React State

let's understand it better

Agenda:

- Why keep state in React
- State hooks
  - useState
  - useReducer
- What asynchronous can mean
- When does React actually update
- Concurrent mode
  - External state & tearing

Why keep state in React?

- It's the only way of telling React to re-render
  - Class components used to have a `forceUpdate` method
  - It can be simulated in stateless components… by using state
- It allows for concurrent mode patterns
- It helps performance
  - React only re-renders the subtree affected
  - React automatically batches updates
  - State updates are asynchronous

useState

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

const Component = () => {
  const [state, setState] = useState(initial);
  return <div>{state}</div>
}
```

Initialization

- `initial` can be a value or an initialization function

```
useState(0)
useState(() => 0)
```

- important to remember when storing a function

```
useState(func) // this will store whatever func returns
useState(() => func) // this will store func
```

useState

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

const Component = () => {
  const [state, setState] = useState(initial);
  return <div>{state}</div>
}
```

Updating

- update can be either a value or a function

```
  setState(0)
  setState((current) => 0)
```

- if the next state is derived from the previous one, prefer the latter form
- if you want to store a function you must use the latter form
- state is only updated if the new one is different (Using `Object.is` to compare)

b.

useState is only syntactic sugar for useReducer

useReducer

```
import { useReducer } from 'react';
import reducer from './reducer';

const Component = () => {
  const [state, dispatch] = useReducer(reducer, initialValue, initializer);
  return <div>{state}</div>
}
```

Initialization

- `initialValue` is used as-is, even if it's a function, unlike `useState`
- if you need initialization logic, use the `initializer`, which is a 3rd, optional argument
  - it is called with the `initialValue`
  - the returned value will be used as the initial state instead
- `reducer` is a function with the following signature

```
type Reducer<State, Action> = (prevState: State, action: Action) => State
```

useReducer

```
import { useReducer } from 'react';
import reducer from './reducer';

const Component = () => {
  const [state, dispatch] = useReducer(reducer, initialValue, initializer);
  return <div>{state}</div>
}
```

Updating

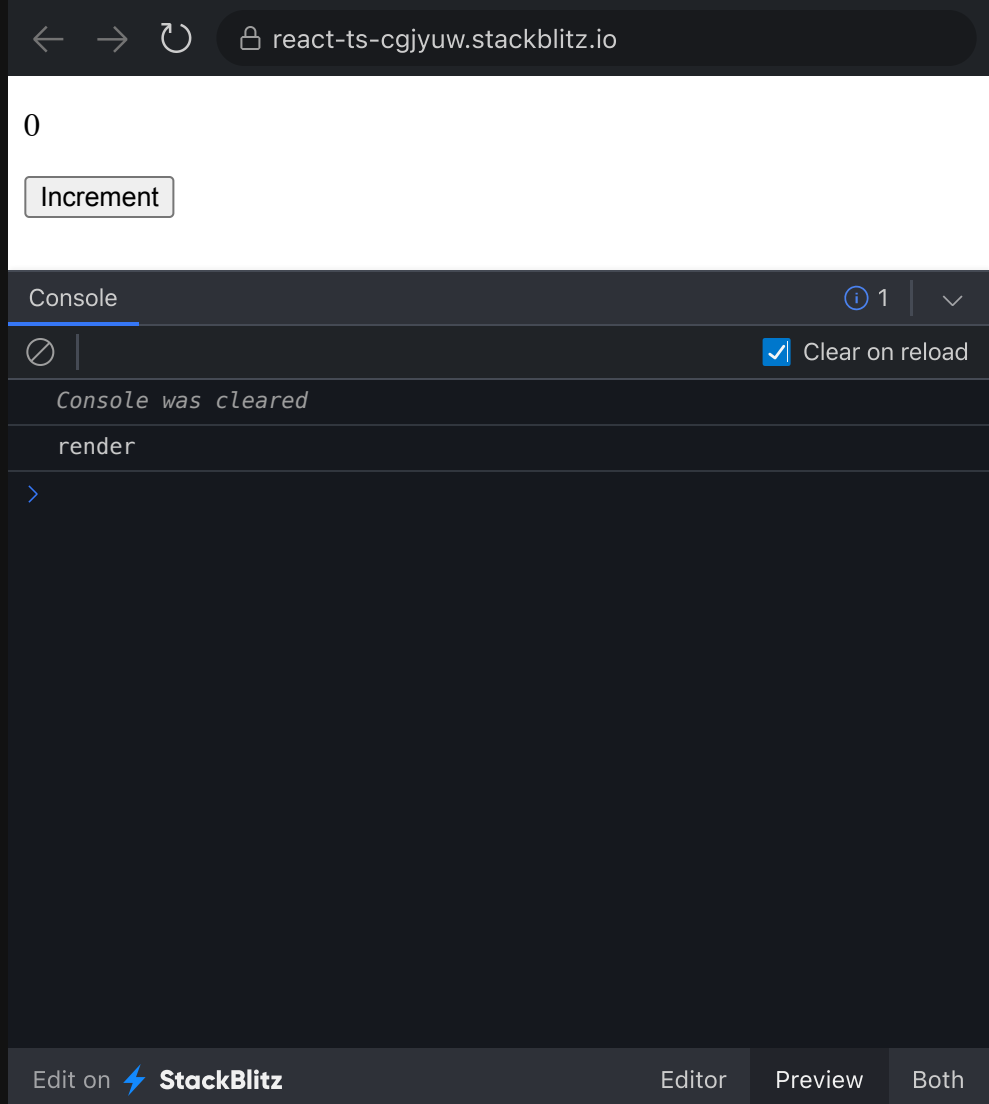
```
const [state, dispatch] = useReducer(reducer, initialValue, initializer)
//...
dispatch(action)
```

Reducer

```tsx
import { useReducer } from 'react';

type SetStateAction<State> = State | ((prev: State) => State);

const reducer = <State>(prevState: State, action: SetStateAction<State>): State => {
  if(typeof action === 'function') {
    return action(prevState);
  } else {
    return action;
  }
}

export const useState = <State>(initialState?: State | (() => State)) =>
  useReducer(
    reducer,
    undefined,
    () => typeof initialState === 'function' ? initialState() : initialState
  )
```

State updates are asynchronous

```
const Counter = () => {
  const [state, setState] = useState(0);
  const increment = () => {
    console.log('before increment');
    setState((current) => {
      console.log('increment');
      return current + 1;
    });
    console.log('after increment');
  };
  console.log('render');
  return (
    <div>
      <p>{state}</p>
      <button onClick={increment}>
        Increment
      </button>
    </div>
  );
}
```



