

# React

Components, rendering, hooks, state management and performance

## 1 What React does

### Definition

React is a library for building UIs from **components** – functions that return what the UI should look like for the current data.  $UI = f(state)$ .



- Declarative:** describe the result, React works out the DOM updates.
- One-way data flow:** props go down, events go up through callbacks.
- Reconciliation:** different element type → replace the subtree; same type → update props; lists are matched by **key**.

2

## JSX, components and props

### A typed component

```
type CardProps = { title: string; count?: number; onSelect: (title: string) =>
void; children?: React.ReactNode }

export function Card({ title, count = 0, onSelect, children }: CardProps) {
  return (
    <article className="card" onClick={() => onSelect(title)}>
      <h3>{title}</h3>
      {count > 0 && <span>{count} new</span>}    {/* not {count && ...} - 0 would
render */}
      {children}
    </article>
  )
}
```

- JSX compiles to `jsx()` / `React.createElement` calls that return plain objects.
- Props are read-only. A component re-renders when its state, its parent or a consumed context changes.
- Rules: one root (or a Fragment `<>`), `className`, `htmlFor`, camelCase events, `{}` for expressions.

## State and rendering

### Updater functions and immutability

```
const [count, setCount] = useState(0)
setCount(count + 1); setCount(count + 1) // → +1 (same snapshot)
setCount((c) => c + 1); setCount((c) => c + 1) // → +2

const [todos, setTodos] = useState<Todo[]>([])
setTodos((t) => [...t, newTodo]) // add
setTodos((t) => t.map((x) => (x.id === id ? { ...x, done: true } : x))) //
update
setTodos((t) => t.filter((x) => x.id !== id)) // remove
```

- ✎ State updates are **batched** and applied on the next render — the variable doesn't change immediately.
- ✎ Never mutate state ( `todos.push(x)` ): React compares references, so it won't re-render.
- ✎ Derive values during render instead of storing them in extra state.
- ✎ **Lift state up** to the closest common parent when siblings need it.

### Common mistake

Using an array index as `key` for a list that can be reordered or filtered — input state ends up on the wrong row. Use a stable id.

## Definition

`useEffect` synchronises a component with something outside React (network, subscriptions, timers, the DOM). It runs after the browser paints.

### Fetch with cleanup (no race conditions)

```
useEffect(() => {
  const controller = new AbortController()
  setLoading(true)
  fetch(`/api/users/${id}`, { signal: controller.signal })
    .then((r) => r.json())
    .then(setUser)
    .catch((e) => { if (e.name !== 'AbortError') setError(e) })
    .finally(() => setLoading(false))
  return () => controller.abort() // cleanup when id changes or on unmount
}, [id])
```

| Dependency array    | Runs                                                                   |
|---------------------|------------------------------------------------------------------------|
| none                | after every render                                                     |
| <code>[]</code>     | once after mount (twice in dev Strict Mode, to catch missing cleanups) |
| <code>[a, b]</code> | after mount and whenever a or b changes                                |

You might not need an effect: derive data during render, handle user actions in event handlers, and use a data library (TanStack Query / SWR) for server state.

| Hook                                                          | Use it for                                                             |
|---------------------------------------------------------------|------------------------------------------------------------------------|
| <code>useRef</code>                                           | a mutable value that survives renders without re-rendering; DOM refs   |
| <code>useMemo</code>                                          | cache an expensive calculation                                         |
| <code>useCallback</code>                                      | cache a function identity (for memoised children / effect deps)        |
| <code>useContext</code>                                       | read a context value without prop drilling                             |
| <code>useReducer</code>                                       | complex state transitions: <code>(state, action) =&gt; newState</code> |
| <code>useLayoutEffect</code>                                  | measure the DOM before paint                                           |
| <code>useTransition</code> /<br><code>useDeferredValue</code> | mark updates as non-urgent so typing stays responsive                  |
| <code>useId</code>                                            | stable ids for accessibility attributes                                |

### Common mistake

Rules of hooks: call them at the top level of components or custom hooks — never inside conditions or loops. React matches hooks by call order.

### A custom hook

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

| Kind of state                                 | Best tool                                       |
|-----------------------------------------------|-------------------------------------------------|
| Local UI state (open, input value)            | <code>useState</code> / <code>useReducer</code> |
| Shared, rarely changing (theme, user, locale) | Context                                         |
| Server data (lists, details, caching)         | TanStack Query / SWR                            |
| Complex global client state                   | Redux Toolkit / Zustand                         |
| Shareable UI state (filters, page)            | URL search params                               |

- Context re-renders **every consumer** when its value changes — split contexts and memoise values.
- Redux Toolkit: `createSlice` (reducers with Immer), `configureStore`, `useSelector` picks just what a component needs, RTK Query for data fetching.

- ✎ Measure first with the React DevTools Profiler.
- ✎ `React.memo` skips re-rendering a child when its props are shallow-equal (pair with `useCallback/useMemo` for function/object props).
- ✎ Code-split routes and heavy widgets with `lazy()` + `<Suspense>`.
- ✎ Virtualise long lists (TanStack Virtual / react-window).
- ✎ Keep state close to where it is used so fewer components re-render.
- ✎ The **React Compiler** can add memoisation automatically.

**React 19** adds Actions (`useActionState`, `useFormStatus`, `useOptimistic`), the `use()` hook for promises and context, `ref` as a normal prop, and document metadata tags inside components.

### Revision

UI = f(state). Immutable updates and updater functions, stable keys, effects only for external sync (with cleanup), hooks at the top level, the right tool for each kind of state, measure before memoising.

### Q. What is the Virtual DOM / reconciliation?

A. React builds a new element tree on each render, diffs it with the previous one, and applies only the needed DOM changes. Element type and key drive the matching.

### Q. State vs props?

A. Props come from the parent and are read-only; state is owned by the component and changed with its setter.

### Q. Why do lists need keys?

A. So React can match items between renders and keep each item's state and DOM. Keys must be stable and unique.

### Q. Why can't hooks be called conditionally?

A. React tracks hooks by their call order; changing the order between renders mixes up the state slots.

### Q. useMemo vs useCallback vs React.memo?

A. useMemo caches a value, useCallback caches a function, React.memo skips re-rendering a component whose props are shallow-equal.

### Q. useEffect vs useLayoutEffect?

A. useEffect runs after paint; useLayoutEffect runs after DOM changes but before paint — use it to measure layout.

### Q. Why does `setCount(count + 1)` twice add only 1?

A. Both calls read the same render's snapshot. Use the updater form `setCount(c => c + 1)`.

### Q. Context vs Redux?

A. Context passes values down without props; Redux is a full state system with middleware, devtools and selectors that limit re-renders.

### Q. Controlled vs uncontrolled inputs?

A. Controlled: the value lives in React state (value + onChange). Uncontrolled: the DOM keeps it and you read it with a ref or FormData.

### Q. How do you avoid a fetch race condition in useEffect?

A. Abort the previous request in the cleanup (AbortController) or ignore stale responses with a flag.

Q. What is lifting state up?

A. Moving shared state to the closest common parent and passing it down as props.

Q. What is new in React 19?

A. Actions with useActionState/useFormStatus/useOptimistic, the use() hook, ref as a prop, and metadata tags rendered from components.