**🧩 What is useEffect?**

useEffect is a **side-effect hook** in React.

🧠 “Side effect” = anything that **affects something outside React’s rendering** —  
like calling an API, updating localStorage, setting up event listeners, or manually manipulating the DOM.

**🔍 Syntax**

useEffect(() => {

// side effect code (runs after render)

return () => {

// cleanup code (runs before re-run or unmount)

};

}, [dependencies]);

**⚙️ When It Runs**

| **Dependency** | **When It Runs** | **Cleanup Timing** |
| --- | --- | --- |
| [] (empty) | Once after first render (mount) | When component unmounts |
| [a, b] | After first render + whenever a or b changes | Before next run or unmount |
| No array | After every render | Before every next render |

**🎯 Real-Life Analogy**

Imagine you are **mounting a CCTV camera** (component).

* useEffect is the process where you **set up things after the camera is installed**, like connecting it to power or internet.
* The **cleanup** is like **unplugging or resetting the camera** before reconfiguring or removing it.

**⚡ Common Interview Question Patterns**

| **Question** | **Explanation** |
| --- | --- |
| What does useEffect(() => {}, []) do? | Runs only once after the first render — good for API calls. |
| What happens if you omit the dependency array? | It runs **after every render** → may cause infinite loops if you set state inside it. |
| How to clean up side effects? | Return a cleanup function → runs before component unmount or next effect run. |
| Can we make useEffect async? | Not directly. Define an async function **inside** the effect and call it. |
| What causes an infinite loop in useEffect? | When you update a state inside an effect **without properly handling dependencies**, causing it to re-trigger itself. |
| What’s the difference between useEffect and useLayoutEffect? | useLayoutEffect runs **synchronously** after DOM updates but **before painting**, while useEffect runs **after painting** (non-blocking). |

**🧠 In Short**

| **Concept** | **Description** |
| --- | --- |
| 🪄 Purpose | Handle side effects (API calls, subscriptions, event listeners, etc.) |
| ⚙️ Timing | Runs **after render** |
| 🔁 Dependencies | Controls *when* the effect re-runs |
| 🧹 Cleanup | Prevents memory leaks (unsubscribe, clear intervals, etc.) |
| 🚫 Common mistake | Forgetting dependency array → causes infinite loops |

see its like if we are calling api using useeffect without using [ ] then whenevr i try to do any different chnages in that screen it will rerender againg this api also but if we use this [ ] then no matter what chaagese i doo it wont rerender this api also if i type specifi [searchterm] then it will rerender only when i try to search an name

**🧠 Scenario 1: No dependency array → useEffect(() => { ... })**

useEffect(() => {

fetchUsers(); // 👈 API call

});

**🔄 Behavior:**

* Runs **after every single render**.
* That means if you:
  + type in an input
  + change a state
  + click a button
  + update anything on the screen

→ it will **re-run the API call again** 😬 (which can cause an **infinite loop** if that API updates state).

**⚠️ Example:**

useEffect(() => {

fetch("https://api.example.com/users")

.then(res => res.json())

.then(data => setUsers(data)); // ❌ this causes re-render

});

Here, setUsers() triggers a re-render → useEffect runs again → API is called again → infinite loop 🚨

**🧠 Scenario 2: Empty dependency array → useEffect(() => { ... }, [])**

useEffect(() => {

fetchUsers();

}, []);

**✅ Behavior:**

* Runs **only once** — right after the first render (mount).
* After that, **no matter what changes** you make (input typing, state updates, etc.), it **won’t re-run**.

💡 Perfect for **initial API calls**, e.g., fetching dashboard data, profile info, etc.

**🧩 Example:**

useEffect(() => {

fetch("https://api.example.com/users")

.then(res => res.json())

.then(data => setUsers(data));

}, []); // ✅ Runs once

**🧠 Scenario 3: With dependencies → useEffect(() => { ... }, [searchTerm])**

useEffect(() => {

fetchUsers(searchTerm);

}, [searchTerm]);

**⚙️ Behavior:**

* Runs **once on mount** and **whenever searchTerm changes**.
* So:
  + When the component loads → API runs once ✅
  + When user types “Veda” → API runs again ✅
  + When user types “Deva” → API runs again ✅
  + When user updates some *other state* (like theme or sidebar toggle) → ❌ API **does not run again**

**✅ Example:**

useEffect(() => {

if (!searchTerm) return;

fetch(`https://api.example.com/users?q=${searchTerm}`)

.then(res => res.json())

.then(data => setUsers(data));

}, [searchTerm]);

**what happens if we use [] instead of [searchTerm]**.

**1️⃣ With [searchTerm]**

useEffect(() => {

fetchUsers(searchTerm);

}, [searchTerm]);

* Runs **once on mount**
* Runs **every time searchTerm changes**
* Does **not run** if **other state changes**

**2️⃣ With [] (empty dependency array)**

useEffect(() => {

fetchUsers(searchTerm);

}, []);

**Behavior:**

1. **Runs only once** — right after **first render / mount**
2. **Never runs again**, even if searchTerm changes
3. **Does not react to any state changes**, including searchTerm

**Example Flow:**

| **Action** | **API Call?** |
| --- | --- |
| Component mounts | ✅ Runs once |
| Type “Veda” | ❌ Does NOT run |
| Type “Deva” | ❌ Does NOT run |
| Update sidebar or theme | ❌ Does NOT run |

**⚡ Key Difference**

| **Dependency** | **Runs On** | **Use Case** |
| --- | --- | --- |
| [searchTerm] | Mount + whenever searchTerm changes | Dynamic search/filter |
| [] | Mount only | Initial data fetch (dashboard, profile, etc.) |

So basically:

* [searchTerm] → “Run this effect when THIS thing changes”
* [] → “Run this effect only ONCE when component mounts”

**Why would we need to change data for searchTerm?**

When you’re building a **search or filter functionality**, the data you display depends on what the user types.

Think of it like this:

* You have a **list of users**:

data = [

{name: "Veda"},

{name: "Deva"},

{name: "Reva"}

]

* Initially, you show all names.
* User types "Veda" in the search box.

Now you want to **update the displayed data** to only show "Veda".

useEffect(() => {

const filtered = data.filter(user => user.name.includes(searchTerm));

setFilteredData(filtered);

}, [searchTerm]);

✅ Here, the **effect depends on searchTerm**:

* If user types something new → effect runs → filtered data updates → UI re-renders
* If user doesn’t change searchTerm → effect doesn’t run → no unnecessary work

**Without using [searchTerm]**

useEffect(() => {

const filtered = data.filter(user => user.name.includes(searchTerm));

setFilteredData(filtered);

}, []); // empty dependency

* Effect runs **only once on mount**
* searchTerm changes later → **effect does NOT run**
* Filtered data **does not update** → UI still shows the initial data
* **Search functionality breaks**

**Simple Analogy**

Think of a **filter on a water tap**:

* searchTerm = the shape of the filter
* useEffect = the tap that applies the filter
* [searchTerm] = “Run the filter whenever the filter shape changes”
* [] = “Run the filter only once when the tap is first turned on”

If you used [], the filter shape changing later (typing in search) wouldn’t do anything — wrong output.

So basically, **we need [searchTerm]** to ensure the displayed data **always matches what the user typed**.