What asynchronous can mean

The browser environment gives us multiple choices when to run our asynchronous code:

- microtasks
  - Run immediately after the call stack is empty
  - Can be created using `queueMicrotask` or `Promise.resolve().then`
- tasks
  - Run sometime in the future, when the event loop gets to it
  - Can be created using `setTimeout`, `MessageChannel` ports, event callbacks
- `requestAnimationFrame`
  - Runs before the next paint

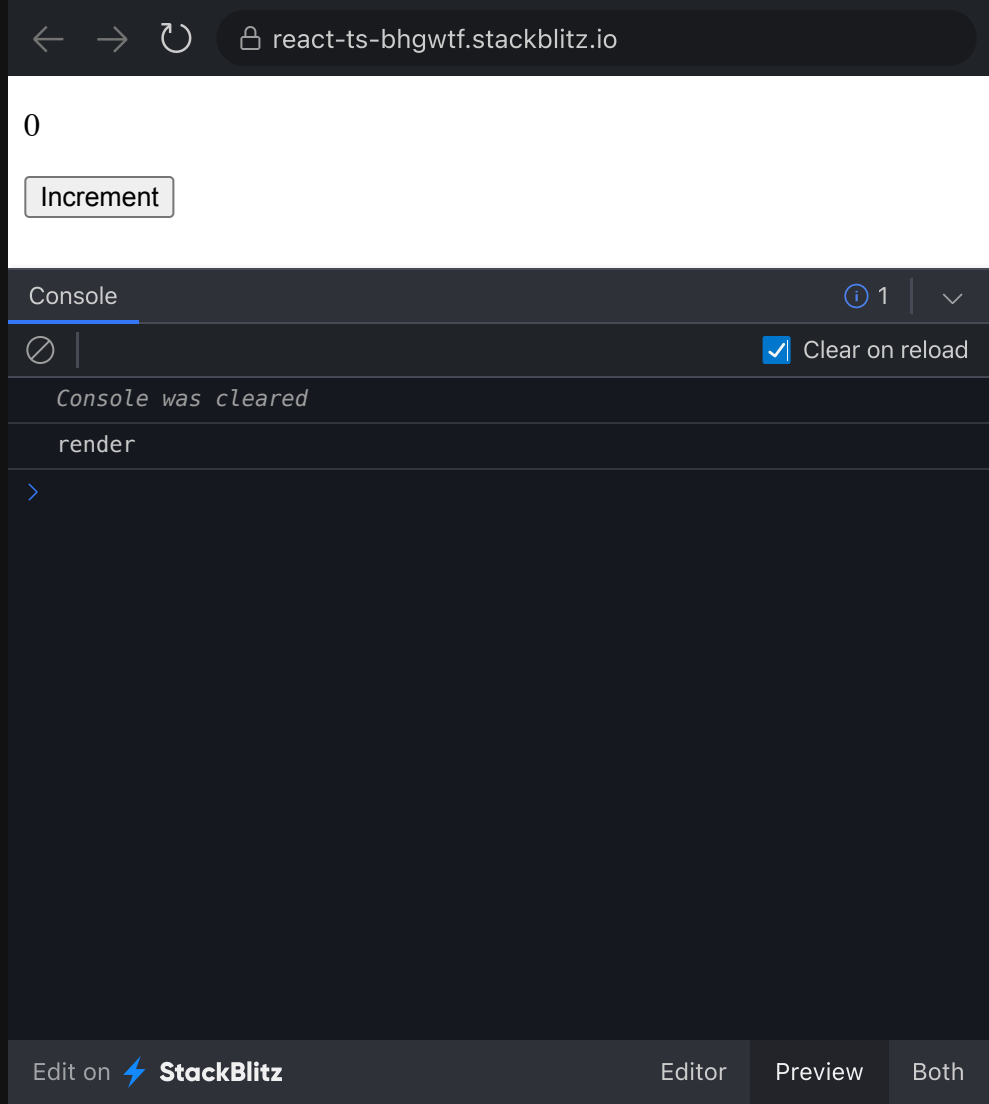
Which mechanism does React use?

Let's find out!

```
const asyncLog = (
  message: string,
  queueMechanism: (cb: () => void) => void
) => {
  queueMechanism(() => {
    console.log(message);
  });
};

const logMicrotask = (message: string) =>
  asyncLog(message, queueMicrotask);
const logTimeout = (message: string) =>
  asyncLog(message, (cb) => setTimeout(cb, 0));
const logAnimationFrame = (message: string) =>
  asyncLog(message, requestAnimationFrame);
```

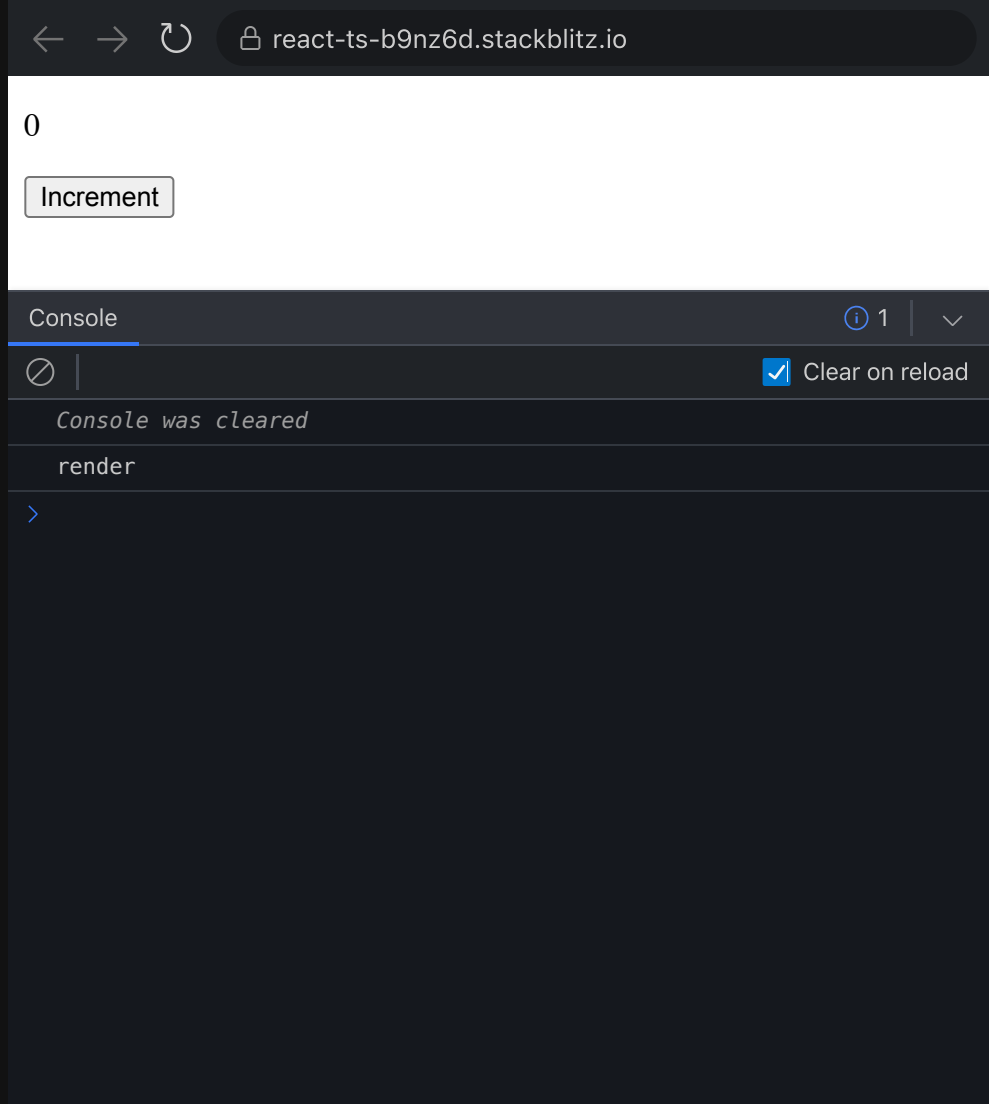
For most event callbacks, React uses microtasks

```
const Counter = () => {
  const [state, setState] = useState(0);
  const increment = () =>
    setState((current) => current + 1);

  const handler = () => {
    logMicrotask('before increment')
    increment()
    logMicrotask('after increment')
  };

  console.log('render');
  return (
    <div>
      <p>{state}</p>
      <button onClick={handler}>
        Increment
      </button>
    </div>
  );
}
```



