Custom Hooks

What a custom hook is, the use prefix and naming rule, using the same hook more than once, and a useFetch example -- with simple examples.

Intermediate 10 min

Last updated: August 18, 2026
Read online: https://www.learnforgex.com/en/topics/custom-hooks

---

Custom Hooks

In this category we've seen React's built-in hooks: `useState`, `useEffect`, `useRef`, `useMemo`, and `useCallback`. This last lesson covers writing your own hook -- a custom hook.

What Is a Custom Hook?

If you notice yourself writing the same state+effect logic over and over in multiple components, you can extract that logic into a **custom hook**. A custom hook is a regular function you write, that uses React's own hooks (`useState`, `useEffect`, etc.) inside it.

The use Prefix and Naming Rule

The only requirement for a function to count as a custom hook is that its name starts with `use`:

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

// A custom hook is just a NORMAL JavaScript function that starts with "use" --
// inside it, you can call React's own hooks (like useState, useEffect).
function useCounter(initialValue = 0) {
  const [count, setCount] = useState(initialValue);

  function increment() {
    setCount((prev) => prev + 1);
  }

  function decrement() {
    setCount((prev) => prev - 1);
  }

  return { count, increment, decrement };
}

function CustomHookNamingExample() {
  const { count, increment, decrement } = useCounter(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>+1</button>
      <button onClick={decrement}>-1</button>
    </div>
  );
}
```

`useCounter` is a function you wrote that calls `useState` inside -- it's not one of React's own hooks. The `use` prefix signals to both React (for its Rules of Hooks checks) and to other developers reading your code: "this is a hook, and it can call other hooks inside it."

Using the Same Hook More Than Once

You can call a custom hook more than once inside the same component -- each call gets its own INDEPENDENT state:

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

function useCounter(initialValue = 0) {
  const [count, setCount] = useState(initialValue);
  return {
    count,
    increment: () => setCount((prev) => prev + 1),
  };
}

function ReusingCustomHookExample() {
  // We call the same hook twice -- each call has its own INDEPENDENT state.
  const apples = useCounter(0);
  const oranges = useCounter(10);

  return (
    <div>
      <p>Apples: {apples.count}</p>
      <button onClick={apples.increment}>Add Apple</button>

      <p>Oranges: {oranges.count}</p>
      <button onClick={oranges.increment}>Add Orange</button>
    </div>
  );
}
```

`apples` and `oranges` both use the same `useCounter` hook, but they're two completely independent pieces of state. This is the power of custom hooks: you're reusing the LOGIC for managing state, not the state itself.

Example: Fetching Data with useFetch

A custom hook you'll often see in real projects wraps up data-fetching logic -- `useFetch`:

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

// Instead of writing useEffect + useState over and over, we extract the
// "data fetching" logic into a reusable custom hook. (A simplified
// example -- topics like error handling and race conditions will be
// covered in more detail in the "API & Data Fetching" category.)
function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetch(url)
      .then((response) => response.json())
      .then((json) => {
        setData(json);
        setLoading(false);
      });
  }, [url]);

  return { data, loading };
}

function UseFetchExample() {
  const { data, loading } = useFetch("https://api.example.com/users");

  if (loading) {
    return <p>Loading...</p>;
  }

  return <p>{data.length} users found.</p>;
}
```

`useFetch` combines `useState` (for the data and the loading status) with `useEffect` (to fetch the data), gathering them in one place instead of rewriting them in every component. This is a simplified example -- we'll cover things like error handling in more depth later, in the "API & Data Fetching" category.

Summary and Glossary

A custom hook is a regular function you write, starting with `use`, that can use React's own hooks inside it. Extracting repeated state+effect logic into a custom hook lets you reuse that logic across multiple components (or multiple times within the same component) with independent state.

Glossary

Custom Hook — A function you write yourself, starting with `use`, that can use React's own hooks inside it.

Code Reuse — Sharing the same logic from one place (here, a custom hook) instead of rewriting it in multiple places.

Practical Project

There's a real, runnable example project that brings together the concepts from this category (What Are Hooks?, useEffect, useRef, useMemo & useCallback, Custom Hooks): **Hooks Demo** -- a simple stopwatch app with lap tracking.

It shows managing `setInterval` with `useEffect` +cleanup inside a custom hook (`useStopwatch`), using `useRef` both to access a DOM element for auto-scrolling and to keep a persistent counter that doesn't trigger a re-render, recalculating the best lap only when needed with `useMemo` , and keeping a function reference stable with `useCallback` , all working together. You can download it and run it yourself, and read through the code line by line:

```
git clone https://github.com/cdurgun/react-course-projects.git
cd react-course-projects
npm install
cd projects/hooks
npm run dev
```

The `react-course-projects` repo uses **npm workspaces** -- `npm install` only needs to run once, at the repo root, and every project folder shares the same dependencies (no separate `node_modules` per folder). If you've already run `npm install` at the root, you can just `cd react-course-projects/projects/hooks` and run `npm run dev` .