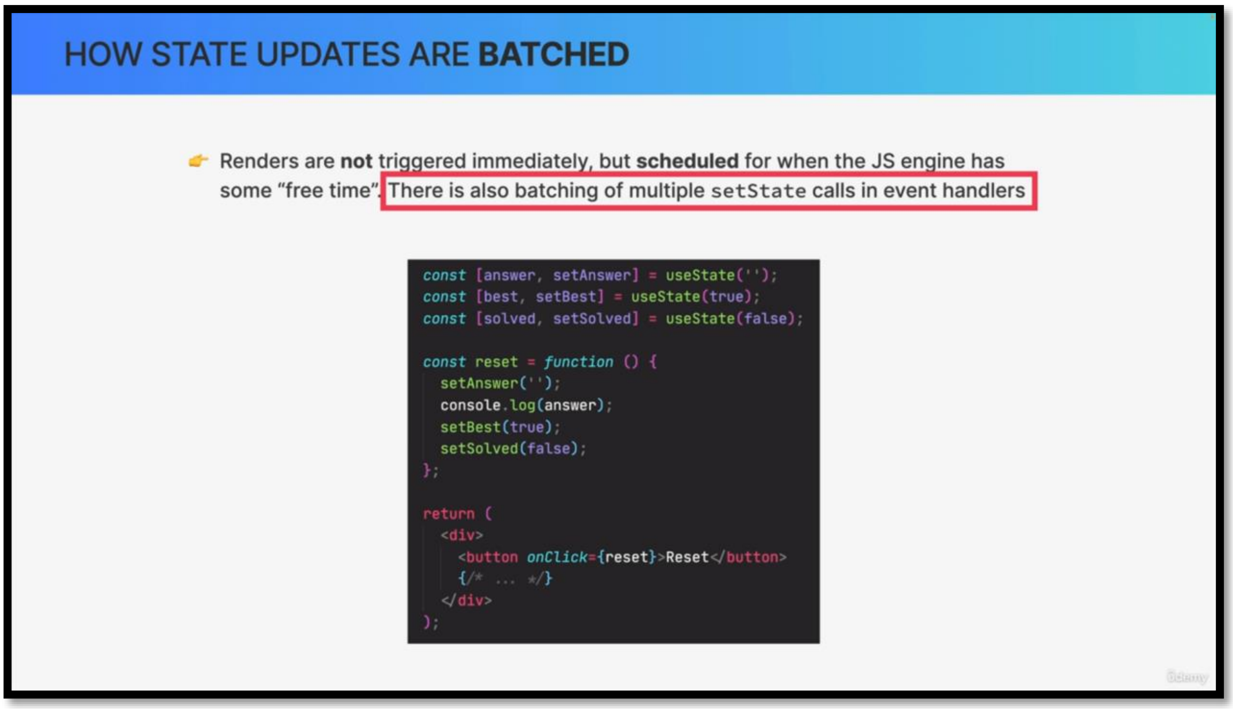
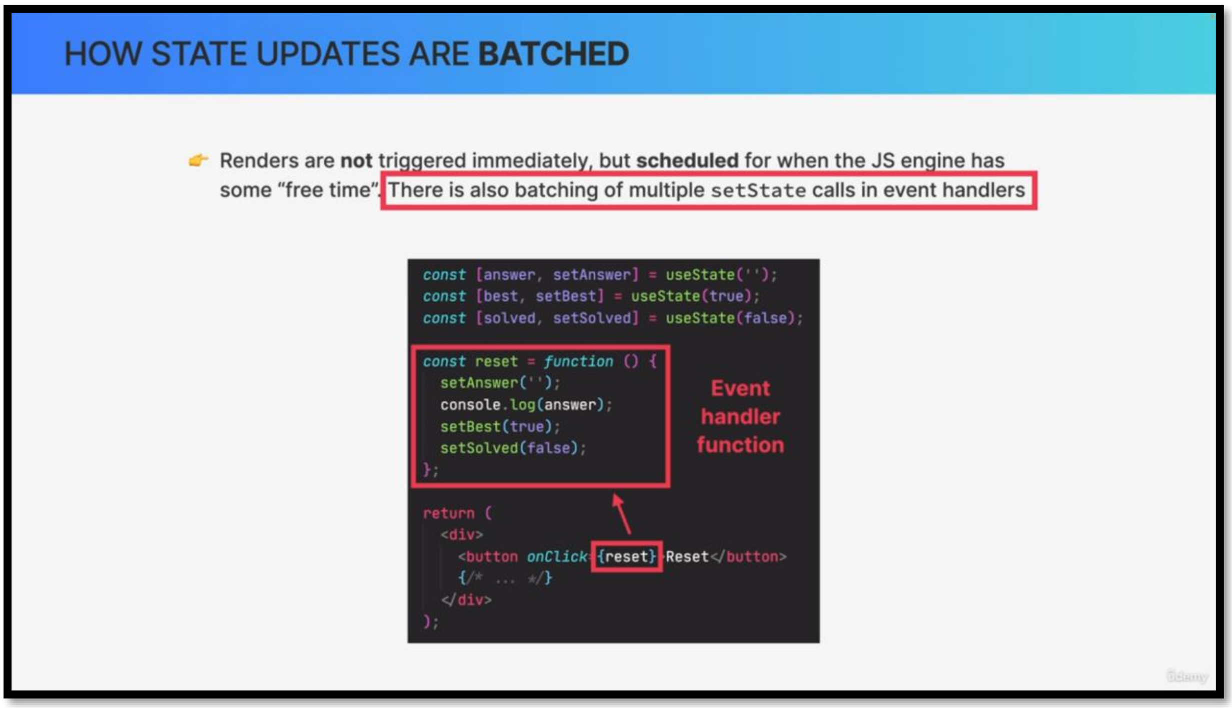


HOW STATE UPDATES ARE **BATCHED**

- Renders are **not** triggered immediately, but **scheduled** for when the JS engine has some "free time". There is also batching of multiple `setState` calls in event handlers

```
const [answer, setAnswer] = useState('');
const [best, setBest] = useState(true);
const [solved, setSolved] = useState(false);

const reset = function () {
  setAnswer('');
  console.log(answer);
  setBest(true);
  setSolved(false);
};

return (
  <div>
    <button onClick={reset}>Reset</button>
    {/* ... */}
  </div>
);
```

So here we have three pieces of state, defined using the useState hook and we also have a button in the user interface. Whenever there is a click on the button, the event handler function named reset is called.



