React Lecture 11

State as Snapshot and Queuing series of state updates

State as Snapshot

Introduction

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.

Introduction

Example: What would be the output of the following code ?

```
1  import { useState } from 'react';
2
3  export default function Counter() {
4    const [number, setNumber] = useState(0);
5
6    return (
7      <>
8        <h1>{number}</h1>
9        <button onClick={() => {
10         setNumber(number + 1);
11         setNumber(number + 1);
12         setNumber(number + 1);
13       }}>+3</button>
14     </>
15   )
16 }
```

Setting state only changes it for the next render. During the first render, number was 0. This is why, in that render's onClick handler, the value of number is still 0 even after setNumber(number + 1) was called:

Here is what this button's click handler tells React to do:

1. setNumber(number + 1): number is 0 so setNumber(0 + 1).
    - React prepares to change number to 1 on the next render.
2. setNumber(number + 1): number is 0 so setNumber(0 + 1).
    - React prepares to change number to 1 on the next render.
3. setNumber(number + 1): number is 0 so setNumber(0 + 1).
    - React prepares to change number to 1 on the next render.

Queuing a series of state updates

Introduction

Setting a state variable will queue another render. But sometimes you might want to perform multiple operations on the value before queueing the next render. To do this, it helps to understand how React batches state updates.

React Batches State Updates

You might expect that clicking the “+3” button will increment the counter three times because it calls setNumber(number + 1) three times:

However, as you might recall from the previous section, 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 setNumber(1):

But there is one other factor at play here. React waits until all code in the event handlers has run before processing your state updates. This is why the re-render only happens after all these setNumber() calls.

Updating the same state multiple times before the next render

It is an uncommon use case, but if you would like to update the same state variable multiple times before the next render, instead of passing the next state value like setNumber(number + 1), you can pass a function that calculates the next state based on the previous one in the queue, like setNumber(n => n + 1)

It is a way to tell React to “do something with the state value” instead of just replacing it.

Updating the same state multiple times before the next render

Here, n => n + 1 is called an **updater function**. When you pass it to a state setter:

1. React queues this function to be processed after all the other code in the event handler has run.
2. During the next render, React goes through the queue and gives you the final updated state.

Updating the same state multiple times before the next render

So If we replace the previous code with state updater function

Here's how React works through these lines of code while executing the event handler:

```
setNumber(n => n + 1);
setNumber(n => n + 1);
setNumber(n => n + 1);
```

1. setNumber(n => n + 1): n => n + 1 is a function. React adds it to a queue.
2. setNumber(n => n + 1): n => n + 1 is a function. React adds it to a queue.
3. setNumber(n => n + 1): n => n + 1 is a function. React adds it to a queue.

Updating the same state multiple times before the next render

When you call useState during the next render, React goes through the queue. The previous number state was 0, so that's what React passes to the first updater function as the n argument. Then React takes the return value of your previous updater function and passes it to the next updater as n, and so on:

queued update	n	returns
`n => n + 1`	`0`	`0 + 1 = 1`
`n => n + 1`	`1`	`1 + 1 = 2`
`n => n + 1`	`2`	`2 + 1 = 3`

React stores 3 as the final result and returns it from useState.

What happens if you update state after replacing it ?

What about this event handler? What do you think number will be in the next render?

```
1  import { useState } from 'react';
2
3  export default function Counter() {
4    const [number, setNumber] = useState(0);
5
6    return (
7      <>
8        <h1>{number}</h1>
9        <button onClick={() => {
10         setNumber(number + 5);
11         setNumber(n => n + 1);
12       }}>Increase the number</button>
13     </>
14   )
15 }
16
```

What happens if you update state after replacing it ?

Here's what this event handler tells React to do:

1. setNumber(number + 5): number is 0, so setNumber(0 + 5). React adds "replace with 5" to its queue.
2. setNumber(n => n + 1): n => n + 1 is an updater function. React adds that function to its queue.

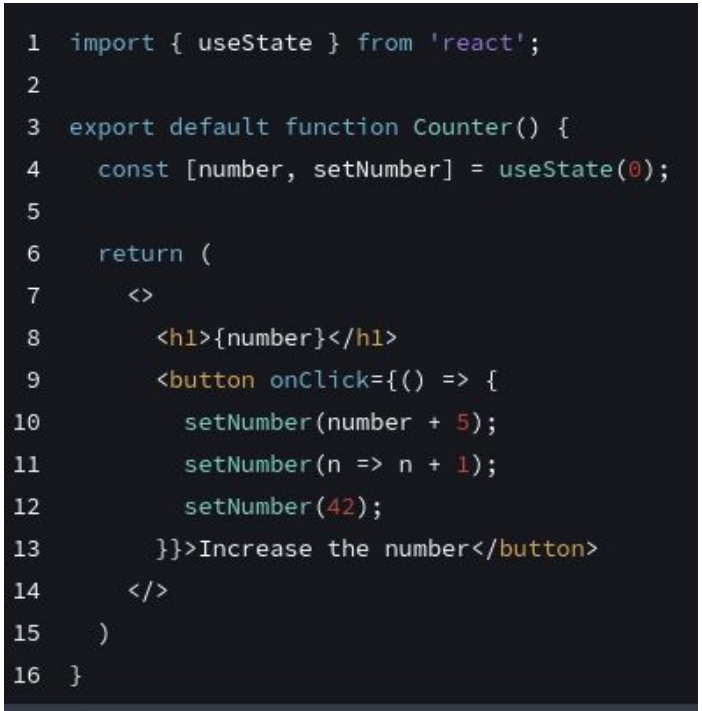
During the next render, React goes through the state queue:

queued update	n	returns
"replace with `5`"	`0` (unused)	`5`
`n => n + 1`	`5`	`5 + 1 = 6`

React stores 6 as the final result and returns it from useState.

What happens if you replace state after updating it

Let’s try one more example. What do you think number will be in the next render?



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

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

  return (
    <>
      <h1>{number}</h1>
      <button onClick={() => {
        setNumber(number + 5);
        setNumber(n => n + 1);
        setNumber(42);
      }}>Increase the number</button>
    </>
  )
}
```

What happens if you replace state after updating it

Here's how React works through these lines of code while executing this event handler:

1. setNumber(number + 5): number is 0, so setNumber(0 + 5). React adds "replace with 5" to its queue.
2. setNumber(n => n + 1): n => n + 1 is an updater function. React adds that function to its queue.
3. setNumber(42): React adds "replace with 42" to its queue.

During the next render, React goes through the state queue:

Then React stores 42 as the final result and returns it from useState.

queued update	n	returns
"replace with `5`"	`0` (unused)	`5`
`n => n + 1`	`5`	`5 + 1 = 6`
"replace with `42`"	`6` (unused)	`42`

Naming Convention

It’s common to name the updater function argument by the first letters of the corresponding state variable:

```
setEnabled(e => !e);
setLastName(ln => ln.reverse());
setFriendCount(fc => fc * 2);
```

If you prefer more verbose code, another common convention is to repeat the full state variable name, like setEnabled(enabled => !enabled), or to use a prefix like setEnabled(prevEnabled => !prevEnabled).