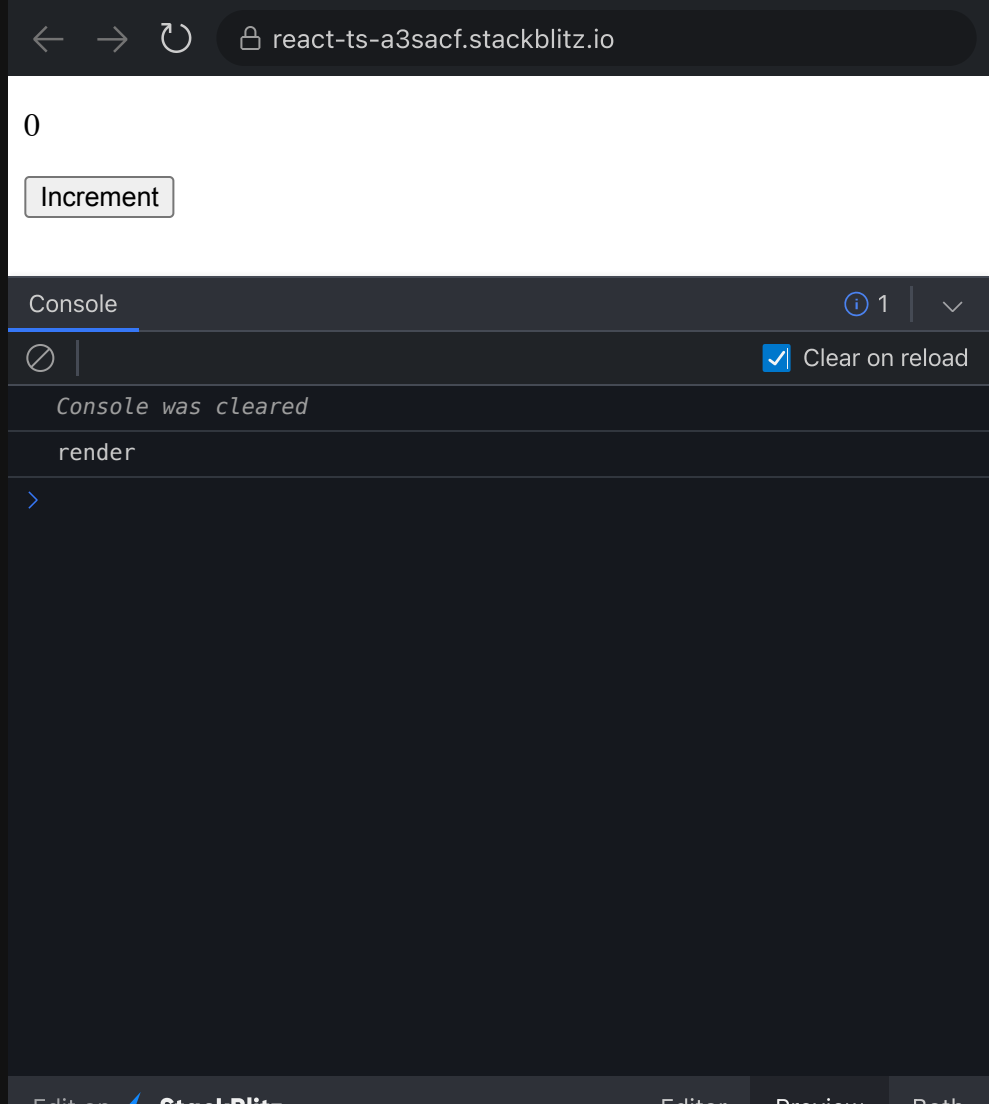
It doesn't matter how the callback was registered

```
const Counter = () => {
  const [state, setState] = useState(0);
  const increment = () =>
    setState((current) => current + 1);
  const handler = () => {
    logMicrotask('before increment');
    increment();
    logMicrotask('after increment');
  };
  const buttonRef = useRef(null);
  useEffect(() => {
    const btn = buttonRef.current;
    btn.addEventListener('click', handler);
    return () =>
      btn.removeEventListener('click', handler);
  });

  console.log('render');
  return (
    <div>
      <p>{state}</p>
      <button ref={buttonRef}>Increment</button>
    </div>
  );
}
```