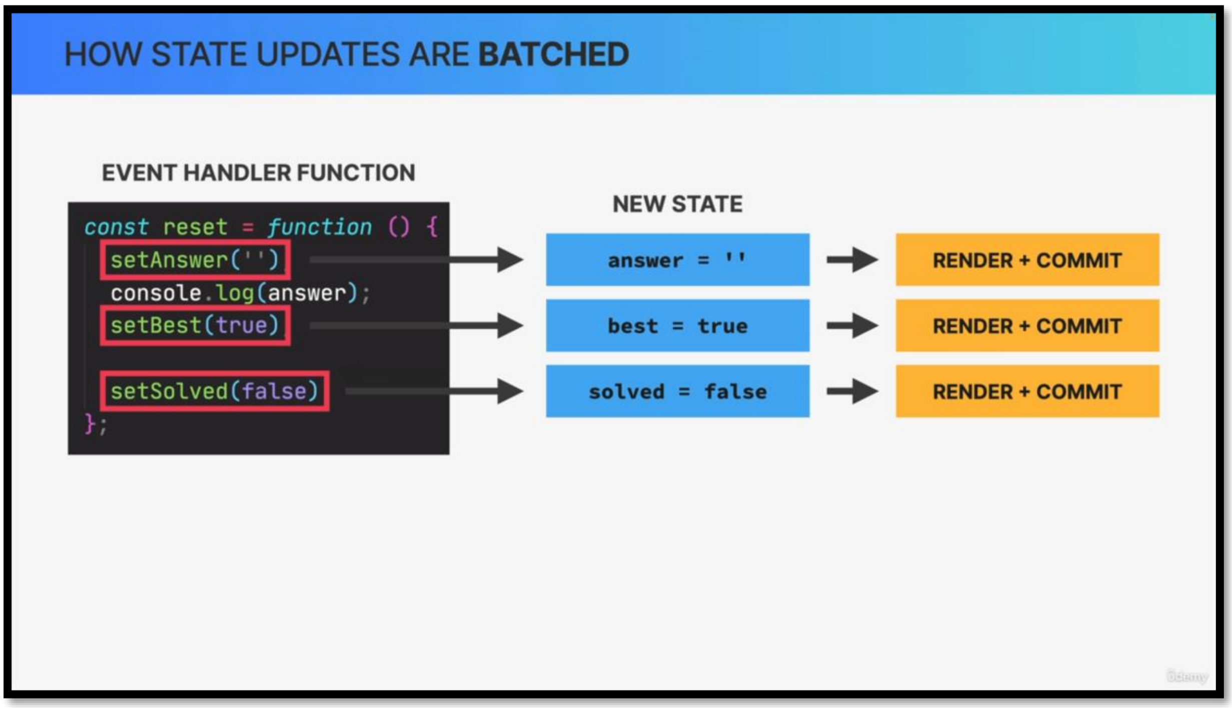
HOW STATE UPDATES ARE **BATCHED**

- Renders are **not** triggered immediately, but **scheduled** for when the JS engine has some "free time". There is also batching of multiple `setState` calls in event handlers

```
const [answer, setAnswer] = useState('');
const [best, setBest] = useState(true);
const [solved, setSolved] = useState(false);

const reset = function () {
  setAnswer('');
  console.log(answer);
  setBest(true);
  setSolved(false);
};

return (
  <div>
    <button onClick={reset}>Reset</button>
    {/* ... */}
  </div>
);
```

In this function, the three pieces of state, answer, best, and solved, are basically reverted back to their original state and therefore, this function is called reset.

These three pieces of state are actually updated behind the scenes.



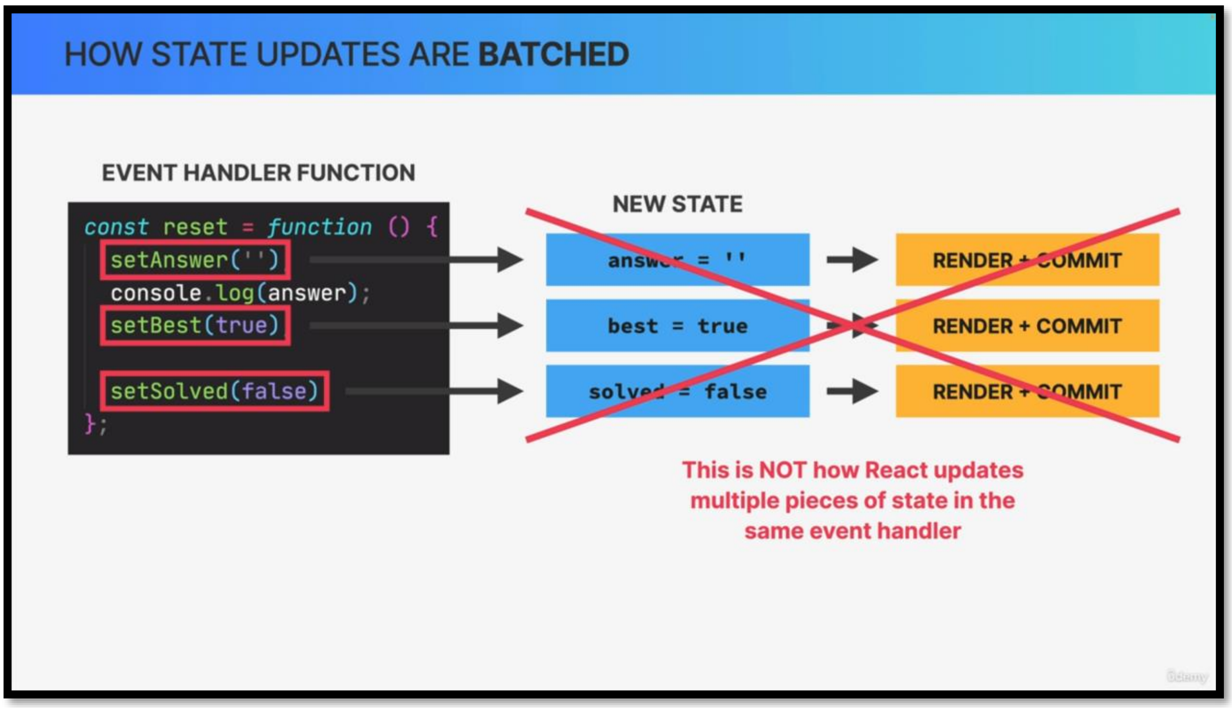
HOW STATE UPDATES ARE **BATCHED**

EVENT HANDLER FUNCTION

```
const reset = function () {
  setAnswer('')
  console.log(answer);
  setBest(true)

  setSolved(false)
};
```

We might think that, as React sees the set answer function call it would update the state to the empty string as requested and then trigger a re-render, and the commit phase then it would move on to the next line and do the same thing again, and finally do the entire thing one more time for the third state update.

