When Should We Use `useEffect` in React?

`useEffect` is a powerful hook in React that helps perform **side effects** in function components. It runs after rendering and can be triggered by changes in dependencies.

📌 Scenarios Where `useEffect` is Necessary

1 Fetching Data from an API (Side Effects)

When you need to fetch data **once** when a component mounts, `useEffect` ensures it only happens at the right time.

```
jsx
CopyEdit
useEffect(() => {
  fetch("https://api.example.com/data")
    .then(res => res.json())
    .then(data => setData(data));
}, []); // Runs only once when the component mounts
```

✅ **Why?**

- Prevents unnecessary API calls on every render.

---

2 Updating State Based on External Changes

If you need to update state when **certain props or variables change**, `useEffect` helps manage that.

```
jsx
CopyEdit
useEffect(() => {
  setFormattedPrice(price.toFixed(2));
}, [price]); // Runs when 'price' changes
```

✅ **Why?**

- Ensures the state updates only when `price` changes.

---

3 Handling Subscriptions, Timers, or Event Listeners

When using **intervals, WebSockets, or event listeners**, `useEffect` is used to set them up and clean them up properly.

```
jsx
CopyEdit
useEffect(() => {
  const interval = setInterval(() => {
    console.log("Updating...");
  }, 1000);

  return () => clearInterval(interval); // Cleanup function
}, []); // Runs only once on mount
```

✅ **Why?**

- Ensures resources (like intervals) are cleaned up to prevent memory leaks.

---

4 Syncing with Local Storage or External Services

If you want to **store data in local storage** when a value changes, `useEffect` ensures that happens only when needed.

```
jsx
CopyEdit
useEffect(() => {
  localStorage.setItem("theme", theme);
}, [theme]); // Runs only when 'theme' changes
```

✅ **Why?**

- Prevents unnecessary local storage updates.

---

5 Triggering Functions on Component Unmount

When a component is removed, `useEffect` helps **clean up** resources like network calls, subscriptions, or event listeners.

```
jsx
CopyEdit
useEffect(() => {
  window.addEventListener("resize", handleResize);

  return () => {
    window.removeEventListener("resize", handleResize); // Cleanup
  };
}, []); // Runs only once when mounted
```

✅ **Why?**

- Prevents memory leaks and unnecessary event listeners.

🚀 When NOT to Use `useEffect`

🚫 Avoid `useEffect` when simple event handlers or direct function calls are enough.

- ✅ **Good:** Directly calling functions inside `onChange`, `onClick`, etc.
- ❌ **Bad:** Using `useEffect` for calculations that can be done inline.

Example: Avoid Unnecessary `useEffect`

Instead of:

```
jsx
CopyEdit
useEffect(() => {
  setSum(a + b);
}, [a, b]);
```

Just do:

```
jsx
CopyEdit
const sum = a + b;
```

✨ Conclusion

✅ Use `useEffect` when:

- You need to fetch data from an API.
- You want to sync a state with local storage or an external system.
- You need to set up event listeners, timers, or subscriptions.
- You want to clean up resources when a component unmounts.

❌ Don't use `useEffect` when:

- You can directly compute values in event handlers or inside render logic.

🎯 Key Rule:

👉 If something **must happen outside React's rendering process** (e.g., API calls, subscriptions, storage updates), use `useEffect`.
👉 If you just need calculations based on state, avoid `useEffect` and use inline functions or derived state.

What is `useEffect` Really Doing?

Think of `useEffect` as a way to **run side effects** in a React component. A side effect is anything that **happens outside** the normal rendering process, like: ✅ Fetching data
✅ Updating the DOM manually
✅ Setting up timers or event listeners
✅ Saving data to local storage

🛠️ `useEffect` Syntax and Execution Rules

```
jsx
CopyEdit
useEffect(() => {
  // 🛠️ Code to run (side effect)
  return () => {
    // 🧹 Cleanup function (optional)
  };
}, [dependencies]); // 🔁 Dependency array
```

How It Works:

Scenario	When It Runs
`useEffect(() => {...}, [])`	Runs **once** when the component mounts (like `componentDidMount`)
`useEffect(() => {...}, [someValue])`	Runs **when `someValue` changes** (like `componentDidUpdate`)
`useEffect(() => {...})`	Runs **after every render** (not recommended, as it causes unnecessary re-renders)
`useEffect(() => {... return () => {...} }, [])`	Runs **on mount**, and cleanup runs on **unmount** (like `componentWillUnmount`)

