerivedStateFromError` (to show the fallback) and `componentDidCatch` (to log). Most teams use the `react-error-boundary`

package.

They do **not** catch errors in:

- event handlers (use `try/catch`)
- async code such as `setTimeout` or promises outside render
- the boundary itself
- server-side rendering

Place them around routes and independent widgets so **one broken part does not blank the page**.

Example

```
class ErrorBoundary extends React.Component<Props, { hasError: boolean }> {
  state = { hasError: false }
  static getDerivedStateFromError() {
    return { hasError: true }
  }
  componentDidCatch(error: Error) {
    logToService(error)
  }
  render() {
    return this.state.hasError ? <p>Something went wrong.</p> :
    this.props.children
  }
}
```

19

How do class lifecycle methods map to hooks?

Medium 🔥🔥 #hooks #classes #lifecycle

- `constructor` / state: `useState` (lazy initialiser for expensive setup)
- `componentDidMount`: `useEffect(fn, [])`
- `componentDidUpdate`: `useEffect(fn, [deps])`
- `componentWillUnmount`: the **cleanup** function returned from `useEffect`
- `shouldComponentUpdate`: `React.memo`
- `getDerivedStateFromProps`: compute during render, or reset with a `key`
- `getSnapshotBeforeUpdate`: `useLayoutEffect` (roughly)
- error lifecycles: no hook yet, still a class

The mindset differs: hooks think in terms of **synchronising with dependencies**, not "when does this run".

? Likely follow-up: `useEffect` vs `useLayoutEffect` ?

20

What is a portal, and when do you use one?

Easy 🔥💡 #dom #ui

`createPortal(children, domNode)` renders children into a **different DOM node**, usually one attached to `document.body`, while keeping them in the same place in the **React** tree.

Use it for modals, tooltips, dropdowns and toasts that must escape a parent's `overflow: hidden` or `z-index` stacking context.

Because the React tree is unchanged, **context and event bubbling still work** as if the portal were nested normally: a click inside the modal bubbles to React ancestors.

Example

```
function Modal({ children }: { children: React.ReactNode }) {  
  return createPortal(  
    <div className="backdrop">{children}</div>,  
    document.body,  
  )  
}
```

21

What is `forwardRef`, and what changed in React 19?

Medium 🔥💡 #refs #react 19

By default a `ref` passed to your component does not reach the DOM element inside it. `forwardRef` wrapped a component so it received the `ref` as a second argument and could attach it to an inner element (for focusing a custom `<Input>`, for example).

In **React 19**, `ref` is **a normal prop** for function components, so you just destructure it; `forwardRef` is no longer needed and will be deprecated.

`useImperativeHandle` lets you expose a custom API (`focus`, `reset`) through the ref instead of the raw DOM node.

Example

```
// React 19
function TextInput({ ref, ...props }: React.ComponentProps<'input'>) {
  return <input ref={ref} {...props} />
}

// before React 19
const OldInput = forwardRef<HTMLInputElement, Props>((props, ref) =>
  <input ref={ref} {...props} />)
)
```

22

Higher-order components vs render props vs hooks?

Medium 🔥💧 #patterns #reuse

A **HOC** is a function that takes a component and returns a new one with extra props (`withAuth(Page)`). **Render props** pass a function as a prop that returns UI.

Both share logic but cause **wrapper hell**, unclear prop origins and name collisions.

Custom hooks solve the same problem with plain function calls: no extra components, explicit inputs and outputs, easy to compose. They are the **default choice today**.

HOCs still appear for cross-cutting wrappers (route guards, `React.memo` itself is one) and in older codebases.

23

What is concurrent rendering? Explain `useTransition` and `useDeferredValue`.

Hard 🔥🔥 #concurrent #performance

Since React 18, rendering is **interruptible**: React can start rendering an update, pause it to handle something more urgent (like typing), and resume or throw it away.