Intuitively, we would think that, if there are three state variables being updated in this event handler, then React would re-render three times.

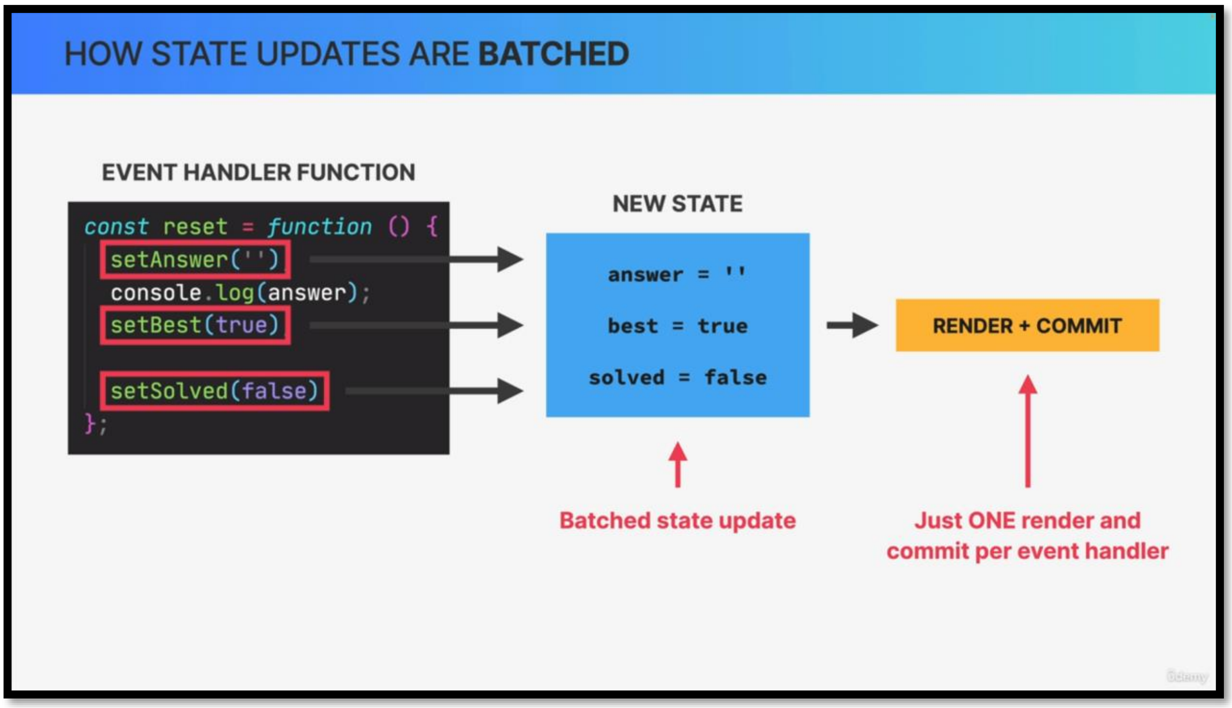


HOW STATE UPDATES ARE **BATCHED**

EVENT HANDLER FUNCTION

```
const reset = function () {
  setAnswer('')
  console.log(answer);
  setBest(true)

  setSolved(false)
};
```

However, this is actually not how it works. So, this is not how React updates multiple pieces of state in the same event handler function.



HOW STATE UPDATES ARE **BATCHED**

EVENT HANDLER FUNCTION

```
const reset = function () {
  setAnswer('')
  console.log(answer);
  setBest(true)

  setSolved(false)
};
```

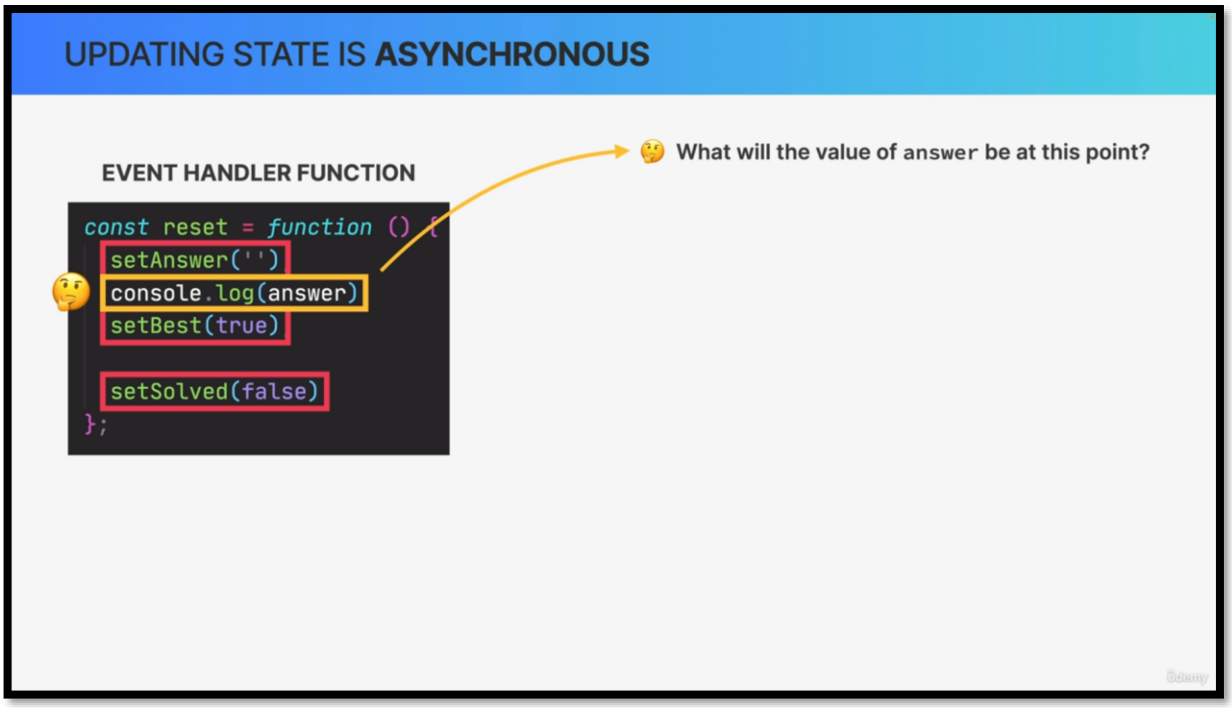
Instead, these state updates will actually get batched into just one state update for the entire event handler. So, updating multiple pieces of state won't immediately cause a re-render for each update. Instead, all pieces of state inside the event handler are updated in one go.

