How does the React State work?

Table of content

State as a Snapshot

```
const handleClick = () => {
   setCount(count + 1);
   console.log(count); // Still logs the previous value, not the updated one
};

return <button onClick={handleClick}>Increment</button>;
```

- Setting a state variable does not change the state variable you already have, but instead triggers a re-render.
- React stores state outside of your component, as if on a shelf.
- When you call `useState`, React gives you a snapshot of the state *for that render*.
- A state variable's value never changes within a render.
- Variables and event handlers don't "survive" re-renders. Every render has its own event handlers.
- Every render (and functions inside it) will always "see" the snapshot of the state that React gave to *that* render.
- Event handlers created in the past have the state values from the render in which they were created.

What happens when React re-renders a component:

1. React calls your function again.

2. Your function returns a new JSX snapshot.
3. React then updates the screen to match the snapshot your function returned.

React state updater function

In React, state updates are **asynchronous**. To ensure the new state is based on the previous state correctly, use the **updater function** inside `setState`.

Why Use the Updater Function?

1. Ensures correctness when updating based on the previous state.
2. Works well with batched updates, preventing stale state issues.

```
const handleClick = () => {
  setNumber(n => n + 1);
  setNumber(n => n + 1);
  setNumber(n => n + 1);
};
```

During the next render, React goes through the queue and gives you the final updated state.

```
setNumber(n => n + 1); // state is now updated to 1
setNumber(n => n + 1); // state is now updated to 2
setNumber(n => n + 1); // state is now updated to 3
```

3. Essential for state updates inside asynchronous functions (e.g., `setTimeout`, API calls).

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

const fetchData = async () => {
  await new Promise((resolve) => setTimeout(resolve, 5000));
  setCount(prev => prev + 1); // won't work if updater function is not used
};
```

```
return <button onClick={fetchData}>Click {count}</button>;
```

React state batch updates

React **batch updates** state changes to improve performance. Instead of re-rendering the component after every state update, React **groups multiple state updates** and applies them together in a single re-render. This reduces unnecessary renders and improves performance.

```
<button onClick={() => {
      setNumber(number + 1);
      setNumber(number + 1);
      setNumber(number + 1);
    }}>+3</button>
```

This will not update the number to 3 as you might expect. Because react will batch all the state updates for the next render.

Each render's state values are fixed, so the value of `number` inside the first render's event handler is always `0`, no matter how many times you call it.

```
setNumber(0 + 1); // state is now updated to 1
setNumber(0 + 1); // state is now updated to 1
setNumber(0 + 1); // state is now updated to 1
```

Benefits of Batching:

- This lets you update multiple state variables—even from multiple components—without triggering too many re-renders.
- But this also means that the UI won't be updated until *after* your event handler, and any code in it, completes.
- It makes the React app run much faster. It also avoids dealing with confusing "half-finished" renders where only some of the variables have been updated.