🔥 Example 1: Fetching Data (Side Effect)

💡 **Scenario:** You want to fetch user data from an API when the component first loads.

```
jsx
CopyEdit
import React, { useState, useEffect } from 'react';

const UserProfile = () => {
  const [user, setUser] = useState(null);

  useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/users/1')
      .then((response) => response.json())
      .then((data) => setUser(data));
  }, []); // Runs once on mount

  return (
```

```
    <div>
      <h1>User Profile</h1>
      {user ? <p>{user.name}</p> : <p>Loading...</p>}
    </div>
  );
};
export default UserProfile;
```

✅ **Why `useEffect`?**

- The API call **should not** be made on every render.
- We only want it to run **once** when the component mounts.

🔥 Example 2: Updating Document Title

💡 **Scenario:** You want to update the tab title based on a counter.

```
jsx
CopyEdit
import React, { useState, useEffect } from 'react';

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

  useEffect(() => {
    document.title = `Count: ${count}`;
  }, [count]); // Runs when `count` changes

  return (
    <div>
      <h1>{count}</h1>
      <button onClick={() => setCount(count + 1)}>Increase</button>
    </div>
  );
};
export default Counter;
```

✅ **Why `useEffect`?**

- The document title **needs to update** only when `count` changes, **not on every render**.

🔥 Example 3: Handling Event Listeners (Cleanup Required)

💡 **Scenario:** You want to track the mouse position **only when the component is mounted** and remove the event listener when it unmounts.

```
jsx
```

```
CopyEdit
import React, { useState, useEffect } from 'react';

const MouseTracker = () => {
  const [position, setPosition] = useState({ x: 0, y: 0 });

  useEffect(() => {
    const handleMouseMove = (e) => {
      setPosition({ x: e.clientX, y: e.clientY });
    };

    window.addEventListener('mousemove', handleMouseMove);

    return () => {
      window.removeEventListener('mousemove', handleMouseMove); // Cleanup
on unmount
    };
  }, []);

  return (
    <div>
      <h1>Mouse Position: {position.x}, {position.y}</h1>
    </div>
  );
};
export default MouseTracker;
```

✅ **Why `useEffect`?**

- **Setup:** The event listener should be added when the component mounts.
- **Cleanup:** The event listener should be **removed when the component unmounts** to avoid memory leaks.

❌ Common Mistakes with `useEffect`

1 Infinite Re-renders (No Dependency Array)

❌ Bad Code:

```
jsx
CopyEdit
useEffect(() => {
  console.log("Runs after every render!");
});
```

🚨 **Problem:** Runs **after every render**, causing an infinite loop in some cases.

✅ Fix:

```
jsx
CopyEdit
useEffect(() => {
  console.log("Runs only once!");
```

```
}, []);
```

---

2 Missing Cleanup for Event Listeners

❌ **Bad Code (Leaks Memory):**

```
jsx
CopyEdit
useEffect(() => {
  window.addEventListener("scroll", () => console.log("Scrolling..."));
}, []);
```

🚨 **Problem:** The event listener **never gets removed**, even when the component unmounts.

✅ **Fix:**

```
jsx
CopyEdit
useEffect(() => {
  const handleScroll = () => console.log("Scrolling...");
  window.addEventListener("scroll", handleScroll);

  return () => {
    window.removeEventListener("scroll", handleScroll); // Cleanup
  };
}, []);
```

---

3 Fetching Data Without Dependencies

❌ **Bad Code (Makes Unnecessary Calls):**

```
jsx
CopyEdit
useEffect(() => {
  fetchData();
});
```

🚨 **Problem:** This will fetch data **on every render**, which is inefficient.

✅ **Fix:**

```
jsx
CopyEdit
useEffect(() => {
  fetchData();
}, []); // Run only once
```

💡 When to Use `useEffect` vs When Not to

Use `useEffect` When...	Don't Use `useEffect` When...
You need to fetch data from an API	You can compute values directly in JSX
You want to add/remove event listeners	You can handle logic inside an event handler
You need to interact with the browser (e.g., update the title)	You are just deriving a new state value
You want to trigger effects when a state/prop changes	You are handling simple calculations inside JSX

🎯 Final Takeaway

🚀 **Use `useEffect` to handle side effects like API calls, event listeners, and subscriptions.**
🚀 **Always clean up resources in `useEffect` to avoid memory leaks.**
🚀 **Avoid unnecessary re-renders by using the dependency array correctly.**