So, they are batched, and only then will React trigger one single render and commit.

Conceptually, it makes sense that React works this way because if we're updating these pieces of state together it probably means that they should just represent one new view, and therefore, React only updates the screen once.

If these state updates belong together, it really wouldn't make much sense to update the screen three times. Doing so would also create two wasted renders, because we're not interested in the first two renders, only the final one, which already contains all the three state updates.

Therefore, the fact that React automatically batches state updates in this way is yet another performance optimization that React gives us out of the box.



UPDATING STATE IS **ASYNCHRONOUS**

EVENT HANDLER FUNCTION

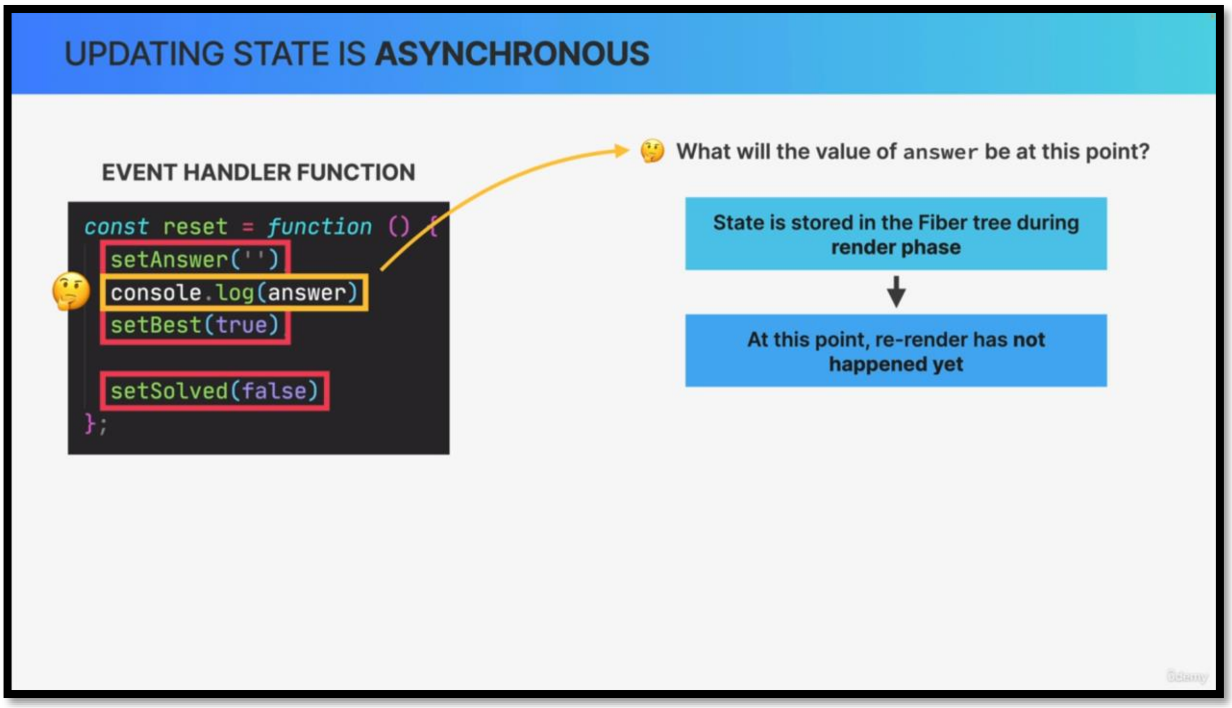
```
const reset = function () {
  setAnswer('')
  console.log(answer)
  setBest(true)

  setSolved(false)
};
```

🤔 What will the value of `answer` be at this point?

Batching state updates is extremely useful but it can also have surprising results.

So, what do you think will be the value of this variable at this point?

Component state is stored in the fiber tree during the render phase.



UPDATING STATE IS **ASYNCHRONOUS**

EVENT HANDLER FUNCTION

```
const reset = function () {
  setAnswer('')
  console.log(answer)
  setBest(true)

  setSolved(false)
};
```

