useEffect

What a side effect is, how to run one with useEffect, the three uses of the dependency array, the cleanup function, and the common infinite loop mistake -- with simple examples.

Intermediate 14 min

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

---

useEffect

In What Are Hooks?, we saw the concept of a hook. This lesson covers one of the most commonly used hooks -- `useEffect` -- which lets your component reach outside its own render output and do something (a side effect).

What Is a Side Effect?

A **side effect** is something a component does OUTSIDE of its own render output (the JSX it returns): changing the browser tab's title, setting up a timer, fetching data, writing to `localStorage`, and so on. `useEffect` is the hook that lets you do this kind of work safely.

Basic useEffect Usage

You give `useEffect` a function; React runs that function AFTER the render finishes:

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

function BasicUseEffectExample() {
  const [count, setCount] = useState(0);

  // A SIDE EFFECT: something a component does outside of its own render
  // output (JSX) -- here we're changing the browser's tab title.
  useEffect(() => {
    document.title = `Count: ${count}`;
  });

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

Here we update the tab title after every render -- something the JSX itself can't do, because `document.title` is a browser feature outside the rendered component tree.

Dependency Array: An Empty Array []

`useEffect`'s second parameter is an optional **dependency array**. If you pass an empty array `[]`, the effect runs only once, the first time the component appears on screen (mounts):

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

function EmptyDependencyArrayExample() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log("Component appeared on screen for the first time (mounted).");
  }, []); // Empty dependency array: runs only once, after the FIRST render.

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

On later renders (whenever state changes), this effect does NOT run again -- only after the very first render.

Dependency Array: Specific Values

If you put specific values in the dependency array, the effect only runs when THOSE values change:

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

function DependencyArrayExample() {
  const [count, setCount] = useState(0);
  const [name, setName] = useState("Alice");

  useEffect(() => {
    console.log("count changed:", count);
  }, [count]); // Only runs when count changes -- not when name changes.

  return (
    <div>
      <p>
        {name}, count: {count}
      </p>
      <button onClick={() => setCount(count + 1)}>Increment Count</button>
      <button onClick={() => setName(name === "Alice" ? "Bob" : "Alice")}>
        Change Name
      </button>
    </div>
  );
}
```

We wrote `[count]`, so the effect only runs when `count` changes -- it doesn't fire even if `name` changes. On every render, React compares the values in the dependency array to the previous render's values; if at least one changed, it runs the effect.

The Cleanup Function

Some effects (setting up a timer, adding an event listener) leave something behind that needs to be "cleaned up." The function you give `useEffect` can return a CLEANUP function:

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

function CleanupFunctionExample() {
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    const intervalId = setInterval(() => {
      setSeconds((prev) => prev + 1);
    }, 1000);

    // Cleanup function: React automatically calls this when the
    // component is removed from the screen (unmount), or RIGHT BEFORE
    // the effect runs again. If we don't clear the interval here, it
    // keeps running in the background even after the component is
    // removed from the screen -- a "memory leak".
    return () => {
      clearInterval(intervalId);
    };
  }, []);

  return <p>Elapsed time: {seconds} seconds</p>;
}
```

React automatically calls the cleanup function when the component is removed from the screen (unmounts), or right before the effect runs again. If we didn't stop the timer with `clearInterval` here, it would keep running in the background even after the component is gone.

Common Mistake: Infinite Loops

The most common `useEffect` mistake is forgetting the dependency array and updating state inside the effect:

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

function InfiniteLoopMistakeExample() {
  const [count, setCount] = useState(0);

  // WRONG: a useEffect without a dependency array runs after EVERY
  // render. If state is updated inside it, that update triggers a new
  // render, which runs the effect again -- an INFINITE LOOP.
  // useEffect(() => {
  //   setCount(count + 1);
  // });

  // RIGHT: make the dependency array [] so it only runs on the first render.
  useEffect(() => {
    setCount((prev) => prev + 1);
  }, []);

  return <p>Count (should only increase once): {count}</p>;
}
```

A `useEffect` without a dependency array runs after EVERY render. If it updates state inside, that update triggers a new render, and that render runs the effect again -- an INFINITE LOOP. The fix is setting up the dependency array correctly: `[]` if it should only run once, or the specific value in the array if it depends on that value.

Summary and Glossary

`useEffect` is used when a component needs to do something (a side effect) outside its render output. The dependency array decides WHEN the effect runs: `[]` only on the first render, `[value]` only when that value changes, and no dependency array at all means after EVERY render. The cleanup function cleans up whatever the effect left behind (a timer, an event listener, etc.). Forgetting the dependency array and updating state inside the effect is the most common mistake, causing an infinite loop.

Glossary

Side Effect — Something a component does outside its render output (directly changing the DOM, fetching data, etc.).

`useEffect` — The hook that gives a component the ability to run side effects.

Dependency Array — `useEffect`'s second parameter; decides which values, when changed, cause the effect to run again.

Cleanup Function — The function returned from the function passed to `useEffect`, which cleans up whatever the effect left behind; called when the component unmounts or before the effect runs again.

Mount / Unmount — A component appearing on screen for the first time (mount), and being removed from the screen entirely (unmount).