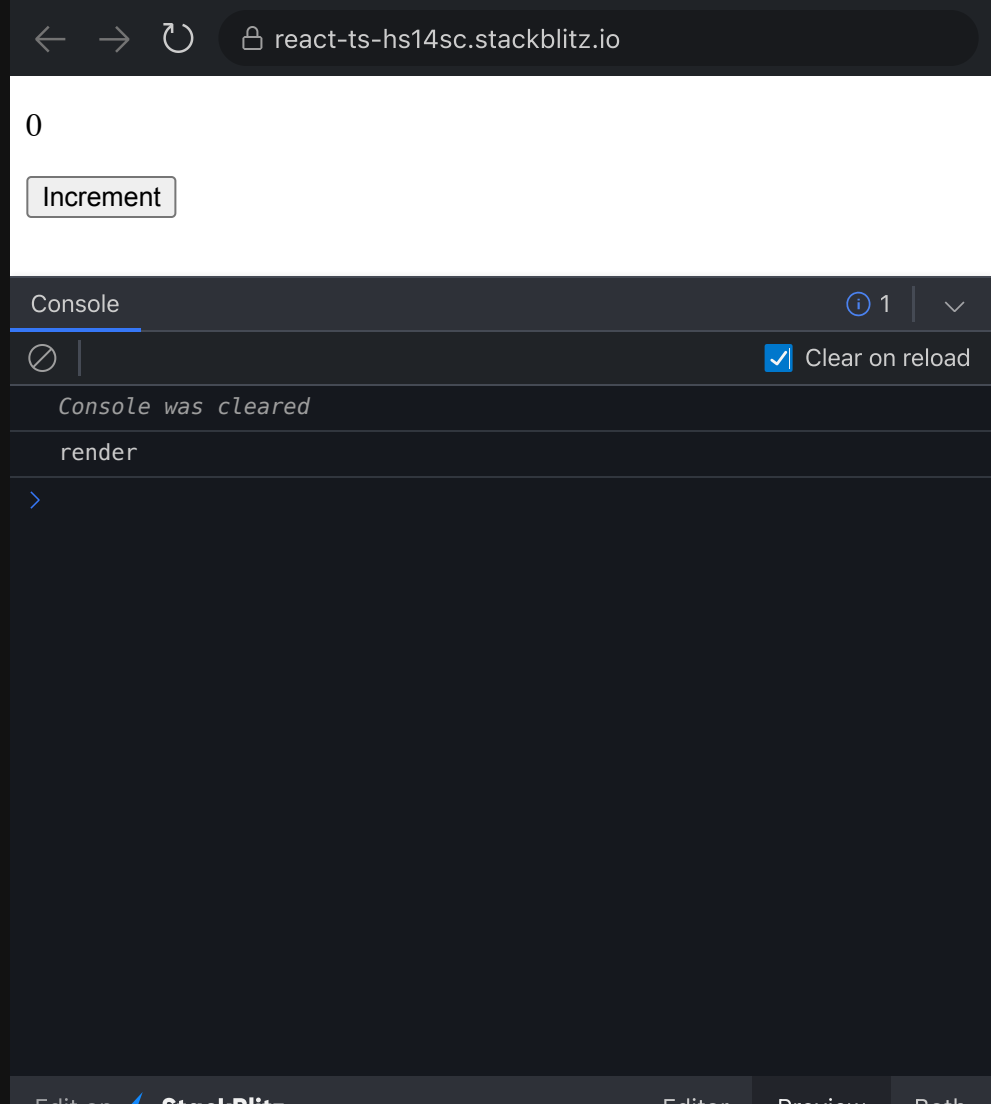
But it does matter when it was triggered

```
const Counter = () => {
  const [state, setState] = useState(0);
  const increment = () =>
    setState((current) => current + 1);

  const handler = () => {
    logMicrotask('before increment');
    increment();
    logMicrotask('after increment');
  };

  console.log('render');
  return (
    <div>
      <p>{state}</p>
      <button
        onClick={() => setTimeout(handler,0)}
      >
        Increment
      </button>
    </div>
  );
}
```



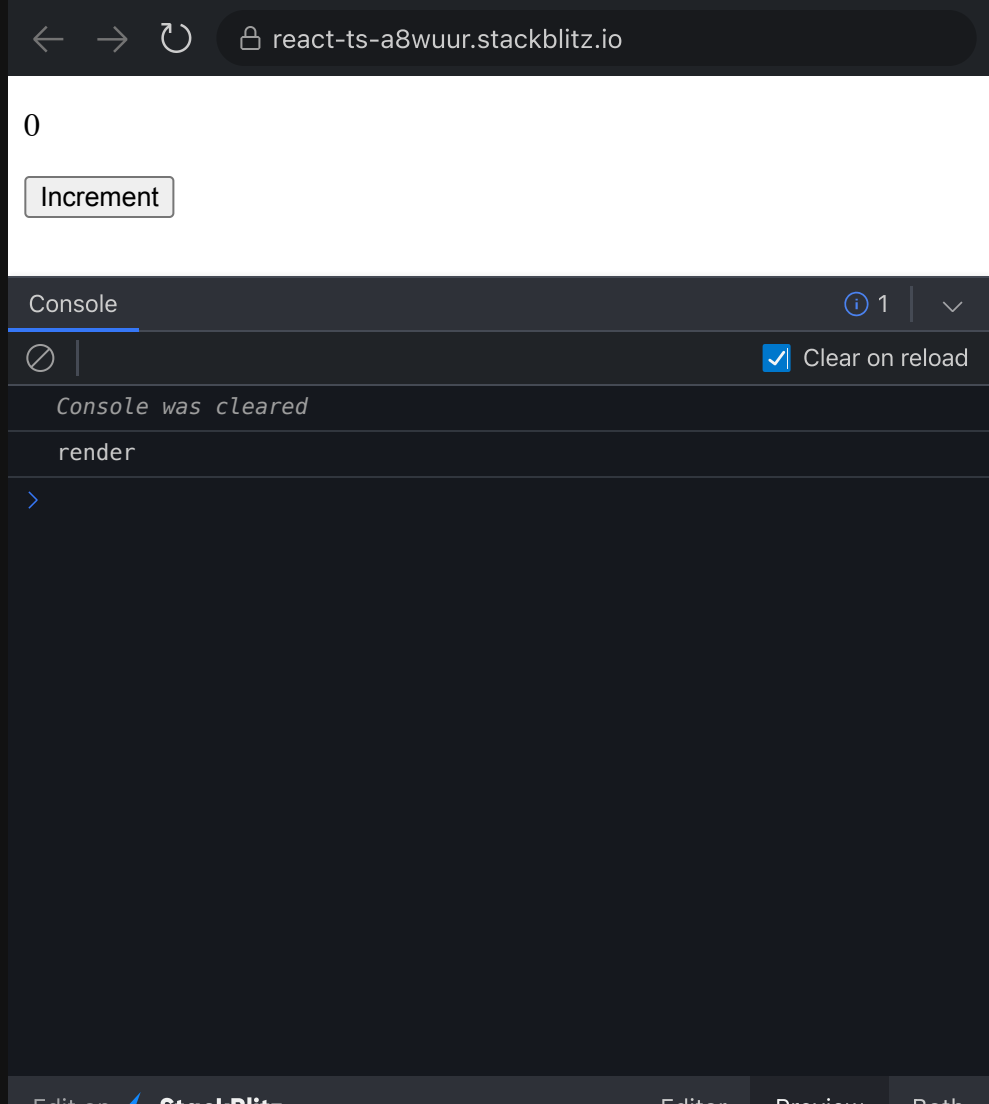
In which case React will use a timeout

```
const Counter = () => {
  const [state, setState] = useState(0);
  const increment = () =>
    setState((current) => current + 1);

  const handler = () => {
    logTimeout('before increment');
    increment();
    logTimeout('after increment');
  };

  console.log('render');
  return (
    <div>
      <p>{state}</p>
      <button
        onClick={() => setTimeout(handler,0)}
      >
        Increment
      </button>
    </div>
  );
}
```