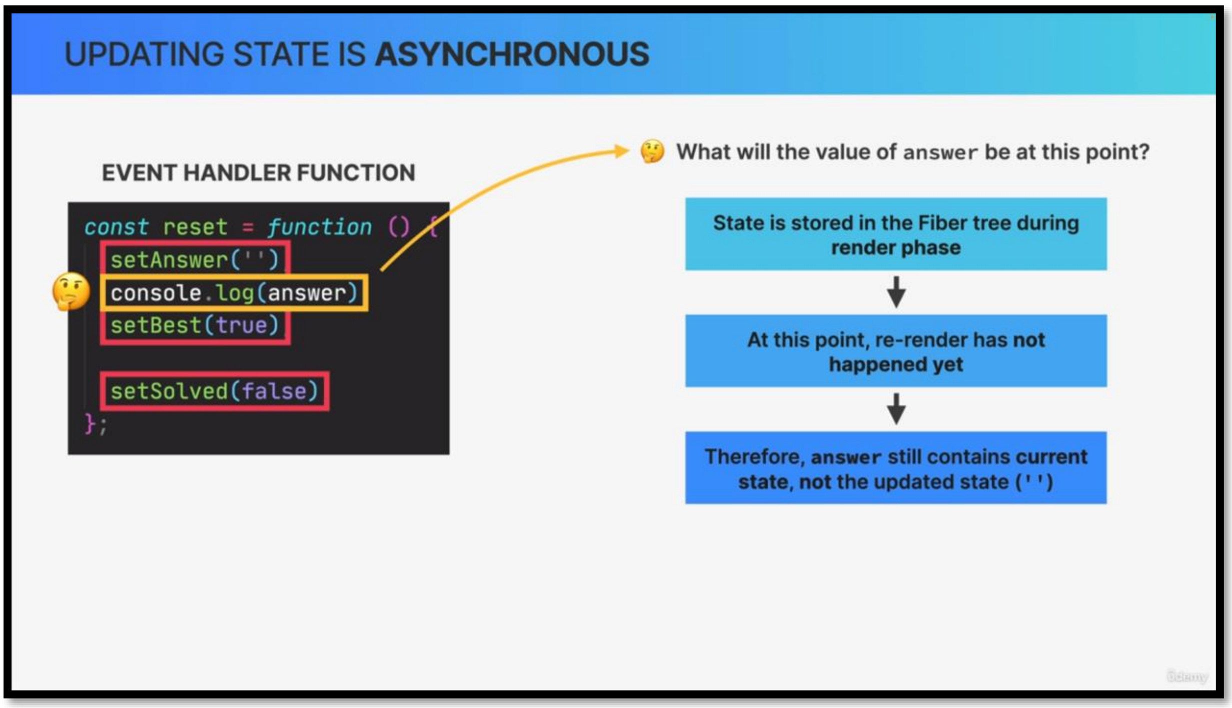
🤔 What will the value of `answer` be at this point?

State is stored in the Fiber tree during render phase

↓

At this point, re-render has not happened yet

Now, at this point in the code, the render phase has not happened yet. So, React is still reading the function line by line to figure out what state needs to be updated, but it hasn't actually updated the state yet, and it also hasn't re-rendered yet.



UPDATING STATE IS **ASYNCHRONOUS**

EVENT HANDLER FUNCTION

```
const reset = function () {
  setAnswer('')
  console.log(answer)
  setBest(true)

  setSolved(false)
};
```

What will the value of `answer` be at this point?

State is stored in the Fiber tree during render phase

↓

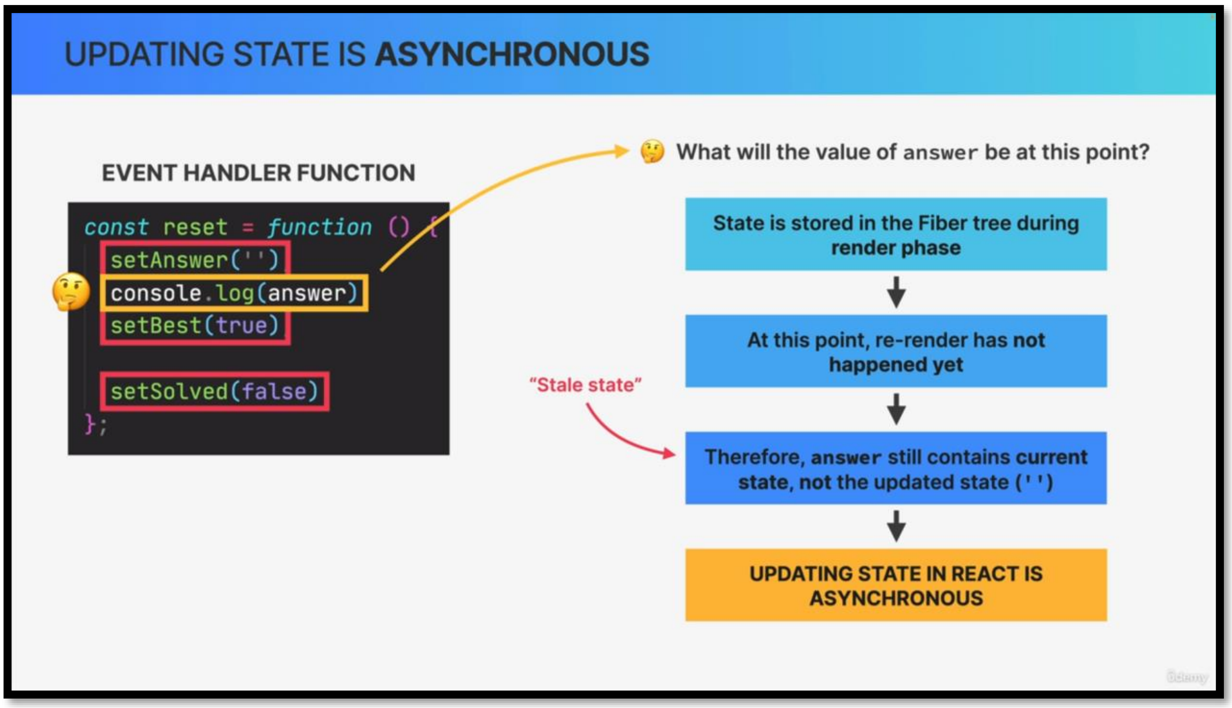
At this point, re-render has not happened yet

↓

Therefore, `answer` still contains current state, not the updated state (`''`)

That's the whole point of batching state updates in the first place.

So, what this means is that, at this point of the code, the answer variable will still hold the current state. So, the state before the update, even though we already told React to update it. So, at this point we say that our state is stale, meaning that the state is no longer fresh and updated, because in fact, a state update will only be reflected in the state variable after the re-render.



UPDATING STATE IS **ASYNCHRONOUS**

EVENT HANDLER FUNCTION

```
const reset = function () {
  setAnswer('')
  console.log(answer)
  setBest(true)

  setSolved(false)
};
```

What will the value of answer be at this point?