`useTransition` marks a state update as **non-urgent**. Typing stays responsive while the expensive re-render (a big filtered list, a tab switch) happens in the background.

`isPending` lets you show a hint. In React 19, transitions can also be **async** functions.

`useDeferredValue(value)` gives you a lagging copy of a value, useful when you do not own the setter (the value comes from props).

Neither makes code faster; they change **priority**.

Example

```
function Search() {
  const [query, setQuery] = useState('')
  const [results, setResults] = useState<Item[]>([])
  const [isPending, startTransition] = useTransition()

  function onChange(e: React.ChangeEvent<HTMLInputElement>) {
    setQuery(e.target.value) // urgent
    startTransition(() => setResults(filter(e.target.value))) // can wait
  }

  return <><input value={query} onChange={onChange} />{isPending &&
<Spinner />}<List items={results} /></>
}
```

24

What is new in React 19?

Medium 🔥🔥 #react 19

The main additions:

- **Actions:** async functions passed to `<form action>` or `startTransition`, with pending state, errors and form reset handled for you
- `useActionState` for form state and result, `useFormStatus` for a submit button's pending state, `useOptimistic` for optimistic UI
- `use(promise)` / `use(context)`: read a resource during render, works with Suspense and inside conditions
- **Server Components** and **Server Actions** stable (used by frameworks such as Next.js)
- `ref` as a prop, `<Context>` as a provider, ref cleanup functions
- `<title>` and `<meta>` rendered anywhere are hoisted to `<head>`
- better hydration error messages

Alongside it, the **React Compiler** `auto-memoises` components.

🔗 Likely follow-up: How does `useOptimistic` work?

25

What are React Server Components and how are they different from SSR?

Hard



#server components #react 19

Server Components (RSC) run only on the server (at build time or per request). They can be `async`, read the database or file system directly, and `send zero JavaScript` to the browser for themselves. They cannot use state, effects or browser APIs.

Client Components (marked with `'use client'`) are the interactive ones; they are still pre-rendered to HTML on the server, then hydrated.

SSR is about producing HTML for the first load. RSC is about *where components live*: the server sends a serialised component tree, which can be refetched on navigation without losing client state.

- Server Components can import and render Client Components, not the reverse (pass them as `children` instead)
- Props crossing the boundary must be serialisable

26

What is hydration, and what causes hydration mismatch errors?

Medium



#ssr #hydration

Hydration is React attaching event handlers and state to HTML that was rendered on the server, instead of rebuilding the DOM.

React expects the first client render to produce `exactly the same markup` as the server. If not, you get a hydration error and React re-renders that part on the client.

Common causes:

- `Date.now()`, `Math.random()`, locale or timezone formatting
- checking `typeof window` or reading `localStorage` during render
- invalid HTML nesting (`<div>` inside `<p>`)
- browser extensions changing the DOM

Fixes: move browser-only values into `useEffect`, render them after mount, or use `suppressHydrationWarning` for a single known value like a timestamp.

How do you handle forms in modern React?

Medium



#forms #react 19

Options, from simplest to richest:

- **Controlled inputs** with `useState` for small forms with live validation
- **Uncontrolled** with `FormData` on submit, which needs less code and fewer re-renders
- **React 19 Actions:** pass a function to `<form action>`; `useActionState` tracks the result and pending state and works with Server Actions (forms even submit before JavaScript loads)
- **React Hook Form + Zod** for large forms: field-level validation, shared schema with the server

Always **validate on the server** too, and handle pending state so users cannot double-submit.

Example

```
function Signup() {
  const [state, formAction, isPending] = useActionState(
    async (_prev: string | null, data: FormData) => {
      const res = await signup(data.get('email') as string)
      return res.ok ? null : 'Email already used'
    },
    null,
  )
  return (
    <form action={formAction}>
      <input name="email" type="email" required />
      <button disabled={isPending}>Sign up</button>
      {state && <p role="alert">{state}</p>}
    </form>
  )
}
```