In React, **Hooks** are special functions that let you "hook into" React features like state, lifecycle methods, and context—without writing a class component. They were introduced in React 16.8 and have since become the standard for writing modern, functional components.

Here are some of the most commonly used hooks:

- **useState** – Adds state to functional components.
- **useEffect** – Handles side effects like data fetching, subscriptions, or manually changing the DOM.
- **useContext** – Accesses context values without needing a wrapper component.
- **useRef** – Persists values across renders without triggering re-renders (often used for DOM references).
- **useReducer** – An alternative to useState for complex state logic.
- **useMemo** and **useCallback** – Optimize performance by memoizing values or functions.

For example, here's a simple use of useState:

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

function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  );
}
```

Hooks must follow a few rules:

1. Only call hooks at the top level of your component.
2. Only call hooks from React functions (not regular JS functions).
3. Custom hooks should start with use.

The **useEffect** hook in React lets you perform **side effects** in function components—things like fetching data, setting up subscriptions, or manually updating the DOM. It's React's way of saying, "Hey, do something after rendering."

Basic Syntax

```
useEffect(() => {
```

```
  // Your side effect logic here


  return () => {
    // Optional cleanup logic
  };
}, [dependencies]);
```

How It Works

- **No dependency array**: Runs after every render.
- **Empty array []**: Runs only once after the initial render (like componentDidMount).
- **With dependencies**: Runs after the initial render and whenever any dependency changes.

Example: Updating the document title

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


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


  useEffect(() => {
    document.title = `Clicked ${count} times`;
  }, [count]);


  return <button onClick={() => setCount(count + 1)}>Click me</button>;
}
```

When you use useEffect **without a dependency array**, it runs **after every render**—initial and subsequent ones. This can be useful in rare cases, but it often leads to performance issues or unintended behavior if not handled carefully.

Example: No dependency array

```
useEffect(() => {
  console.log("Effect ran");
});
```

This will log "Effect ran" after **every render**, including when state or props change.

Why it matters

React re-renders components when state or props change. Without a dependency array, useEffect runs each time, which might:

- Trigger unnecessary API calls
- Cause infinite loops if the effect updates state
- Slow down your app

When it's okay

You might skip the dependency array if:

- You want to run code after **every** render (e.g., logging or syncing with non-React systems)
- You're experimenting or debugging

But in most cases, it's better to **include a dependency array** to control when the effect runs.

Want to see what happens if we update state inside such an effect? It's a classic infinite loop trap—I can show you that too if you're curious.

In React, you **can't conditionally call useEffect**—but you *can* include conditions *inside* the effect's callback. This is a subtle but important distinction.

❌ **What you should avoid:**

```
if (someCondition) {
  useEffect(() => {
    // This is invalid!
  }, []);
}
```

React expects hooks to be called in the same order on every render. Conditional calls break that rule and can cause bugs.

✅ **What you should do instead:**

Call useEffect unconditionally, and put your logic inside it:

```
useEffect(() => {
  if (someCondition) {
    // Run your side effect here
  }
}, [someCondition]);
```

This way, the hook is always called, but the effect only runs when the condition is met.

Example: Fetch data only when shouldFetch is true

```
useEffect(() => {
  if (!shouldFetch) return;

  fetch("https://api.example.com/data")
    .then((res) => res.json())
    .then((data) => setData(data));
}, [shouldFetch]);
```

You can even optimize further by computing the condition outside:

```
const shouldFetchData = someFlag && !dataLoaded;

useEffect(() => {
  if (shouldFetchData) {
    // fetch logic
  }
}, [shouldFetchData]);
```