React Basic

Version 19.2 · verified 2026-08-11

A gradual walk through client-side React — components, lists and keys, the core hooks, and context-based state sharing.

Contents

Mental model

A component is a function that returns JSX describing the UI for the current props and state. React re-renders by calling that function again and diffing the result — never by mutating the DOM yourself. State changes trigger a re-render; Effects exist to synchronize with things outside React, not to compute values you render. Everything here runs in the browser; the server half of React 19 is a separate sheet.

Components and props

Syntax	Meaning
`function C(props)`	A component is a plain function
`<C name="x" />`	Props are passed like attributes
`props.children`	The JSX nested inside the tag

Props are read-only inputs, exactly like function arguments. A component re-renders when its own state changes or when its parent re-renders and hands it new props — and its return value must be a pure function of

those two inputs, because React may call it more often than you expect.

```
function Greeting({ name }: { name: string }) {
  return <p>Hello, {name}!</p>;
}
```

> **Gotcha:** Mutating a prop (`props.name = "x"`) changes nothing on screen and breaks the one-way data flow every other React feature assumes. Send a value up through a callback instead of writing downward.

Rendering lists and conditionals

Pattern	Renders
`cond && <A />`	`<A />` , or nothing
`cond ? <A /> : <B />`	One of two branches
`items.map(…)`	One keyed element per item

JSX has no template language: conditionals and loops are ordinary JavaScript expressions. Keys are how React matches elements across renders once items can move, be inserted, or be deleted, so they must be unique among siblings and must not change — never generate one during render. A key is a hint to React, not a prop: components never receive it, so pass the id separately if the child needs it.

```
type Note = { id: string; title: string };

function Notes({ notes }: { notes: Note[] }) {
  if (notes.length === 0)
    return <p>Nothing yet.</p>;
  return (
    <ul>
      {notes.map(n => (
        <li key={n.id}>{n.title}</li>
      ))}
    </ul>
```

```
  );
}
```

> **Gotcha:** Array index as a key “often leads to subtle and confusing bugs”. Insert a row at the front and every index shifts, so React reuses the wrong DOM nodes — the first row’s typed input text stays behind on the row that took its place.

useState

Syntax	Meaning
`useState(initial)`	A `[value, setter]` pair
`setValue(v)`	Re-render with this value
`setValue(v => …)`	Next value from the previous one

State behaves like a snapshot: calling the setter does not change the variable in the render that is already running, it requests the next one. React batches the

updates in an event handler and re-renders once they have all been queued. State is tied to a component's position in the tree, which is why changing a component's `key` is the supported way to reset all of its state.

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

function Counter() {
  const [count, setCount] = useState(0);
  return (
    <button onClick={() => setCount(c => c + 1)}>
      {count}
    </button>
  );
}
```

> **Gotcha:** `setCount(count + 1)` twice in one handler increments by one, not two — both calls read the same snapshot. The updater form `setCount(c => c + 1)` queues functions that each see the pending value, so use it whenever the next value depends on the previous.

useEffect

Dependency array	Setup runs
(omitted)	After every render
`[]`	Once, after the first render
`[a, b]`	After any render where `a` or `b` changed
Cleanup	Before every re-run, and on unmount

An Effect synchronizes a component with an external system — a subscription, a timer, a non-React widget. It runs after the browser paints, and its cleanup runs

before the next setup and on unmount. Every reactive value the Effect reads must appear in the dependency array; removing one to stop a loop hides the bug rather than fixing it.

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

function Ticker() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    const id = setInterval(() => {
      setCount(c => c + 1);
    }, 1000);
    return () => clearInterval(id);
  }, []);
  return <p>{count}</p>;
}
```

Cleanup is not only for unmounting. With a non-empty dependency array it also runs **before each re-run**, and that is what makes debouncing a three-line Effect: every

keystroke cancels the timer the previous keystroke started, so only the last one ever fires.

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

function useDebounced<T>(value: T, ms = 300) {
  const [out, setOut] = useState(value);
  useEffect(() => {
    const id = setTimeout(() => setOut(value),
ms);
    // Cancels the pending timer on every change.
    return () => clearTimeout(id);
  }, [value, ms]);
  return out;
}
```

The input stays instant because it renders `q`; only the debounced copy reaches the Effect that queries the server.

