



React Hooks

Version 19.2 · verified 2026-10-24

Every built-in React 19.2 hook grouped by job — state, context, refs, effects, memoization, transitions, actions — with the traps.

Contents

Mental model	3
State: useState and useReducer	3
useContext	5
Refs: useRef and useImperativeHandle	7
useEffect	9
Effect variants: layout, insertion, event	10
Memoization: useMemo and useCallback	13
Transitions: useTransition and useDeferredValue . .	15
Actions and forms: useActionState and useOptimistic	17
Utility hooks: external stores, ids, DevTools	19

Further reading	21
-----------------------	----

Mental model

A hook is a function that lets a component hook into React's machinery: remember a value between renders, read a context, synchronize with the outside world, or schedule work at a lower priority. React identifies each hook by its **call order**, so hooks run only at the top level of a component or custom hook — never in a condition, loop, nested function, or after an early return. Each render is a snapshot: the values a hook returns are fixed for that render, and a setter asks for the *next* one.

State: useState and useReducer

Hook	Returns
<code>useState(init)</code>	<code>[state, setState]</code>
<code>useState(() => init())</code>	Lazy initializer, runs once
<code>setState(prev => next)</code>	Updater based on pending state
<code>useReducer(fn, arg, init?)</code>	<code>[state, dispatch]</code>

Hook	Returns
<code><Form key={id} /></code>	New key resets all state

Use `useState` for independent values and `useReducer` when several fields change together or the next state depends on an action. State is read-only: replace objects and arrays, never mutate them. React skips the re-render if the new value is `Object.is`-equal to the current one. Pass the initializer as a function (`createTodos`, not `createTodos()`) or it runs on every render.

```
import { useReducer } from "react";

type A = { type: "add" | "reset" };
const f = (n: number, a: A) =>
  a.type === "add" ? n + 1 : 0;

export function Counter() {
  const [n, go] = useReducer(f, 0);
```

```
return (  
  <button onClick={() => go({ type: "add" })}>  
    {n}  
  </button>  
)  
};  
}
```

Gotcha: Setting state does not change the variable in the running render. `setN(n + 1)` three times adds one; `setN(x => x + 1)` three times adds three. In a reducer, mutating state and returning it is a no-op because the reference is unchanged.

useContext

Piece	Role
<code>createContext(default)</code>	Defines the context
<code><Ctx value={v}></code>	Provides <code>v</code> to the subtree
<code>useContext(Ctx)</code>	Reads the nearest provider's value

`useContext` reads the value from the closest provider **above** the calling component and re-renders it whenever that value changes (compared with `Object.is`). The default is used only when no provider exists. In React 19 the context itself is the provider; `<Ctx.Provider>` is the older spelling.

```
import { createContext, useContext } from "react";

const Theme = createContext("light");

function Label() {
  return <p>{useContext(Theme)}</p>;
}

export const App = () => (
  <Theme value="dark">
    <Label />
  </Theme>
);
```

Gotcha: `value={{ user, login }}` creates a new object every render, so every consumer re-renders every time. Stabilize it with `useMemo` (and `useCallback` for functions) when that costs you.

Refs: `useRef` and `useImperativeHandle`

Hook	Use
<code>useRef(init)</code>	<code>{ current }</code> that survives renders
<code><input ref={r} /></code>	<code>r.current</code> is the DOM node
<code>useImperativeHandle(ref, fn)</code>	Expose a custom handle

A ref is a mutable box that does **not** trigger a re-render when `current` changes. Use it for DOM nodes, timer IDs, and any value that is not part of the output. Write refs in event handlers or effects, not during render. A component receives `ref` as a plain prop in React 19;

pass it on, or give the parent a narrow handle with `useImperativeHandle` instead of the raw node.

```
import { useRef, type Ref } from "react";
import { useImperativeHandle } from "react";

type H = { focus(): void };

export function Field({ ref }: { ref: Ref<H> }) {
  const el = useRef<HTMLInputElement>(null);
  useImperativeHandle(ref, () => ({
    focus: () => el.current?.focus(),
  }), []);
  return <input ref={el} />;
}
```

