ScopeTreasure Courses

REACT COURSES · LESSON 5.5 · CODE SCOPE SERIES

useEffect: Work After Render

Side effects explained simply: work that runs after render, the three dependency-array shapes, fetching data, the cleanup function, and the infinite-loop trap.

Introduction

Rendering is for drawing the screen from data. But an app also does work that is **not** drawing: fetching data from an API, starting a timer, setting the page title, listening to the window. That work is called a **side effect**, and `useEffect` is the hook that runs it — after the render, at the moments you choose. *(Roman Urdu: useEffect woh kaam karta hai jo screen banane ke BAAD hota hai — data lana, timer chalana, title badalna.)*

Key points

- **Effects run after render:** React first draws the screen from your current state, then runs your effect. The user never waits for the effect to see the page.
- **The dependency array controls when:** `useEffect(fn, [a, b])` re-runs the effect only when `a` or `b` changed since the last render. React compares the values for you.
- **Three shapes to memorise:**
  (Roman Urdu: Array khaali ho to aik baar, values hon to jab woh badlein, array hi na ho to har render par.)
  - **No array** — `useEffect(fn)` runs after **every** render. Rarely what you want.
  - **Empty array** `[]` — runs **once**, after the first render. The classic "on mount" — fetch the initial data here.
  - **Array with values** `[userId]` — runs on mount and again **whenever a listed value changes**.
- **Fetching is an effect:** "when this component appears (or when the id changes), go get the data" is a side effect — the fetch pattern in Example 2 is the one the whole course builds on.
- **Cleanup:** if your effect returns a function, React runs it before the effect runs again and when the component leaves the screen. Timers, intervals and listeners must be cleaned up, or copies pile up.
- **The infinite-loop trap:** an effect that sets state listed in its own dependencies — set state → re-render → effect runs → set state → … The fix is almost always a correct dependency array.
- **Effects are not for everything:** do not fetch in an effect what you can compute during render, and events (clicks) belong in event handlers, not effects.

Code examples

Example 1 — Run once on mount: set the page title

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

function Inbox() {
  const [messages, setMessages] = useState(3);

  useEffect(() => {
    document.title = `Inbox (${messages})`;
  }, [messages]);

  return (
    <div>
      <p>You have {messages} messages.</p>
      <button onClick={() => setMessages(messages + 1)}>New message</button>
    </div>
  );
}
```

What it does: The effect watches `messages`. On the first render and every time the count changes, it rewrites the browser tab's title. The render draws the page; the effect does the outside-world work of touching `document.title`.

What you will see: the tab title reads "Inbox (3)", and each click updates it — "Inbox (4)", "Inbox (5)" — without you ever writing title code inside the click handler.

Example 2 — The fetch pattern: data when the component appears

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

function UserCard({ userId }) {
  const [user, setUser] = useState(null);

  useEffect(() => {
    fetch(`https://jsonplaceholder.typicode.com/users/${userId}`)
      .then((res) => res.json())
      .then((data) => setUser(data));
  }, [userId]);

  if (user === null) return <p>Loading user…</p>;
  return (
    <div className="card">
      <h3>{user.name}</h3>
      <p>{user.email}</p>
    </div>
  );
}
```

What it does: The effect runs on mount and whenever `userId` changes: fetch that user, save the answer into state, and the re-render swaps "Loading user…" for the real card. This shape — state + effect + loading view — is the skeleton of every data-driven screen (and lesson 5.19 builds it fully).

What you will see: "Loading user…" for a moment, then the user's name and email in a card; change the `userId` prop and the card reloads with the new person.

Example 3 — Cleanup: a clock that does not pile up

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

function Clock() {
  const [time, setTime] = useState(new Date());

  useEffect(() => {
    const id = setInterval(() => setTime(new Date()), 1000);
    return () => clearInterval(id);
  }, []);

  return <p>Time: {time.toLocaleTimeString()}</p>;
}
```

What it does: On mount (empty array), the effect starts a 1-second interval that keeps the time fresh. The returned function is the **cleanup**: React runs it when the component leaves the screen, stopping the interval.

Without it, every visit to this page would leave another invisible clock ticking forever.

What you will see: a live clock updating every second — and when you navigate away, it stops cleanly instead of leaking.

Quick recap

- `useEffect` runs side work **after render**: fetching, timers, titles, listeners.
- The **dependency array** is the schedule: no array = every render, `[]` = once on mount, `[values]` = when those values change.
- Return a **cleanup function** for anything that must stop (intervals, listeners) — and fear the loop: an effect that changes its own dependency never rests.

Common mistakes

1. **Forgetting the dependency array** — `useEffect(fn)` with no array runs after every render; if it also sets state, the app loops forever (the video's famous trap). The fix: always write the array on purpose, even when it is empty `[]`.
2. **Lying to the array** — using a value inside the effect but not listing it, so the effect works with stale data. The fix: list everything from the component that the effect reads; if an update feels "one step behind", check this first.
3. **No cleanup for intervals and listeners** — each mount adds another timer or listener, and the app gets stranger and slower. The fix: return a cleanup that clears exactly what the effect started.

Practice

1. Build a `WindowWidth` component: state holds the window's width, an effect with `[]` adds a `resize` listener that updates it, and the cleanup removes the listener. Show "Width: {w}px" and watch it change as you resize the browser.
2. Extend the Example 2 fetch pattern: add an `error` state. If the fetch fails (try a wrong URL), show "Could not load the user — check your connection" instead of the card, and keep the loading state for the waiting moment.

Practice quiz

1. What does useEffect(fn, []) do?

- It runs after every render
- It runs once, after the first render, the classic on mount effect
- It never runs
- It runs before the first render

Answer: It runs once, after the first render, the classic on mount effect

2. What does useEffect(fn, [userId]) do?

- It runs once and never again
- It runs on mount and again whenever userId changes
- It runs after every render
- It runs only when the component leaves the screen

Answer: It runs on mount and again whenever userId changes

3. What is the cleanup function an effect may return for?

- It runs once at the very end of the app
- React runs it before the effect runs again and when the component leaves the screen, so timers, intervals and listeners do not pile up
- It deletes the component
- It is decoration, React ignores it

Answer: React runs it before the effect runs again and when the component leaves the screen, so timers, intervals and listeners do not pile up

---

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://scopetreasure.com/notes/react-useeffect · More free tools and notes at https://scopetreasure.com