React useEffect – Complete Concept Guide

This document is a **complete, structured guide to the `useEffect` hook**, covering everything discussed in your shared material: lifecycle mapping, side effects, dependency arrays, cleanup, data fetching, and rules of hooks.

---

1. What is `useEffect`?

`useEffect` is a **React Hook** that lets you run **side effects** in functional components.

Side effects are operations that: - Interact with external systems - Cannot run during render - Happen *after* React updates the UI

Examples of side effects: - API calls - Timers (`setTimeout`, `setInterval`) - Event listeners - DOM manipulation - Subscriptions (WebSocket, scroll, resize)

---

2. Component Lifecycle (Conceptual Model)

Although functional components don't have lifecycle methods like class components, React components still go through **three phases**:

1. Mounting

- Component is created and added to the DOM
- First render happens

2. Updating

- Happens when **state or props change**
- Component re-renders

3. Unmounting

- Component is removed from the DOM
- Cleanup happens here

`useEffect` allows us to handle **all three phases with one unified API**.

---

3. Basic Syntax of `useEffect`

```
useEffect(() => {
  // side effect code

  return () => {
    // cleanup code (optional)
  };
}, [dependencies]);
```

Parameters

1. **Effect function** → runs after render
2. **Dependency array (optional)** → controls *when* the effect runs

---

4. When Does `useEffect` Run?

Case 1: Empty Dependency Array (`[]`)

Runs **only once** after initial render (mounting).

```
useEffect(() => {
  console.log("Component mounted");
}, []);
```

Used for: - Initial API calls - One-time setup

Case 2: Dependency Array with Values (`[count]`)

Runs: - After first render - Whenever **any dependency changes**

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

Used for: - Syncing state with external systems - Reacting to specific state/prop changes

---

Case 3: No Dependency Array

Runs **after every render**.

```
useEffect(() => {
  console.log("Component updated");
});
```

⚠️Use carefully — can cause performance issues.

---

5. Understanding Side Effects

Side effects: - Happen **after render** - Do not block UI rendering - Keep React UI and external systems in sync

Common Side Effects

- Fetching data
- Updating document title
- Logging
- Timers
- Event listeners

---

6. Practical Example – Counter with `useEffect`

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

  useEffect(() => {
    console.log("useEffect run", count);
    document.title = `You clicked ${count} times`;
  }, [count]);

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

What Happens Here?

- Component renders
- `useEffect` runs **after render**
- Runs again whenever `count` changes

---

7. Cleaning Up Effects (Very Important)

Some effects **must be cleaned up** to avoid memory leaks.

Example: Timer Cleanup

```
useEffect(() => {
  const timer = setInterval(() => {
    console.log("Tick");
  }, 1000);
  return () => {
    clearInterval(timer);
  };
}, []);
```

Cleanup runs: - Before component unmounts - Before the effect re-runs

---

8. Data Fetching with `useEffect`

```
function FetchDataComponent() {
  const [data, setData] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      const response = await fetch(
        "https://jsonplaceholder.typicode.com/users"
      );
      const result = await response.json();
      setData(result);
    };

    fetchData();
  }, []);

  return (
    <div>
      {data ? <p>{data[0].name}</p> : <p>Loading...</p>}
    </div>
  );
}
```

Key Points

- Effect runs once
- Fetch happens after render
- State update triggers re-render

---

9. Dependency Array – Summary Table

Dependency Array	Behavior
`[]`	Runs once on mount
`[value]`	Runs when value changes
omitted	Runs after every render

10. Mental Model: `useEffect` as a Reminder

Think of `useEffect` as telling React:

> "After rendering, remind me to run this code when these things change."

---

11. Rules of Using Hooks (Very Important)

Rule 1: Call Hooks at the Top Level

❌Not inside loops, conditions, or nested functions

Rule 2: Call Hooks Only from React Functions

- Function components
- Custom hooks

Rule 3: Custom Hooks Must Follow Same Rules

- Must start with `use`
- Can call other hooks

---

12. Why These Rules Exist

React relies on **hook call order** to preserve state. Breaking the rules can cause: - Bugs - State mismatch - Unexpected behavior

---

13. When NOT to Use `useEffect`

❌For pure calculations ❌For derived state ❌For logic that can be done during render

---

14. Real-World Patterns with `useEffect`

Pattern 1: Data Fetching on Mount

Use case: Load data when a page loads

```
useEffect(() => {
  fetchData();
}, []);
```

✔️Correct because: - Runs once - Avoids unnecessary re-fetching

---

Pattern 2: Sync State with External Systems

Use case: Update document title, localStorage, analytics

```
useEffect(() => {
  localStorage.setItem("theme", theme);
}, [theme]);
```

✔️Keeps React state and external systems in sync

Pattern 3: Event Listeners with Cleanup

Use case: Window resize, scroll, keyboard events

```
useEffect(() => {
  const handleResize = () => console.log(window.innerWidth);
  window.addEventListener("resize", handleResize);

  return () => {
    window.removeEventListener("resize", handleResize);
  };
}, []);
```

✔️Prevents memory leaks

Pattern 4: Timers & Intervals

Use case: Polling, clocks, animations

```
useEffect(() => {
  const id = setInterval(() => {
    setTime(Date.now());
  }, 1000);

  return () => clearInterval(id);
}, []);
```

✔️Always clean up intervals

Pattern 5: Dependent Side Effects

Use case: Fetch data when a filter or ID changes

```
useEffect(() => {
  fetchUser(userId);
}, [userId]);
```

