# React `useState` Hook – Complete Notes

## 1. What is `useState`?

`useState` is a React hook that allows **functional components to have state**.

Functional components are just functions; variables reset on each render. `useState` **persists state across re-renders**.

**Syntax:**

```
const [state, setState] = useState(initialValue);
```

**Returns:** - `state` → current state value
- `setState` → function to update state and trigger re-render

**Example:**

```
const [count, setCount] = useState(0);
<button onClick={() => setCount(count + 1)}>Count: {count}</button>
```

## 2. Internal Working

- Each component has a **fiber node** in React.
- Each hook has a **queue of pending updates**.

**When** `setState` **is called:** 1. Update is queued in fiber
2. Component is marked as "dirty"
3. React waits until the end of the current event (batching)
4. Updates are processed sequentially → final state applied
5. Component re-renders once

**Key:** `useState` values persist, **initial value runs once** with lazy initializer.

## 3. Initial Value

- **Direct value** → runs every render (avoid for expensive calculations)

```
const [count, setCount] = useState(expensiveCalculation());
```

- **Lazy initializer** → runs **only on first render** (recommended for expensive computations)

```
const [count, setCount] = useState(() => expensiveCalculation());
```

**Use Cases:** dashboards, analytics, forms with large datasets.

## 4. Functional Updates (prev state)

Use when **new state depends on previous state**:

```
setCount(prev => prev + 1);
```

- Avoids **stale closure problem**
- Works perfectly with batching:

```
setCount(prev => prev + 1);
setCount(prev => prev + 1);
setCount(prev => prev + 1); // final count correct
```

**Use Cases:** - Counters, likes, votes
- Queues, chat messages
- Timers / intervals

## 5. Batching

- React **combines multiple updates** in one render for performance
- **React 18+** → batching works even in async events

**Normal updates:**

```
setCount(count + 1);
setCount(count + 1); // may only increment once due to stale closure
```

**Functional updates:**

```
setCount(prev => prev + 1);
setCount(prev => prev + 1); // increments correctly
```

**Mental Model:** - Normal = stale snapshot from render
- Functional = sequential computation from latest queued state

## 6. State Types

**Primitive:** number, string, boolean

```
const [isOpen, setIsOpen] = useState(false);
```

**Array:** always use **immutable updates**

```
setItems(prev => [...prev, newItem]);
```

**Object:** always use **immutable updates**

```
setForm(prev => ({ ...prev, email: "x" }));
```

> For deeply nested objects → consider **useReducer** or libraries like **Immer**

## 7. Immutable Updates — Key Principle

**Objects**

```
// ❌ Mutable (wrong)
obj.a = 2;
setObj(obj); // React may skip re-render

// ✅ Immutable (correct)
setObj(prev => ({ ...prev, a: 2 })); // new reference → triggers re-render
```

**Arrays**

```
// ❌ Mutable
items.push(newItem);
setItems(items); // may skip re-render

// ✅ Immutable
setItems(prev => [...prev, newItem]);
setItems(prev => prev.filter(i => i !== 2));
```

**Nested Objects**

```
setUser(prev => ({
  ...prev,
  profile: { ...prev.profile, email: "new@mail.com" }
}));
```

Reason: React uses **shallow comparison**. Mutable objects keep same reference → React may skip re-render.

## 8. Closures & Async Updates

```
setTimeout(() => {
  setCount(count + 1); // ❌ may be stale
  setCount(prev => prev + 1); // ✅ safe
}, 1000);
```

**Always use functional updates for async operations.**

## 9. Multiple State Variables vs Single Object

**Multiple** `useState` → independent fields → **less unnecessary re-renders**

```
const [name, setName] = useState("");
const [email, setEmail] = useState("");
```

**Single object** → grouped state

```
const [form, setForm] = useState({ name: "", email: "" });
setForm(prev => ({ ...prev, email: "x" }));
```

## 10. Async Update Gotchas

- `setState` is **not a promise** → state does not update immediately

  ```
  setCount(count + 1);
  console.log(count); // still old value ❌
  ```

✅Correct:

```
useEffect(() => {
  console.log(count); // runs after update ✅
}, [count]);
```

## 11. Derived State

**Don't store derived state**; compute on render.

```
// ❌ bad
const [fullName, setFullName] = useState(first + last);

// ✅ good
const fullName = first + last;
```

## 12. Performance Tips

- Lazy initialization for expensive calculations
- Functional updates for dependent state
- Immutable updates for objects/arrays
- Multiple small states vs single large object to avoid unnecessary renders
- Combine with `useCallback` / `React.memo` for child components

## 13. Production Use Cases

**Toggle modal**

```
const [isOpen, setIsOpen] = useState(false);
<button onClick={() => setIsOpen(prev => !prev)}>Toggle</button>
```

**Form state**

```
const [form, setForm] = useState({ username: "", email: "" });
const handleChange = e => {
  const { name, value } = e.target;
  setForm(prev => ({ ...prev, [name]: value }));
};
```

**Counter with multiple updates**

```
setCount(prev => prev + 1);
setCount(prev => prev + 1);
```

**Chat messages / feeds**

```
setMessages(prev => [...prev, newMessage]);
```

**Expensive dashboard initialization**

```
const [stats, setStats] = useState(() => computeStats(largeDataset));
```

## ✅Key Takeaways

- `useState` allows functional components to manage **persistent state**
- Use **functional updates** when state depends on previous state
- Always **update objects/arrays immutably**
- Async updates need care → prefer functional form
- Split state into multiple variables for **better performance**