It will also use the timeout for some events

```
const Counter = () => {
  const [state, setState] = useState(0);
  const increment = () =>
    setState((current) => current + 1);

  const handler = () => {
    logTimeout('before increment');
    increment();
    logTimeout('after increment');
  };

  console.log('render');
  return (
    <div>
      <p>{state}</p>
      <button
        onMouseEnter={handler}
      >
        Increment
      </button>
    </div>
  );
}
```



React state update timing

- For most event handlers, React will use microtasks
  - This ensures that all listeners for this event get triggered before the next render
- For other callbacks and some event handlers, React will schedule a task
  - This ensures that all pending tasks are handled before the next render

This also means, that this function will trigger 1 or 2 re-renders, depending on how it was called

```
const handleIncrement = async () => {
  await increment()
  await increment()
}
```

There is also an escape hatch. You can use `flushSync` from `react-dom` to synchronously update state

When to `flushSync`?

- If you need to call an imperative API after updating the state

```
const [items, setItems] = useState([])
...
const handleAdd = (newItem) => {
  flushSync(() => {
    setItems(current => [...current, newItem])
  })
  // these lines run after render
  const element = getElementForItem(newItem)
  element.scrollIntoView()
}
```

- If you need to integrate with a 3rd party library that exposes an imperative API
- Overall, should be used with care and not overused, as it opts-out of batching.
- `flushSync` may flush updates outside the callback when necessary to flush the updates inside the callback

There is another

Calling `setState` during a render

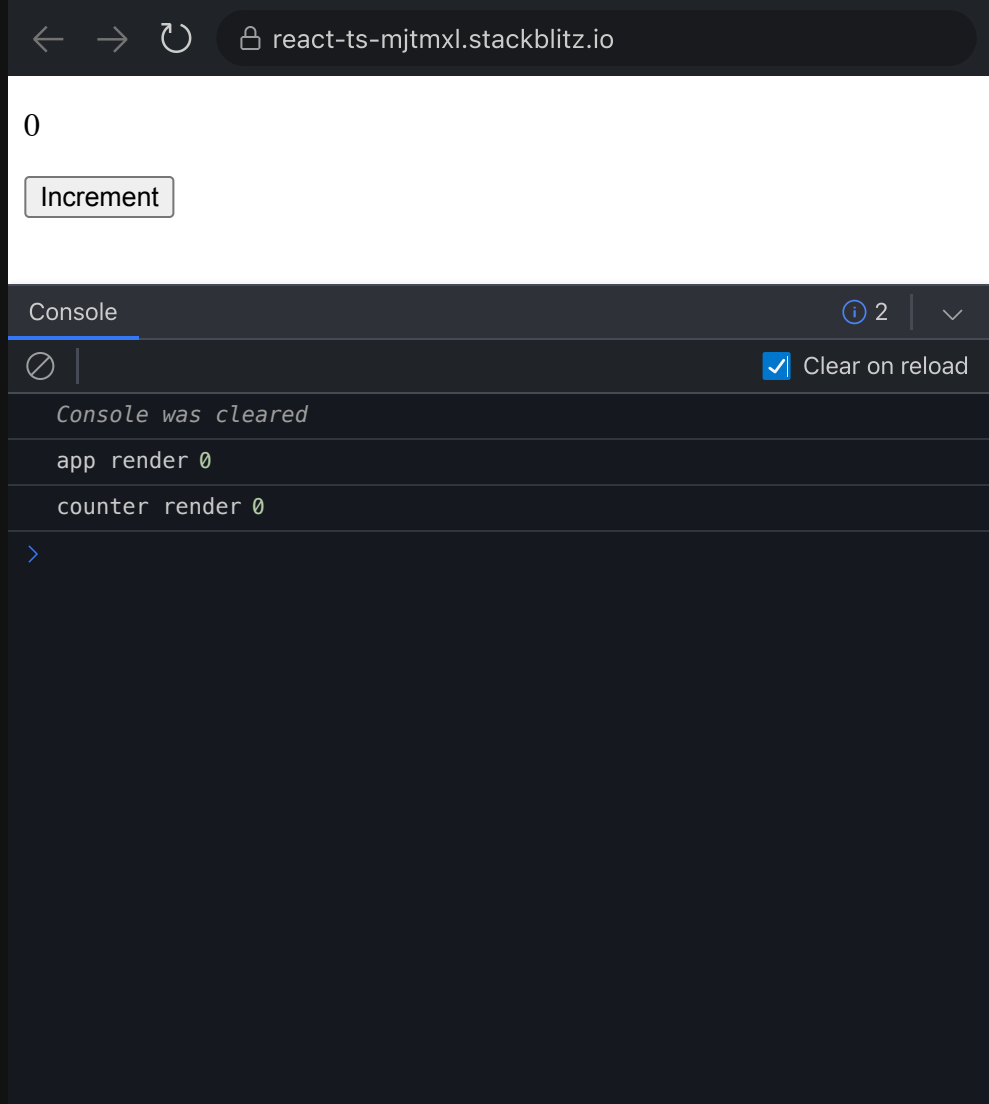
- Should be avoided,
- For derived state, you should use `useMemo` instead,
- Is only possible for the current component's state,
- In the rare cases where you actually need this, it is better than setting state in `useEffect`

```
const [state, setState] = useState(initial);
if(shouldUpdateState(deps)) {
  setState(newState)
}
```

```
const [state, setState] = useState(initial);
useEffect(() => {
  if(shouldUpdateState(deps)) {
    setState(newState)
  }
}, [deps])
```