Gotcha: Reading `ref.current` while rendering gives stale or null values and makes output unpredictable. The one exception is lazy init:

```
if (r.current === null) r.current = new Thing().
```

useEffect

Dependencies	Runs after
[a, b]	Mount, then when a or b changes
[]	Mount only
(omitted)	Every render

An Effect synchronizes the component with something **outside** React: a socket, a timer, a browser API, a third-party widget. List every reactive value the Effect reads. When dependencies change, React runs the old cleanup, then the new setup; on unmount it runs the cleanup. Write cleanup so setup, cleanup, setup is indistinguishable from setup once — development Strict Mode does exactly that extra cycle to prove it.

```
import { useEffect, useState } from "react";

export function Profile({ id }: { id: string }) {
  const [name, setName] = useState<string>();
  useEffect(() => {
```

```
let ignore = false;

fetch(`/api/users/${id}`)
  .then(r => r.json())
  .then(u => ignore || setName(u.name));

return () => void (ignore = true);
}, [id]);

return <p>{name ?? "Loading..."}</p>;
}
```

Gotcha: An object or function created in the component body is new every render, so listing it makes the Effect re-run every render. Create it inside the Effect, or depend on its primitive parts. Don't silence the lint rule, and don't use an Effect to derive values or answer a click.

Effect variants: layout, insertion, event

Hook	Fires
<code>useLayoutEffect</code>	After DOM update, before paint

Hook	Fires
<code>useInsertionEffect</code>	Before any DOM change (CSS-in-JS)
<code>useEffectEvent(fn)</code>	Non-reactive callback for Effects

`useLayoutEffect` has the same signature as `useEffect` but blocks the browser's paint. Use it to measure the DOM and re-render before the user sees a flicker, such as positioning a tooltip. It does not run on the server.

`useInsertionEffect` is for CSS-in-JS library authors only; refs are not attached yet and you can't set state.

`useEffectEvent` (stable since React 19.2) wraps logic that belongs to an Effect but should always see the latest props and state without re-triggering it. Never list it in the dependency array, and call it only from Effects.

```
import { useEffect, useEffectEvent } from "react";
```

```
type Props = { room: string; muted: boolean };

export function Chat({ room, muted }: Props) {
  const onUp = useEffectEvent(() => {
    if (!muted) console.log("Connected");
  });
  useEffect(() => {
    const t = setTimeout(onUp, 1000); // "connect"
    return () => clearTimeout(t);
  }, [room]); // muted is not a dependency
  return <p>{room}</p>;
}
```

Warning: Don't use `useEffectEvent` to hide a dependency that *should* re-run the Effect. It is only for logic that is genuinely an event fired from inside an Effect.

Memoization: useMemo and useCallback

Hook	Caches
<code>useMemo(() => v, deps)</code>	A computed value
<code>useCallback(fn, deps)</code>	A function (same as <code>useMemo(() => fn)</code>)
<code>memo(Component)</code>	Skips render if props are equal

Both hooks are performance hints, not guarantees — React may discard the cache. Reach for them only when the calculation is measurably slow, when the value is a prop of a `memo`-wrapped child, or when it is a dependency of another hook. They do not speed up the first render, and the React Compiler can add most of this memoization for you.

```
import { useCallback, useMemo } from "react";
```

```
export function Total({ xs }: { xs: number[] }) {  
  const total = useMemo(  
    () => xs.reduce((a, b) => a + b, 0),  
    [xs],  
  );  
  const show = useCallback(  
    () => alert(total),  
    [total],  
  );  
  return <button onClick={show}>{total}</button>;  
}
```

Gotcha: A memoized callback passed to a child that is not wrapped in `memo` does nothing – the child re-renders anyway. One unstable dependency, such as an object built during render, also defeats the cache on every render.

