

Custom Hooks: Your Own Hooks

Pack repeated logic into your own hook: spotting the pattern, building `useLocalStorage` step by step, and the two rules every hook must obey.

Introduction

You have met React's hooks: `useState`, `useEffect`, `useContext`, `useRef`. A **custom hook** is simply a function **you** write that uses those hooks inside and hands back whatever a component needs — so the same logic (a fetch, a persisted value, a window size) is written once and shared by every component that needs it. (Roman Urdu: *Custom hook aap ka apna hook hai — dohra hua logic aik jagah band karke har component ko de do.*)

Key points

- **A custom hook is a plain function** whose name starts with `use` — `useLocalStorage`, `useFetch`, `useWindowWidth`. The `use` prefix is not decoration: it is how React and the linter know the Rules of Hooks apply inside.
- **It shares logic, not state:** two components using `useLocalStorage("name", "")` each get their **own** state — the recipe is shared, the cooked dish is not. (Shared data is Context's job, lesson 5.7.)
- **Spot the pattern:** when two components copy-paste the same `useState` + `useEffect` dance, that dance wants to become a hook.
(Roman Urdu: *Jahan wohi hook wala code teesri baar copy ho, wahan custom hook banta hai.*)
- **The two Rules of Hooks (now in full):** (1) call hooks **only at the top level** of a component or custom hook — never in loops, conditions or nested functions; (2) call hooks **only from React functions** — components or other custom hooks, never plain helper functions. React matches hooks by call order, so the order must be identical on every render.
- **Return whatever shape fits:** a single value, an array like `useState` (`[value, setValue]`), or an object with named pieces (`{ data, loading, error }`). There is no single required shape.
- **Custom hooks compose:** a hook can call other custom hooks — `useUser` might use `useLocalStorage` inside. Apps grow whole layers this way.
- **The video's build:** `useLocalStorage` — state that survives a page refresh by saving to the browser's `localStorage` (which you met in the JavaScript series).

Code examples

Example 1 — Building `useLocalStorage`, step by step

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

function useLocalStorage(key, startValue) {
  const [value, setValue] = useState(() => {
    const saved = localStorage.getItem(key);
    return saved !== null ? JSON.parse(saved) : startValue;
  });

  useEffect(() => {
    localStorage.setItem(key, JSON.stringify(value));
  }, [key, value]);

  return [value, setValue];
}

function NameSaver() {
  const [name, setName] = useLocalStorage("student-name", "");
  return (
    <input
      value={name}
      onChange={(e) => setName(e.target.value)}
      placeholder="Your name – it survives a refresh"
    />
  );
}
```

What it does: The hook wraps `useState` with two extras: the **lazy initialiser** (the function given to `useState`) reads any saved value on the first render, and an effect writes every change back to `localStorage`.

`NameSaver` uses it exactly like `useState` — the storage machinery is hidden inside the hook.

What you will see: type a name, refresh the page — the name is still there. Plain `useState` would have forgotten it.

Example 2 — One hook, many components: useWindowWidth

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

function useWindowWidth() {
  const [width, setWidth] = useState(window.innerWidth);

  useEffect(() => {
    const onResize = () => setWidth(window.innerWidth);
    window.addEventListener("resize", onResize);
    return () => window.removeEventListener("resize", onResize);
  }, []);

  return width;
}

function LayoutBadge() {
  const width = useWindowWidth();
  return <p>{width < 600 ? "Mobile layout" : "Desktop layout"} – {width}px</p>;
}
```

What it does: The resize listener, its cleanup and the state live in one hook. Any component — a badge, a sidebar, a chart — asks `useWindowWidth()` and gets a live number, with zero listener code of its own.

What you will see: the badge reads "Desktop layout — 1280px" and flips to "Mobile layout" as you narrow the window — and five other components could use the same hook without writing a single listener.

Example 3 — The rules, broken and fixed

```
// BROKEN: hook inside a condition – order changes between renders
function Broken({ showExtra }) {
  if (showExtra) {
    const [extra, setExtra] = useState("");
  }
  const [name, setName] = useState("");
  return null;
}

// FIXED: hooks at the top, conditions afterwards
function Fixed({ showExtra }) {
  const [extra, setExtra] = useState("");
  const [name, setName] = useState("");
  if (!showExtra) return <p>{name}</p>;
  return <p>{name} – {extra}</p>;
}
```

What it does: In the broken version, toggling `showExtra` changes *how many* hooks run, and React — which pairs hooks by order — matches values to the wrong hooks. The fixed version calls every hook unconditionally at the top and puts the condition in the rendering, where conditions belong.

What you will see: the broken component misbehaves or errors when the prop flips; the fixed one behaves identically on every render.

Quick recap

- A **custom hook** is your own `use...` function that packs hooks-based logic once for every component to share — logic is shared, **state is not**.
- The **Rules of Hooks**: top level only, and only from components or other hooks — because React pairs hooks by their call order.
- If you have pasted the same `useState` + `useEffect` pattern twice, it is asking to become a hook.

Common mistakes

1. **Naming it without `use`** — a helper called `getLocalStorage` that calls `useState` breaks the rules invisibly and confuses the linter. The fix: any function that calls hooks is a hook — name it `use...` and follow the rules inside it.
2. **Expecting shared state** — two components calling `useLocalStorage("cart", [])` get two separate states that can disagree. The fix: hooks share *logic*; for one shared value, lift the hook into a parent or into Context.
3. **Hiding a fetch's pieces** — a hook that returns only `data` leaves components unable to show loading or error states. The fix: return the whole story — `{ data, loading, error }` — like the pattern in lesson 5.19.

Practice

1. Write a `useToggle(start)` hook that returns `[on, toggle]` — a boolean and a function that flips it. Use it for a "Show answer" reveal and for a dark-mode switch in the same `App`; notice each use has its own independent boolean.
2. Refactor: take any component from an earlier lesson that reads the window width or saves to `localStorage` inline, and rebuild it on `useWindowWidth` / `useLocalStorage`. The component should end with no `useEffect` of its own for that job.

Watch the Code Scope channel: <https://www.youtube.com/@CodeScopePK> — this lesson's own video joins the channel when it is recorded.

Read online: <https://toolscope.netlify.app/notes/react-custom-hooks> · More free tools and notes at <https://toolscope.netlify.app>