State is stored in the Fiber tree during render phase

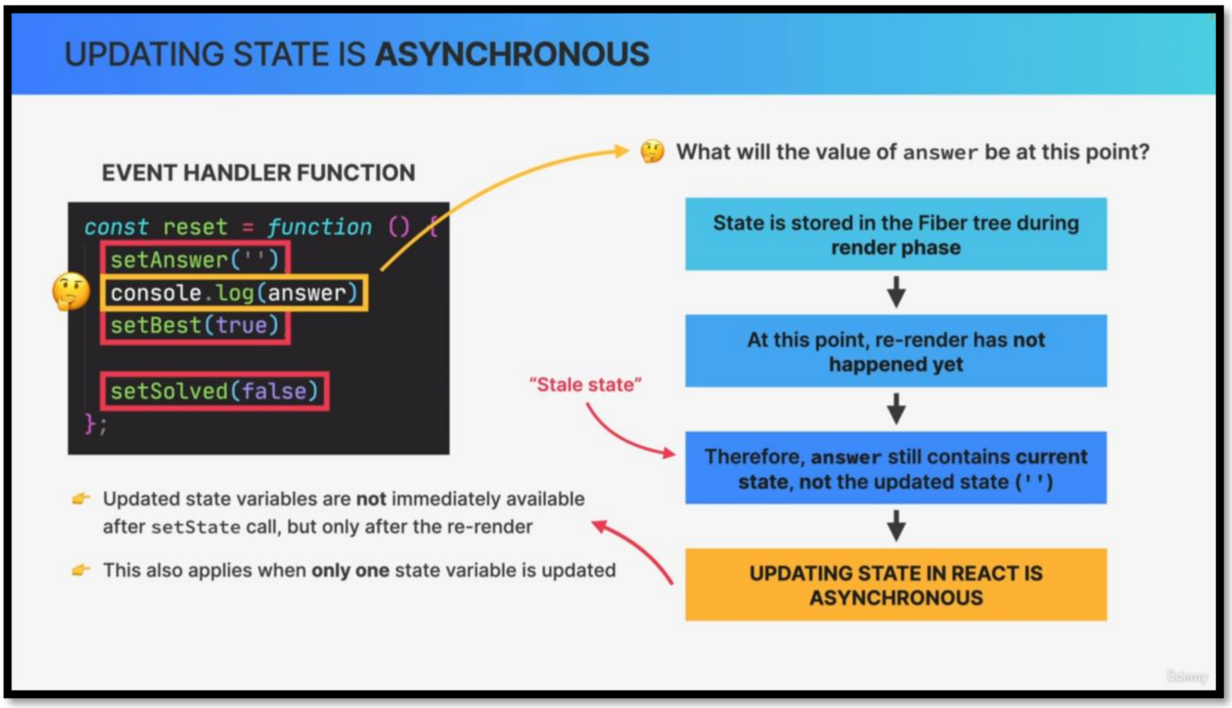
At this point, re-render has not happened yet

"Stale state"

Therefore, answer still contains current state, not the updated state ('')

UPDATING STATE IN REACT IS ASYNCHRONOUS

We say that updating state in React is asynchronous, and again, it is asynchronous because React does not give us the updated state variable immediately after the set answer call, but only after the re-render has happened.



UPDATING STATE IS **ASYNCHRONOUS**

EVENT HANDLER FUNCTION

```
const reset = function () {
  setAnswer('')
  console.log(answer)
  setBest(true)

  setSolved(false)
};
```

What will the value of answer be at this point?

State is stored in the Fiber tree during render phase

At this point, re-render has not happened yet

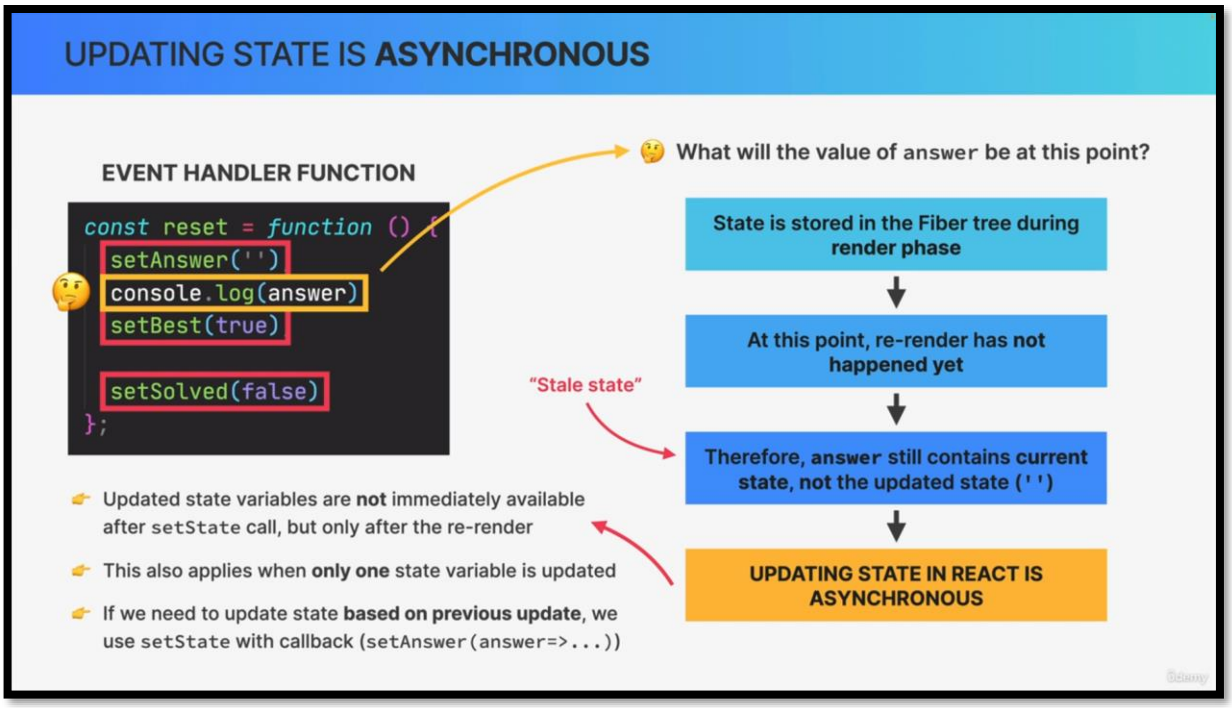
"Stale state"

Therefore, answer still contains current state, not the updated state ('')

UPDATING STATE IN REACT IS ASYNCHRONOUS

- Updated state variables are **not** immediately available after `setState` call, but only after the re-render
- This also applies when **only one** state variable is updated

The same thing is also true whenever there is only one piece of state being updated. So, no matter how many state variables are being updated, the updated state is only available after the re-render, not immediately.



