# useEffect Complete Guide - English & Tamil

---

## 1. What is useEffect? (useEffect என்றால் என்ன?)

**English:** useEffect is a React Hook that lets you perform side effects in functional components. Side effects are operations that interact with the outside world like API calls, DOM manipulation, or subscriptions.

**Tamil:** useEffect என்பது React Hook ஆகும். இது functional components-ல் side effects செய்ய உதவுகிறது. Side effects என்பது API calls, DOM manipulation, subscriptions போன்ற வெளி உலகத்துடன் இடம்பெறும் செயல்களாகும்.

---

## 2. useEffect vs Lifecycle Methods Comparison

### Class Components (Old Way - Lifecycle Methods)

```javascript
class Counter extends React.Component {
  componentDidMount() {
    // Runs once after component mounts
    console.log("Component mounted");
  }

  componentDidUpdate(prevProps, prevState) {
    // Runs after every render or when specific props change
    console.log("Component updated");
  }

  componentWillUnmount() {
    // Cleanup before component unmounts
    console.log("Component will unmount");
  }

  render() {
    return <div>Count: {this.state.count}</div>;
  }
}
```

### Functional Components (New Way - useEffect)

```javascript
```

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

  // Replaces ALL three lifecycle methods
  useEffect(() => {
    console.log("Component mounted or updated");

    // Cleanup function (replaces componentWillUnmount)
    return () => {
      console.log("Cleanup before unmount or re-run");
    };
  }, []); // Empty dependency array

  return <div>Count: {count}</div>;
}
```

### Comparison Table

| Lifecycle Method | useEffect Equivalent | When it runs |
|---|---|---|
| `componentDidMount` | `useEffect(() => {...}, [])` | Once after mount |
| `componentDidUpdate` | `useEffect(() => {...}, [dep1, dep2])` | After deps change |
| `componentWillUnmount` | `return () => {...}` inside useEffect | Before unmount |
| Both Mount & Update | `useEffect(() => {...})` | After every render |

---

## 3. useEffect Hook - Detailed Explanation

### Syntax

```javascript
useEffect(
  () => {
    // Side effect code runs here

    return () => {
      // Cleanup function (optional)
    };
  },
  [dependencies] // Optional dependency array
);
```

### Different Scenarios

#### Scenario 1: No Dependency Array (Runs After Every Render)

```javascript
useEffect(() => {
  console.log("Runs after EVERY render");
});

// Results in:
// Mount → Render 1 → Effect runs
// User clicks button → Render 2 → Effect runs
// User clicks button → Render 3 → Effect runs
```

### Scenario 2: Empty Dependency Array (Runs Only Once on Mount)

```javascript
useEffect(() => {
  console.log("Runs ONLY ONCE when component mounts");

  // Perfect for API calls, initialization
  fetchUserData();

}, []);
```

### Scenario 3: Dependency Array with Values (Runs When Dependencies Change)

```javascript
const [userId, setUserId] = useState(1);

useEffect(() => {
  console.log("Runs when userId changes");
  fetchUserData(userId);

}, [userId]); // Only re-run when userId changes
```

---

## 4. Understanding Cleanup Function (Interval Clearing)

### Why Do We Need Cleanup?

When you set up subscriptions, timers, or event listeners, they can cause:

- Memory leaks
- Multiple subscriptions running simultaneously
- Unexpected behavior

### Example: Timer Without Cleanup ❌ (WRONG)

javascript

```javascript
function Timer() {
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    setInterval(() => {
      setSeconds(prev => prev + 1);
    }, 1000);
  }, []); // Problem: New interval created every time!

  return <div>Seconds: {seconds}</div>;
}

// What happens:
// Component mounts → Interval 1 created
// Component re-renders → Interval 2 created
// Component re-renders → Interval 3 created
// Multiple intervals running = multiple console logs
```

### Example: Timer WITH Cleanup ✅ (CORRECT)

javascript

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

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

    // Return cleanup function
    return () => {
      clearInterval(intervalId); // Stop the interval
      console.log("Interval cleaned up");
    };
  }, []); // Only set up once

  return <div>Seconds: {seconds}</div>;
}

// What happens:
// Component mounts → Interval 1 created
// Component unmounts → Cleanup runs → Interval cleared ✓
```

### How Cleanup Works - Step by Step

javascript

```
useEffect(() => {
  console.log("1. Effect runs");

  const timer = setInterval(() => {
    console.log("2. Timer ticking");
  }, 1000);

  return () => {
    console.log("3. Cleanup runs BEFORE:");
    console.log("   - Component unmounts");
    console.log("   - OR dependencies change");
    clearInterval(timer); // Stop the timer
  };
}, []);

// Output Timeline:
// Page loads → "1. Effect runs"
// 1 second passes → "2. Timer ticking"
// 2 seconds pass → "2. Timer ticking"
// User leaves page → "3. Cleanup runs" → Timer stops
```

## 5. Real-World Program Examples

### Example 1: API Call with Loading State

javascript

```javascript
function UserProfile({ userId }) {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    let isMounted = true; // Prevent state update if unmounted

    const fetchUser = async () => {
      try {
        setLoading(true);
        const response = await fetch(`/api/users/${userId}`);
        const data = await response.json();

        if (isMounted) {
          setUser(data);
          setError(null);
        }
      } catch (err) {
        if (isMounted) {
          setError(err.message);
        }
      } finally {
        if (isMounted) {
          setLoading(false);
        }
      }
    };

    fetchUser();

    // Cleanup function
    return () => {
      isMounted = false; // Prevent memory leaks
    };
  }, [userId]); // Re-fetch when userId changes

  if (loading) return <div>Loading...</div>;
  if (error) return <div>Error: {error}</div>;
  return <div>{user.name}</div>;
}
```

