Queueing a Series of State Updates

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.

You will learn

- What “batching” is and how React uses it to process multiple state updates
- How to apply several updates to the same state variable in a row

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:

App.js Download Reset Fork

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

```
        }}>+3</button>
      </>
    )
  }
```

Show more

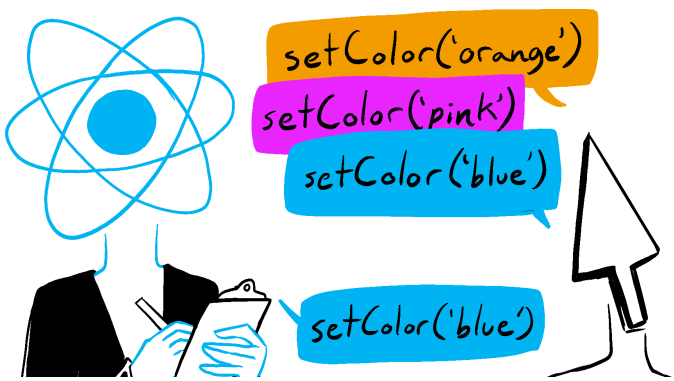
0 +3

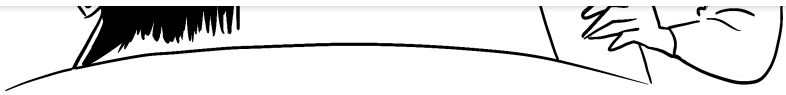
However, as you might recall from the previous section, each render's state values are fixed, so the value of `number` inside the first render's event handler is always `0`, no matter how many times you call `setNumber(1)`:

```
setNumber(0 + 1);
setNumber(0 + 1);
setNumber(0 + 1);
```

But there is one other factor at work here to discuss. **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.

This might remind you of a waiter taking an order at the restaurant. A waiter doesn't run to the kitchen at the mention of your first dish! Instead, they let you finish your order, let you make changes to it, and even take orders from other people at the table.





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. This behavior, also known as **batching,** makes your React app run much faster. It also avoids dealing with confusing "half-finished" renders where only some of the variables have been updated.

React does not batch across *multiple* intentional events like clicks—each click is handled separately. Rest assured that React only does batching when it's generally safe to do. This ensures that, for example, if the first button click disables a form, the second click would not submit it again.

Updating the same state variable 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.

Try incrementing the counter now:

App.js Download Reset Fork

```
1  import { useState } from 'react';
2
3
4  export default function Counter() {
5    const [number, setNumber] = useState(0);
6
7
8
9
```

```
15
16        <button onClick={() => {
17          setNumber(n => n + 1);
            setNumber(n => n + 1);
            setNumber(n => n + 1);
          }}>+3</button>
        </>
      )
    }
```

Show more

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.

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

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

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.

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:

`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`.

This is why clicking "+3" in the above example correctly increments the value by 3.

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?

```
<button onClick={() => {
  setNumber(number + 5);
  setNumber(n => n + 1);
}}>
```

App.js

Download Reset Fork

```
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);
      }}>Increase the number</button>
    </>
  )
```

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`.

> You may have noticed that `setState(x)` actually works like `setState(n => x)`, but `n` is unused!

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?

```
<button onClick={() => {
  setNumber(number + 5);
  setNumber(n => n + 1);
  setNumber(42);
}}>
```

App.js Download Reset Fork

```
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>
    </>
  )
}
```

Show more

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:

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

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

- **An updater function** (e.g. `n => n + 1`) gets added to the queue.
- **Any other value** (e.g. number `5`) adds "replace with `5`" to the queue, ignoring what's already queued.

After the event handler completes, React will trigger a re-render. During the re-render, React will process the queue. Updater functions run during rendering, so **updater functions must be pure** and only *return* the result. Don't try to set state from inside of them or run other side effects. In Strict Mode, React will run each updater function twice (but discard the second result) to help you find mistakes.

Naming conventions

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)`.

Recap

- Setting state does not change the variable in the existing render, but it requests a new render.
- React processes state updates after event handlers have finished running. This is called batching.
- To update some state multiple times in one event, you can use `setNumber(n => n + 1)` updater function.

1. Fix a request counter 2. Implement the state queue yourself

Challenge 1 of 2:
Fix a request counter

You're working on an art marketplace app that lets the user submit multiple orders for an art item at the same time. Each time the user presses the "Buy" button, the "Pending" counter should increase by one. After three seconds, the "Pending" counter should decrease, and the "Completed" counter should increase.

However, the "Pending" counter does not behave as intended. When you press "Buy," it decreases to `-1` (which should not be possible!). And if you click fast twice, both counters seem to behave unpredictably.

Why does this happen? Fix both counters.

App.js Download Reset Fork

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


export default function RequestTracker() {
  const [pending, setPending] = useState(0);
  const [completed, setCompleted] = useState(0);

  async function handleClick() {
    setPending(pending + 1);
    await delay(3000);
    setPending(pending - 1);
    setCompleted(completed + 1);
  }

  return (
    <>
      <h3>
```

Show more

Show solution

Next Challenge

PREVIOUS
State as a Snapshot

NEXT
Updating Objects in State

FACEBOOK

Open Source


React Docs