```
import { useState, useEffect } from "react";
declare function useDebounced(v: string): string;
export function Search() {
  const [q, setQ] = useState("");
  const query = useDebounced(q);
  useEffect(() => {
    if (query) fetch(`/search?q=${query}`);
  }, [query]);
  return (
    <input
      value={q}
      onChange={(e) => setQ(e.target.value)}
    />
  );
}
```

> **Gotcha:** In development, Strict Mode runs setup, cleanup, then setup again on mount. A "double" request or a doubled subscription is the point of that, not a bug — it proves the cleanup mirrors the setup, which is what makes the Effect correct in production.

useRef

Use a ref for	Why
A DOM node	Focus, measure, scroll, read a field
A timer or subscription id	Survives renders, must not cause one
The previous value of a prop	Comparison without re-rendering
Anything the user sees	Wrong — that is state

Refs do two different jobs. The first is a **handle to a DOM node**: pass the ref to an element and React writes the

node into `.current` once it is on screen, which is how you focus an input, read an uncontrolled field, scroll something into view, or measure it.

```
import { useRef, useEffect } from "react";
function SearchBox() {
  const input = useRef<HTMLInputElement>(null);
  useEffect(() => {
    input.current?.focus(); // focus on mount
  }, []);
  const read = () => {
    console.log(input.current?.value);
  };
  return (
    <form onSubmit={read}>
      <input ref={input} name="q" />
    </form>
  );
}
```

The second job is a **mutable value that survives re-renders without being one**: a timer id, a WebSocket, a previous value, an “already submitted” flag. A plain variable is recreated on every render and state would re-render on every write, so a ref is the only box that fits — it is the function-component version of an instance field.

```
import { useRef, useState } from "react";
function Stopwatch() {
  const [n, setN] = useState(0);
  const id = useRef<number>(undefined);
  const tick = () => setN((v) => v + 1);
  const pause = () => clearInterval(id.current);
  const start = () => {
    pause(); // never run two timers at once
    id.current = setInterval(tick, 1000);
  };
  return (<><p>{n} seconds</p>
    <button onClick={start}>Start</button>
```

```
    <button onClick={pause}>Pause</button>
  </>);
}
```

The count is state, so it renders; the interval id is a ref, so `pause` can reach the running timer without a render ever having carried it. Clearing on unmount belongs in an Effect cleanup, exactly as the `Ticker` above does it.

Both jobs meet in the click-outside pattern behind every dropdown and modal. The ref holds the panel's DOM node so the listener can ask "was this click inside me?", and the Effect owns the listener: registered on mount, removed in the cleanup. Skip that cleanup and every open-close cycle leaves another listener on `document`, all of them still firing.

```
import { useRef, useEffect, useState } from
"react";
function Menu() {
```

```
  const box = useRef<HTMLDivElement>(null);
  const [open, setOpen] = useState(true);
  useEffect(() => {
    const away = (e: MouseEvent) => {
      const t = e.target as Node;
      if (!box.current?.contains(t))
setOpen(false);
    };
    document.addEventListener("click", away);
    return () =>
      document.removeEventListener("click", away);
  }, []);
  return <div ref={box}>{open && "Menu"}</div>;
}
```

Read and write refs in handlers and Effects, never during render — that is what keeps rendering repeatable. Refs are about **remembering** a value; `useCallback` below is about **identity**, and the two get confused because both survive re-renders.

> **Gotcha:** If the UI must reflect a value, it belongs in state, not a ref. A ref change is invisible until something else happens to re-render the component, which makes the bug look intermittent — the number is right in the console and stale on screen.

useMemo and useCallback

Hook	Hands back	Changes when
`useMemo(fn, deps)`	The value `fn()` returned	A dep changes
`useCallback(fn, deps)`	The function `fn` itself	A dep changes
`useRef(v)`	The same box, always	Never — you assign it

`useCallback(fn, deps)` is exactly `useMemo(() => fn, deps)`: one caches a result, the other caches the function you would have called. Reach for `useMemo` when a computation is measurably expensive,

or when a child needs a stable object or array; reach for `useCallback` only when a function is handed to a `memo()` -wrapped child or listed in an Effect's dependency array, because a fresh function every render defeats both. Reach for `useRef` when you need a stable *slot* to write into rather than a stable *value* to read. None of them is required for correctness — an app should still work with every one deleted.

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

declare function Child(p: {
  items: string[];
  onPick: (s: string) => void;
}): null;
function Parent({ items }: { items: string[] }) {
  // Same array unless items changes.
  const sorted = useMemo(() => [...items],
[items]);
  // Same function on every single render.
```

```
  const onPick = useCallback((s: string) => {
    console.log(s);
  }, []);
  return <Child items={sorted} onPick={onPick} />;
}
```

