How React State Works

State variables might look like regular JavaScript variables that you can read and write to. However, state behaves more like a **snapshot**. Setting it does not change the state variable you already have, but instead triggers a **re-render**.

You will learn:

- How setting state triggers re-renders
- When and how state updates
- Why state does not update immediately after you set it
- How event handlers access a “snapshot” of the state

Setting State Triggers Renders

You might think of your user interface as changing directly in response to a user event like a click. In React, it works a little differently from this mental model. Setting state requests a re-render from React. This means that for an interface to react to the event, you need to update the state.

Here’s an example:

```
import { useState } from 'react';

export default function Form() {
  const [isSent, setIsSent] = useState(false);
  const [message, setMessage] = useState('Hi!');

  if (isSent) {
    return <h1>Your message is on its way!</h1>;
  }

  return (
    <form
      onSubmit={(e) => {
        e.preventDefault();
        setIsSent(true);
        sendMessage(message);
      }}
    >
      <textarea
        placeholder="Message"
        value={message}
        onChange={(e) => setMessage(e.target.value)}
      />
      <button type="submit">Send</button>
    </form>
  );
}
```

```
function sendMessage(message) {
  // ...
}
```

Here's what happens when you click the button:

1. The `onSubmit` event handler executes.
2. `setIsSent(true)` sets `isSent` to `true` and queues a new render.
3. React re-renders the component according to the new `isSent` value.

Rendering Takes a Snapshot in Time

"Rendering" means that React is calling your component, which is a function. The JSX you return from that function is like a snapshot of the UI in time. Its props, event handlers, and local variables were all calculated using its state at the time of the render.

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.

State Over Time

Here's an example of how state updates behave:

```
import { useState } from 'react';

export default function Counter() {
  const [number, setNumber] = useState(0);

  return (
    <>
      <h1>{number}</h1>
      <button
        onClick={() => {
          setNumber(number + 1);
          setNumber(number + 1);
          setNumber(number + 1);
        }}
      >
        +3
      </button>
    </>
  );
}
```

What happens here?

1. Clicking the button increments the counter **only once**.
2. Why? Because React batches state updates for the **next render**.

Even though you called `setNumber(number + 1)` three times, React processes all these updates together for the next render.

Example: State with Timers

What happens when you delay an alert?

```
import { useState } from 'react';

export default function Counter() {
  const [number, setNumber] = useState(0);

  return (
    <>
      <h1>{number}</h1>
      <button
        onClick={() => {
          setNumber(number + 5);
          setTimeout(() => {
            alert(number);
          }, 3000);
        }}
      >
        +5
      </button>
    </>
  );
}
```

Why does the alert show `0` instead of `5`?

- The `setTimeout` callback uses the **state snapshot** from the current render, where `number` was `0`. React doesn't pass the updated state to previously created event handlers.

Recap

1. Setting state **requests a new render**.
2. React stores state **outside** of your component.
3. State updates are **batched** for the next render.
4. Event handlers always use the **state snapshot** from their render.