Transitions: `useTransition` and `useDeferredValue`

Hook	Returns
<code>useTransition()</code>	<code>[isPending, startTransition]</code>
<code>startTransition(() => ...)</code>	Marks updates as non-urgent
<code>useDeferredValue(v)</code>	A lagging copy of <code>v</code>

Both keep the UI responsive during a slow render by letting urgent updates, like typing, interrupt non-urgent ones. Use `useTransition` when you own the setter; use `useDeferredValue` when you only have the value, such as a prop. The deferred render restarts if a newer value arrives, so wrap a slow child in `memo` or it re-renders with the urgent update anyway.

```
import { memo, useDeferredValue } from "react";
```

```
const Slow = memo(({ q }: { q: string }) => (  
  <p>{q}</p>  
));  
  
export function Results({ q }: { q: string }) {  
  const late = useDeferredValue(q);  
  const stale = q !== late;  
  return (  
    <div style={{ opacity: stale ? 0.5 : 1 }}>  
      <Slow q={late} />  
    </div>  
  );  
}
```

Gotcha: Transitions can't control a text input — keep the input's state synchronous. State set after an `await` inside `startTransition(async ...)` must be wrapped in another `startTransition`.

Actions and forms: `useActionState` and `useOptimistic`

Hook	Returns
<code>useActionState(fn, init)</code>	<code>[state, action, isPending]</code>
<code>useOptimistic(value)</code>	<code>[optimistic, setOptimistic]</code>

An **Action** is an async function run inside a transition.

`useActionState` holds the state an Action returns: `fn` receives `(prevState, formData)`, and the returned `action` goes straight on `<form action>`. Dispatches queue and run in order. Return expected errors such as validation as state; a thrown error goes to the nearest Error Boundary.

`useOptimistic` shows a temporary value while an Action is pending, then reverts to the real `value` when it finishes — or if it throws. Call its setter inside an Action.

```
import { useActionState } from "react";

const save = async (_, d: FormData) =>
  d.get("name") ? "" : "Name is required";

export function NameForm() {
  const [err, go, wait] = useActionState(save,
    "");
  return (
    <form action={go}>
      <input name="name" />
      <button disabled={wait}>Save</button>
      {err && <p role="alert">{err}</p>}
    </form>
  );
}
```

Gotcha: The Action's signature changes with `useActionState`: `prevState` comes **first**, so the `FormData` you used to receive alone is now the second argument.

Utility hooks: external stores, ids,

DevTools

Hook	Purpose
<code>useSyncExternalStore(sub, snap)</code>	Subscribe to a non-React store
<code>useId()</code>	Stable unique id for ARIA links
<code>useDebugValue(v, fmt?)</code>	Label a custom hook in DevTools

`useSyncExternalStore` reads a store that lives outside React (browser APIs, third-party state) without tearing during concurrent renders. `subscribe` must have a

stable identity — define it outside the component. For server rendering, add a third argument `getServerSnapshot`. `useId` generates an id that matches between server and client; call it once and append suffixes (`id + "-name"`) for related fields. Use `useDebugValue` sparingly, for shared hooks that are hard to inspect.

```
import { useSyncExternalStore } from "react";

function subscribe(cb: () => void) {
  addEventListener("resize", cb);
  return () => removeEventListener("resize", cb);
}

export function useWidth() {
  return useSyncExternalStore(
    subscribe,
    () => innerWidth,
    () => 1024, // server value
  );
}
```

```
);  
}
```

Gotcha: `getSnapshot` must return the same value until the store changes. Returning a fresh object each call (`{ todos }`) loops forever with “The result of `getSnapshot` should be cached.”

Further reading

- [Built-in React Hooks](#)
- [Rules of Hooks](#)
- [You Might Not Need an Effect](#)
- [Lifecycle of Reactive Effects](#)
- [Reusing Logic with Custom Hooks](#)