UPDATING STATE IS **ASYNCHRONOUS**

EVENT HANDLER FUNCTION

```
const reset = function () {
  setAnswer('')
  console.log(answer)
  setBest(true)

  setSolved(false)
};
```

🤔 What will the value of `answer` be at this point?

State is stored in the Fiber tree during render phase

↓

At this point, re-render has not happened yet

↓

"Stale state" → Therefore, `answer` still contains current state, not the updated state ('')

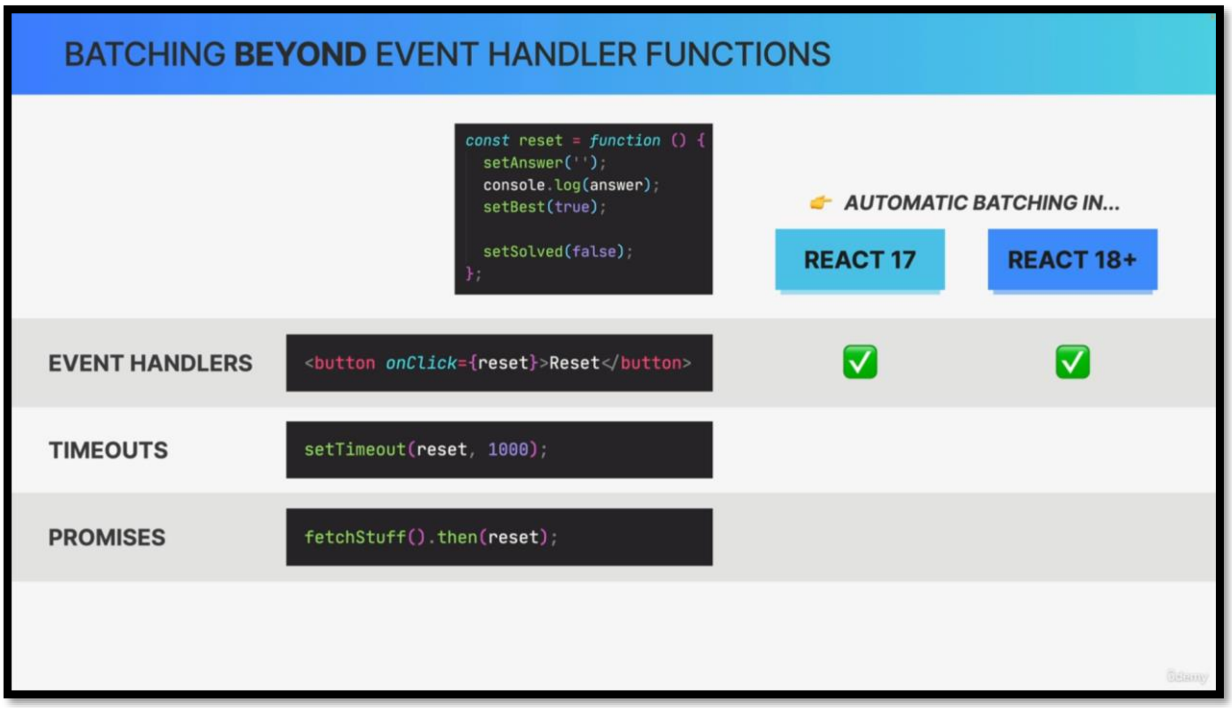
↓

UPDATING STATE IN REACT IS ASYNCHRONOUS

- Updated state variables are **not** immediately available after `setState` call, but only after the re-render
- This also applies when **only one** state variable is updated
- If we need to update state **based on previous update**, we use `setState` with callback (`setAnswer(answer=>...)`)

Now, sometimes we actually do need the new value immediately after updating it, and in the case, that we need the new value in order to update the same state again.

In other words, if we need to update state based on a previous state update in the same event handler, we can pass a callback function into the set state function instead of a single value.



BATCHING **BEYOND** EVENT HANDLER FUNCTIONS

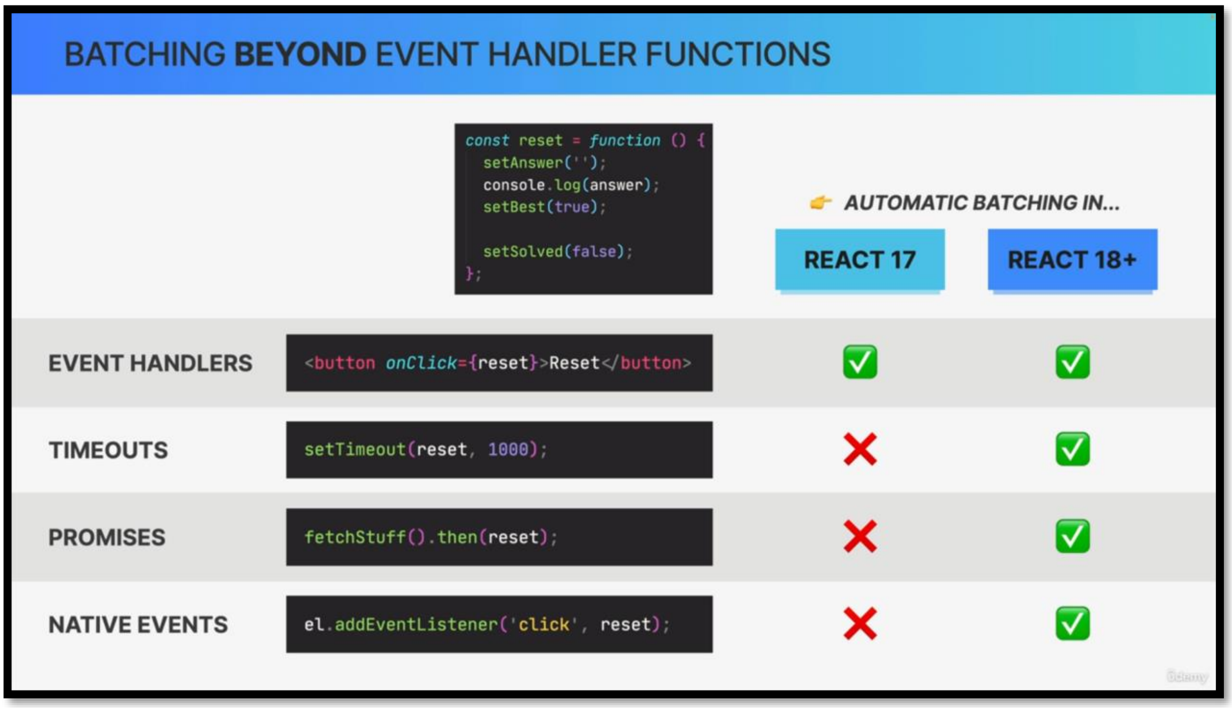
```
const reset = function () {
  setAnswer('');
  console.log(answer);
  setBest(true);

  setSolved(false);
};
```

AUTOMATIC BATCHING IN...