> **Gotcha:** `useCallback` on its own changes nothing. The child re-renders anyway unless it is wrapped in `memo()`, so the hook only pays off at a memoization boundary — and the React Compiler now inserts most of these for you.

useReducer

Syntax	Meaning
`useReducer(reducer, init)`	A `[state, dispatch]` pair
`dispatch({ type: "…" })`	Sends an action to the reducer

Reach for a reducer once several pieces of state change together, or the same transition is written in more than one handler: one function then owns every transition and each handler only names what happened. The reducer must be pure — it may run twice in development — so build a new state object instead of editing the one you were given.

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

type Action = "inc" | "dec";
function reducer(n: number, action: Action) {
  return action === "inc" ? n + 1 : n - 1;
}
function Counter() {
  const [n, dispatch] = useReducer(reducer, 0);
  return (
    <button onClick={() => dispatch("inc")}>
      {n}
    </button>
```

```
  );
}
```

> **Gotcha:** Dispatching does not update `n` for the rest of the handler, for the same reason `useState` does not — you are reading this render's snapshot. Compute the value you need locally if you also have to act on it now.

Sharing state with context

Piece	What it does
`createContext(default)`	Creates the channel and its fallback
`<Ctx value={v}>`	Publishes `v` to everything below it
`useContext(Ctx)`	Reads the nearest provider above
The provider's state	Where the value actually lives

Context is a channel, not a store. A provider publishes one value to its entire subtree, and any descendant reads it with `useContext` no matter how deep it sits or what components stand in between — it always gets the nearest provider above it. The value itself still lives in some component's state; context only spares you from threading it down by hand. Publish an updater beside the value and one component can change what a completely different component displays, with neither knowing the other exists.

```
import { createContext, useContext } from "react";
const ThemeCtx = createContext({
  theme: "light",
  toggle: () => {},
});
// Displays the value.
function Display() {
  const { theme } = useContext(ThemeCtx);
  return <p>Theme: {theme}</p>;
```

```
}
// Changes the same value.
function Toggle() {
  const { toggle } = useContext(ThemeCtx);
  return <button onClick={toggle}>Switch</button>;
}
```

Threading a prop through layers that never use it is **prop drilling**, and context is not the first cure for it — composition usually is. Rather than `<Layout posts={posts} />` forwarding data inward, pass the finished JSX as children: `<Layout><Posts posts={posts} /></Layout>`. The middle component now takes no prop at all, so it cannot be broken by a change to data it never touches, and the tree still reads top-to-bottom. Context earns its keep for genuinely ambient values — theme, current account, locale, routing — that distant components in different branches all need.

```tsx
import { useState, type Context } from "react";
type Ctx = { theme: string; toggle: () => void };
declare const ThemeCtx: Context<Ctx>;
declare function Display(): null;
declare function Toggle(): null;
export function App() {
  const [theme, setTheme] = useState("light");
  const toggle = () => setTheme("dark");
  return (
    <ThemeCtx value={{ theme, toggle }}>
      <Display />
      <Toggle />
    </ThemeCtx>
  );
}
```

> **Gotcha:** Every component reading a context re-renders when the value changes, `memo()` included. The object literal in `value={{ theme, toggle }}` is new on every render of the provider — harmless while the provider re-renders only for `theme`, a problem the moment it also holds unrelated state. Wrap it in `useMemo` then.

Custom hooks

Rule	Why
Name starts with `use`	Lint and the compiler rely on it
Calls other hooks	That is what makes it a hook
Returns values, not JSX	Sharing logic, not markup

A custom hook is a function that calls other hooks, and the naming convention is what lets the rules of hooks apply to it. It shares stateful *logic*, never state itself: two

components calling the same hook each get their own independent state.

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

function useToggle(initial: boolean) {
  const [on, setOn] = useState(initial);
  return [on, () => setOn(v => !v)] as const;
}
```

> **Gotcha:** Hooks must run in the same order on every render, so they can never sit inside a condition, a loop, or after an early return — only at the top level of a component or another hook. React tracks them by call order, not by name.

Further reading

- React Reference — Hooks
- Rendering Lists
- You Might Not Need an Effect

- Removing Effect Dependencies
- Preserving and Resetting State