Preventing unnecessary re-renders

```
const Counter = ({ count }) => {
  console.log('counter render', count);

  return <p>{count}</p>;
};

const App() {
  const [count, setCount] = useState(0);
  const increment = () => setCount((c) => c + 1);

  if(count % 2 === 1) {
    increment()
  }

  console.log('app render', count);
  return (
    <div>
      <Counter count={count} />
      <button onClick={increment}>
        Increment
      </button>
    </div>
  );
}
```



Concurrent mode

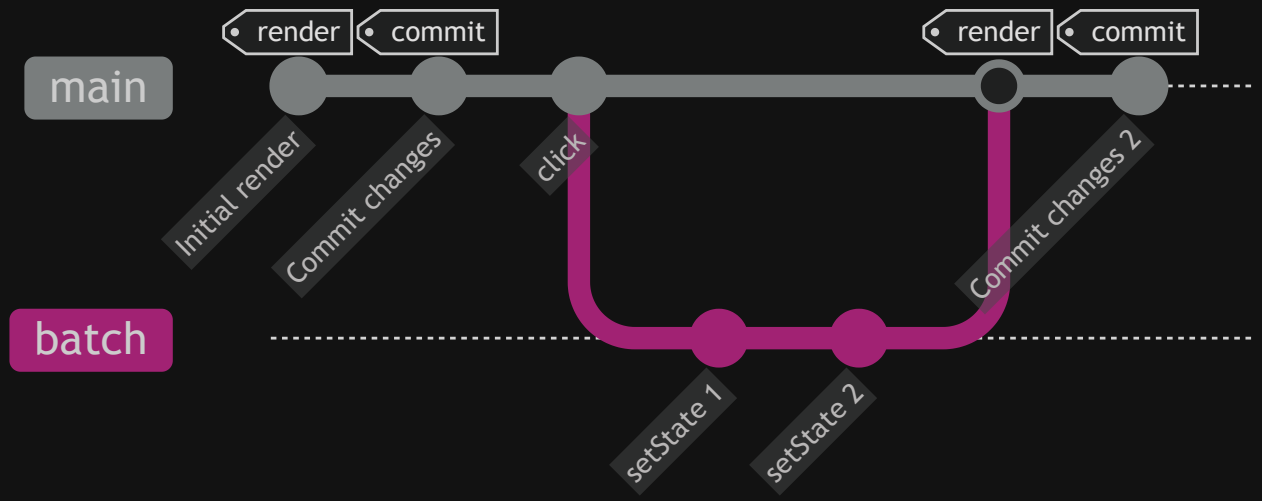
Concurrent mode allows the developer to mark some state updates as non-critical. These changes will be applied in a "forked" state tree, which will be "merged" back, when the background render is finished. React calls the fork a `transition` and provides the following tools:

- `startTransition` - useful to trigger a transition from any place, not necessarily a component.
  - Accepts a single argument, a callback which should call `setState`s to be processed in the background
- `useTransition` - useful to additinally know if we are transitioning
  - accepts no arguments, returns a `[isPending, startTransition]` tuple
- `useDeferredValue` - accepts a single argument and returns a value of the same type
  - the returned value is equal to the argument in the "fork"
  - it may be different in the "main" tree
  - it is a declarative version of `useTransition`

Updates wrapped in a transition never trigger a suspense. The transition will be suspended instead, hence this is a way of avoiding fallback content.