		REACT 17	REACT 18+
EVENT HANDLERS	`<button onClick={reset}>Reset</button>`	✅	✅
TIMEOUTS	`setTimeout(reset, 1000);`		
PROMISES	`fetchStuff().then(reset);`		

Now, so far, we have only talked about batching in event handler functions, like our reset function. That's because before React 18, React only did automatic batching in event handlers but not in situations that happen after a browser event has already happened.

However, there are certain very important situations in which we do need to update state, long after a browser event, like a click, has happened. Examples of that are timeouts and promises, for instance, we might want to run our reset function only a second after a click event or we might want to run it after some data has been fetched.



BATCHING **BEYOND** EVENT HANDLER FUNCTIONS

```
const reset = function () {
  setAnswer('');
  console.log(answer);
  setBest(true);

  setSolved(false);
};
```

👉 AUTOMATIC BATCHING IN...

		REACT 17	REACT 18+
EVENT HANDLERS	`<button onClick={reset}>Reset</button>`	✓	✓
TIMEOUTS	`setTimeout(reset, 1000);`	✗	✓
PROMISES	`fetchStuff().then(reset);`	✗	✓
NATIVE EVENTS	`el.addEventListener('click', reset);`	✗	✓

So, it would be nice to also have automatic batching in those situations to improve performance.

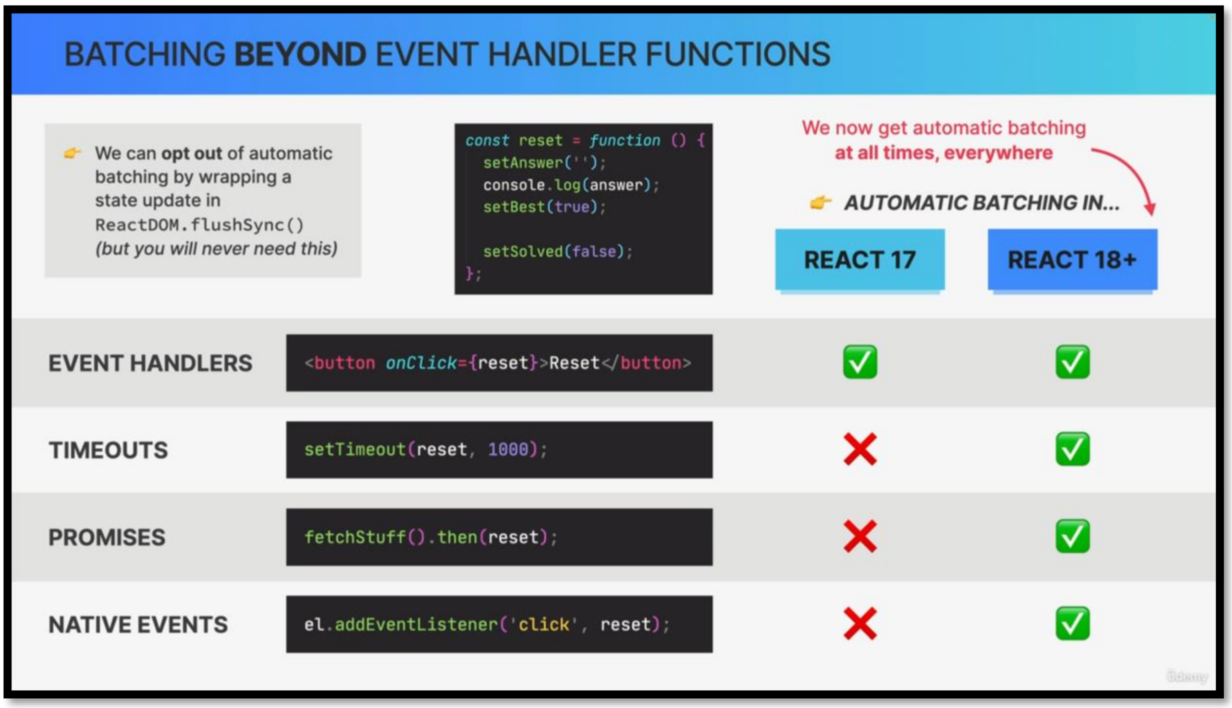
That's actually one of the important features that React 18 gave us.

Before React 18, if this reset function was called by a timeout, or by a promise, state updates inside the function would not be batched. Instead, in these situations, React would actually update the state variables one by one, and therefore, in this case, render three times.

Now another case is handling native events using DOM methods such as addEventListener, where state updates also used to not be batched, but now they are.

If you're using the latest React version, you will now get automatic batching all the time, everywhere in your code.

If, for some reason, you are working with an older version of React, maybe at your work, it's important that you know that batching used to work in a different way before version 18.



BATCHING **BEYOND** EVENT HANDLER FUNCTIONS

We can **opt out** of automatic batching by wrapping a state update in `ReactDOM.flushSync()` *(but you will never need this)*

```
const reset = function () {
  setAnswer('');
  console.log(answer);
  setBest(true);

  setSolved(false);
};
```

We now get automatic batching **at all times, everywhere**

AUTOMATIC BATCHING IN...

		REACT 17	REACT 18+
EVENT HANDLERS	`<button onClick={reset}>Reset</button>`	✓	✓
TIMEOUTS	`setTimeout(reset, 1000);`	✗	✓
PROMISES	`fetchStuff().then(reset);`	✗	✓
NATIVE EVENTS	`el.addEventListener('click', reset);`	✗	✓

Now, there are also some extremely rare situations in which automatic batching can be problematic. So, if you ever find yourself in a situation like that, you can just wrap the problematic state update in a ReactDOM.flushSync function and React will then exclude that update from batching.