### Example 2: Event Listener with Cleanup

javascript

```
function WindowResize() {
  const [width, setWidth] = useState(window.innerWidth);

  useEffect(() => {
    const handleResize = () => {
      setWidth(window.innerWidth);
    };

    // Add listener
    window.addEventListener('resize', handleResize);

    // Cleanup: Remove listener
    return () => {
      window.removeEventListener('resize', handleResize);
      console.log("Event listener removed");
    };
  }, []); // Only set up once

  return <div>Window width: {width}px</div>;
}
```

**Example 3: Document Title Update**

```javascript
function PageTitle({ title }) {
  useEffect(() => {
    // Update document title
    document.title = title;

    // Cleanup: Restore original title
    return () => {
      document.title = "My App";
    };
  }, [title]);

  return <h1>{title}</h1>;
}
```

---

## 6. Interview Q&A Notes

### Q1: What is useEffect and why do we need it?

**Answer:** useEffect is a React Hook that runs side effects in functional components. It replaces the lifecycle methods (componentDidMount, componentDidUpdate, componentWillUnmount) from class components. Side effects include API calls, DOM manipulation, subscriptions, and timers.

**Q2: What are the differences between no dependency array, empty array, and array with values?**

**Answer:**

- **No dependency array**: Runs after EVERY render (performance issue usually)
- **Empty array []**: Runs ONLY ONCE after component mounts
- **Array with values [dep1, dep2]**: Runs when any dependency changes

**Q3: Why do we need to clear intervals?**

**Answer:** If we don't clear intervals, multiple intervals run simultaneously causing:

- Memory leaks (intervals continue in background)
- Multiple console logs/state updates
- Performance degradation
- Unexpected behavior

We return a cleanup function that calls `clearInterval()` to stop the timer.

**Q4: What is the cleanup function in useEffect?**

**Answer:** The cleanup function is an optional return value inside useEffect. It runs:

- Before the component unmounts
- Before the effect runs again (if dependencies changed)

It should clean up resources like timers, subscriptions, and event listeners.

**Q5: Can useEffect be used with async/await directly?**

**Answer:** No. useEffect callback itself cannot be async. Instead, create an async function inside:

```javascript
useEffect(() => {
  const fetchData = async () => {
    const data = await fetch('/api');
    setData(data);
  };

  fetchData();
}, []);
```

**Q6: How do you prevent memory leaks?**

**Answer:**

- Always clean up subscriptions and timers using the cleanup function
- Use a flag like `isMounted` to prevent state updates after unmount
- Cancel API requests using AbortController when component unmounts

**Q7: What's the difference between useEffect and useLayoutEffect?**

**Answer:**

- **useEffect**: Runs asynchronously AFTER the browser paints the screen
- **useLayoutEffect**: Runs synchronously BEFORE the browser paints (blocking)

Use useLayoutEffect for DOM measurements, useEffect for most other cases.

**Q8: How do you optimize useEffect performance?**

**Answer:**

- Use proper dependency arrays
- Avoid creating new objects/functions in dependency array
- Use useCallback/useMemo to memoize dependencies
- Don't forget cleanup functions

---

## 7. Common Mistakes & Solutions

### Mistake 1: Infinite Loop

```javascript
// ❌ WRONG - No dependency array
useEffect(() => {
  setCount(count + 1); // Runs after render → causes re-render → effect runs again
});

// ✅ CORRECT
useEffect(() => {
  // Do something once
}, []);
```

### Mistake 2: Stale Closures

```javascript
```

```
// ❌ WRONG
const [count, setCount] = useState(0);
useEffect(() => {
  const timer = setInterval(() => {
    console.log(count); // Always logs 0
  }, 1000);
}, []);

// ✅ CORRECT
useEffect(() => {
  const timer = setInterval(() => {
    setCount(prev => prev + 1); // Use functional update
  }, 1000);
}, []);
```

### Mistake 3: Missing Cleanup

javascript

```
// ❌ WRONG
useEffect(() => {
  window.addEventListener('click', handleClick);
}, []);

// ✅ CORRECT
useEffect(() => {
  window.addEventListener('click', handleClick);
  return () => window.removeEventListener('click', handleClick);
}, []);
```

---

## 8. Tamil Summary (தமிழ் சாரம்)

### useEffect என்பது என்ன?

useEffect என்பது functional components-ல் side effects செய்ய பயன்படும் React Hook ஆகும்.

### Lifecycle Methods vs useEffect

- **componentDidMount** → `useEffect(() => {...}, [])`
- **componentDidUpdate** → `useEffect(() => {...}, [deps])`
- **componentWillUnmount** → `return () => {...}` useEffect-க்குள்

### Interval Clear செய்யும் முறை

javascript

```
useEffect(() => {
  const intervalId = setInterval(() => {
    // Do something
  }, 1000);

  return () => {
    clearInterval(intervalId); // Interval நிறுத்தி விடு
  };
}, []);
```

**Interview நிறுத்தக் குறிப்புகள்**

1. useEffect எப்போது run ஆகுறது என்று தெரிந்து கொள்
2. Dependency array-ன் முக்கியத்துவம் புரி
3. Cleanup function ஏன் தேவை என்று விளக்க முடிய வேண்டும்
4. Memory leaks தவிர்ப்பது எப்படி என்று தெரிய வேண்டும்
5. Async/await useEffect-உடன் பயன்படுத்தும் முறை தெரிய வேண்டும்