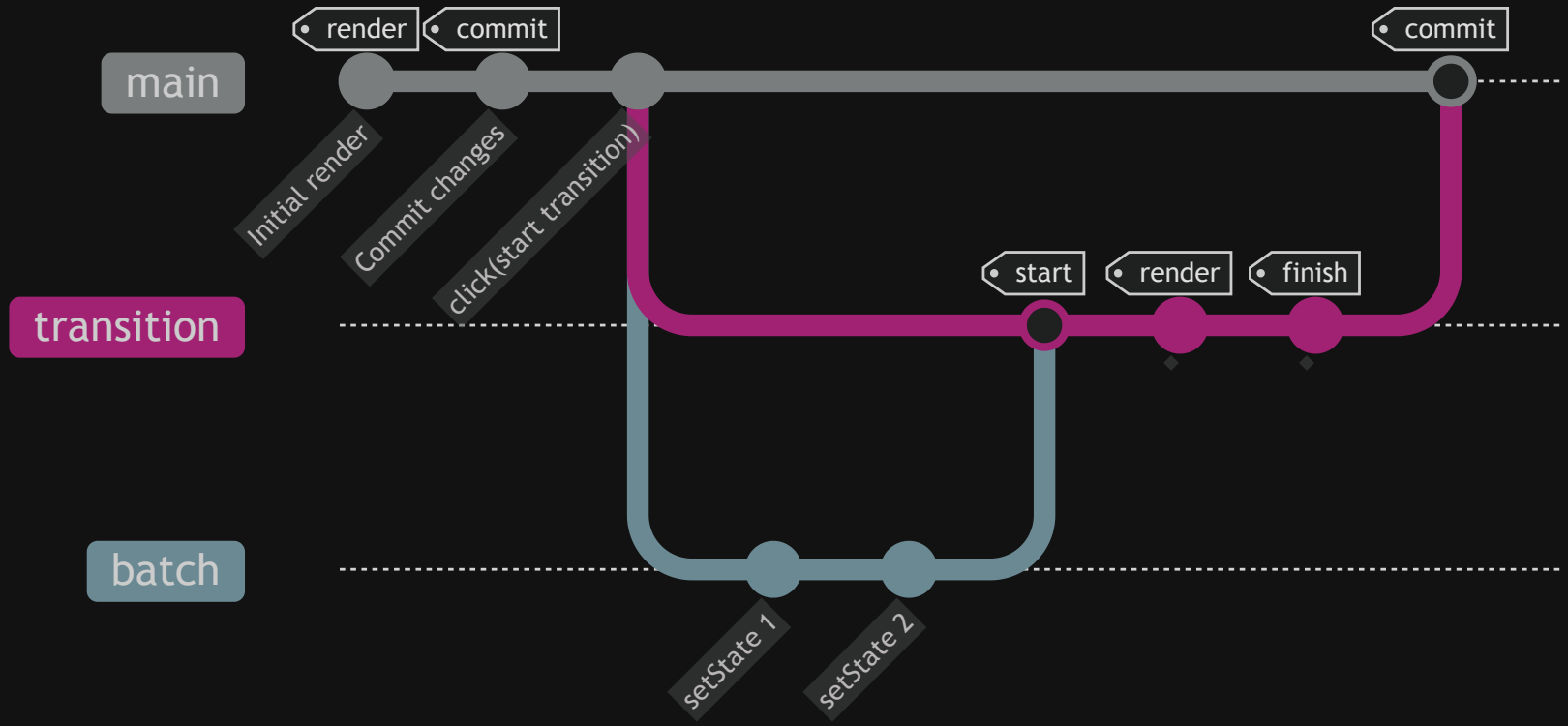
React state update, visualized

Regular update flow

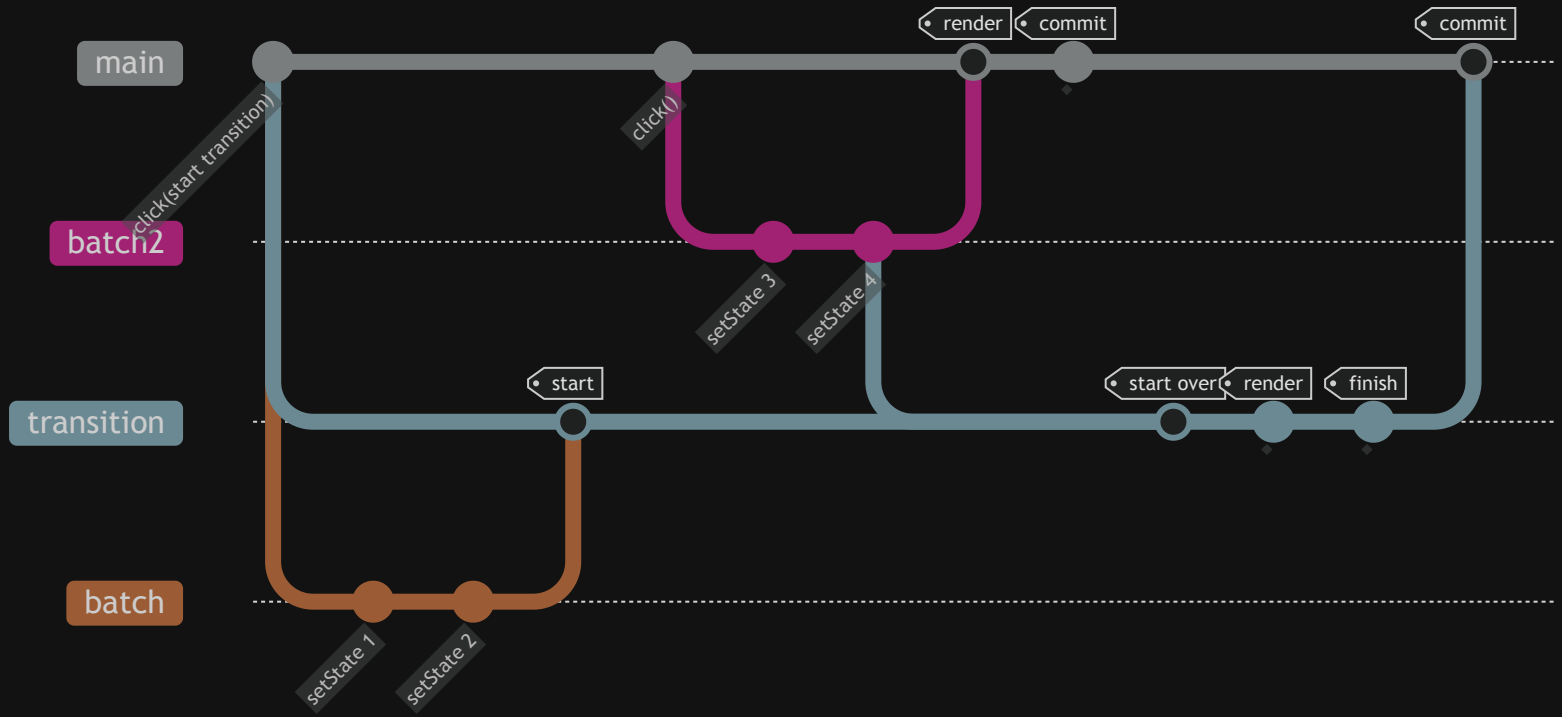


React state update, visualized

Concurrent update flow



State updates in concurrent mode



The above graph holds, as long as React knows about the state updates.

Synchronising external state

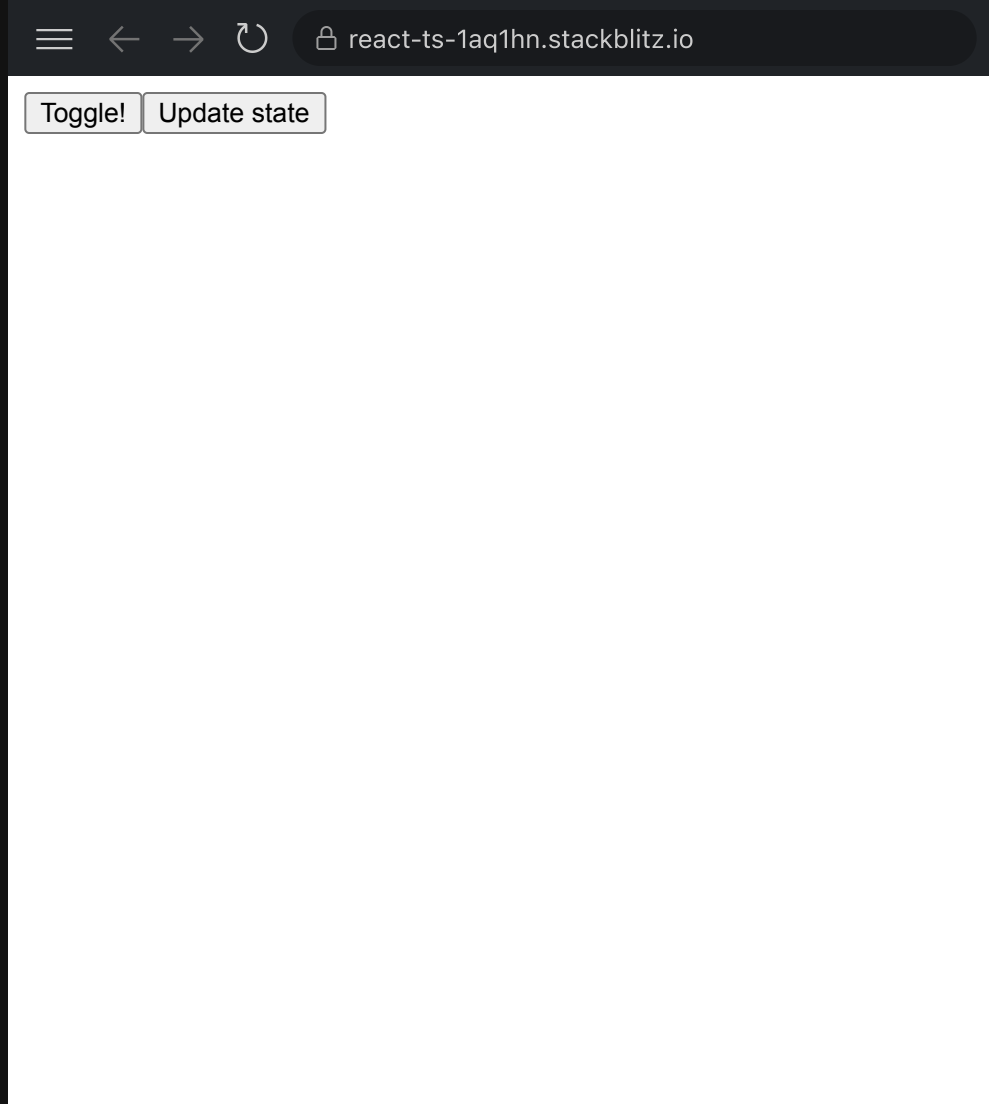
Sometimes our state grows and we want to use better tooling to manage it. Or we need to use it also outside of React. How to keep our tree in sync?