✔️Runs only when relevant data changes

---

15. Anti-Patterns (What NOT to Do)

❌Anti-Pattern 1: Missing Dependency Array

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

🚫Problem: - Runs on every render - Causes unnecessary API calls

---

❌Anti-Pattern 2: Incorrect Dependencies

```
useEffect(() => {
  console.log(count);
}, []);
```

🚫Problem: - Uses `count` but doesn’t list it - Leads to stale values

---

❌Anti-Pattern 3: Infinite Loops

```
useEffect(() => {
  setCount(count + 1);
}, [count]);
```

🚫Problem: - Effect triggers state update - State update retriggers effect forever

---

❌Anti-Pattern 4: Using `useEffect` for Derived State

```
useEffect(() => {
  setFullName(first + " " + last);
}, [first, last]);
```

🚫Wrong approach

✔️Better:

```
const fullName = first + " " + last;
```

❌Anti-Pattern 5: Side Effects in Render

```
if (count > 5) {
  alert("Too high");
}
```

🚫Causes repeated alerts

✔️Correct:

```
useEffect(() => {
  if (count > 5) alert("Too high");
}, [count]);
```

---

16. When You Should Pause Before Using `useEffect`

Ask yourself: - Can this be computed during render? - Is this derived from existing state? - Does this really talk to an external system?

If **NO external system is involved**, you probably don't need `useEffect`.

---

17. Comprehensive Real-World Example: Fetch + Table + Filter + Sort + Pagination

Below is a **complete real-world example** combining: - API data fetching using `useEffect` - Displaying data in a table - Filtering - Sorting - Pagination

This is a **very common interview + production pattern**.

Scenario

We will: - Fetch users from an API - Display them in a table - Filter users by name - Sort users by name - Paginate the results

---

Full Example Code

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

function UsersTable() {
  const [users, setUsers] = useState([]);        // original data
  const [search, setSearch] = useState("");      // filter state
  const [sortOrder, setSortOrder] = useState("asc");
  const [currentPage, setCurrentPage] = useState(1);

  const USERS_PER_PAGE = 5;

  // 1️⃣ Fetch data on mount
  useEffect(() => {
    const fetchUsers = async () => {
      const response = await fetch(
        "https://jsonplaceholder.typicode.com/users"
      );
      const data = await response.json();
      setUsers(data);
    };

    fetchUsers();
  }, []);

  // 2️⃣ Filter logic
  const filteredUsers = users.filter((user) =>
    user.name.toLowerCase().includes(search.toLowerCase())
  );

  // 3️⃣ Sorting logic
  const sortedUsers = [...filteredUsers].sort((a, b) => {
    if (sortOrder === "asc") {
      return a.name.localeCompare(b.name);
    }
    return b.name.localeCompare(a.name);
  });

  // 4️⃣ Pagination logic
  const startIndex = (currentPage - 1) * USERS_PER_PAGE;
  const paginatedUsers = sortedUsers.slice(
    startIndex,
    startIndex + USERS_PER_PAGE
  );

  const totalPages = Math.ceil(sortedUsers.length / USERS_PER_PAGE);

  return (
    <div>
      <h2>User List</h2>
```

```
{/* Filter */}
<input
  type="text"
  placeholder="Search by name"
  value={search}
  onChange={(e) => {
    setSearch(e.target.value);
    setCurrentPage(1); // reset page on filter
  }}
/>

{/* Sort */}
<button onClick={() => setSortOrder("asc")}>Sort A-Z</button>
<button onClick={() => setSortOrder("desc")}>Sort Z-A</button>

{/* Table */}
<table border="1" cellPadding="8">
  <thead>
    <tr>
      <th>Name</th>
      <th>Email</th>
      <th>City</th>
    </tr>
  </thead>
  <tbody>
    {paginatedUsers.map((user) => (
      <tr key={user.id}>
        <td>{user.name}</td>
        <td>{user.email}</td>
        <td>{user.address.city}</td>
      </tr>
    ))}
  </tbody>
</table>

{/* Pagination */}
<div style={{ marginTop: "10px" }}>
  <button
    disabled={currentPage === 1}
    onClick={() => setCurrentPage((p) => p - 1)}
  >
    Prev
  </button>

  <span> Page {currentPage} of {totalPages} </span>

  <button
    disabled={currentPage === totalPages}
    onClick={() => setCurrentPage((p) => p + 1)}
  >
    Next
```

```
        </button>
      </div>
    </div>
  );
}

export default UsersTable;
```

How `useEffect` Is Used Correctly Here

✔️ Data Fetching

- API call happens **once on mount**
- Dependency array is empty (`[]`)

✔️ Separation of Concerns

- `useEffect` → only for side effects (fetching)
- Filtering, sorting, pagination → pure logic (no effects)

✔️ No Anti-Patterns

- No infinite loops
- No unnecessary effects
- No derived state stored in state

Why This Example Is Important

This single example demonstrates: - Real production usage of `useEffect` - Correct mental model - Performance-friendly design - Clean React architecture

If you understand **this example**, you understand **80% of real-world `useEffect` usage**.

---

18. Final Takeaways

- Fetch data in `useEffect`
- Keep UI logic outside `useEffect`
- Filter, sort, paginate using pure functions
- Reset pagination when filters change
- `useEffect` should stay **small and focused**

Master this pattern and you are production-ready 🚯