This is how we used to do it. But it causes a very subtle bug in concurrent mode.

```
import store from './store'

export const useStore = () => {
  const [state, setState] = useState(store.getState());

  useEffect(() => {
    const handleStoreUpdate = () => setState(store.getState());
    const unsubscribe = store.subscribe(handleStoreUpdate);

    return unsubscribe
  }, []);

  return state;
}
```

Tearing

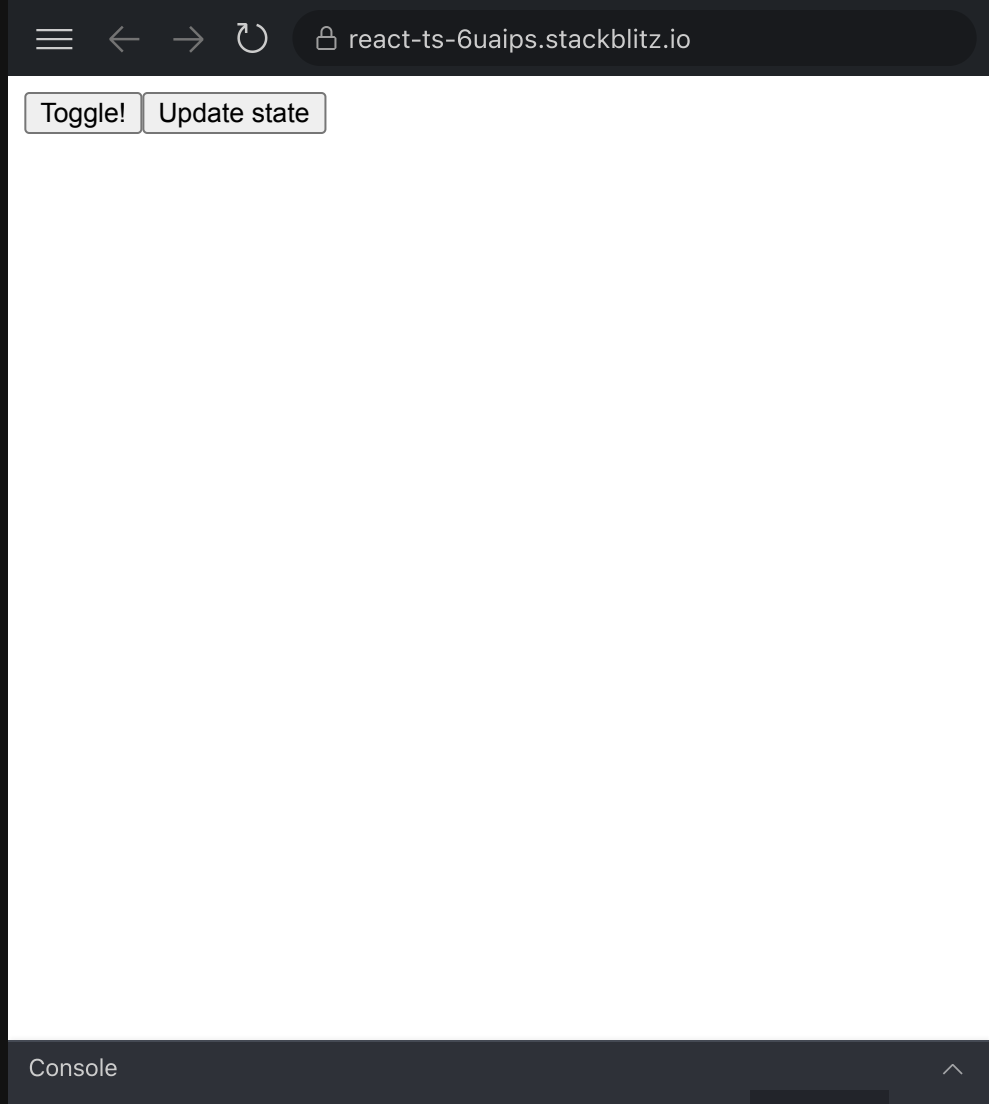
```
const [show, setShow] = useState(false);
const updateStore = () => {
  update(Date.now());
};
const handleClick = () => {
  startTransition(() => {
    setShow((current) => !current);
  });
};
return (
  <div>
    <button onClick={handleClick}>Toggle!</button>
    <button onClick={updateStore}>
      Update state
    </button>
    {show && (
      <div>
        <SlowComponent />
        <SlowComponent />
        <SlowComponent />
        <SlowComponent />
        <SlowComponent />
      </div>
    )}
  </div>
```



Tearing, fixed

`useSyncExternalStore` is a hook which should be used to synchronise external state with React.

- It accepts three arguments:
  - `subscribe` - it takes a single `callback` which subscribes to the store and returns an `unsubscribe` method
  - `getSnapshot` - it returns the store's current state
  - (optional) `getServerSnapshot` - used for SSR & initial hydration
- It returns the current state of the store
- It ensures that it is always in sync with the store



Sources

1. React docs (https://beta.reactjs.org/)
2. In the loop (https://youtu.be/cCOL7MC4Pl0/)
3. React concurrency discussion (https://github.com/reactwg/react-18/discussions/70)