Avoid these 4 useState() Mistakes in React

Chetan Mahajan

@_chetanmahajan

1. Overusing useState

While useState is a powerful tool, overusing it can lead to a cluttered and difficult-to-maintain codebase.

Try to group related state variables into a single state object instead of having multiple useState calls.

Chetan Mahajan
@_chetanmahajan

Avoid this

```
const [title, setTitle] = useState("");
const [description, setDescription] = useState("");
const [location, setLocation] = useState("")
```

Do this

```
const [formState, setFormState] = useState({
  title: "",
  description: "",
  location: "",
});
```

Chetan Mahajan
@_chetanmahajan

2. Failing to optimize re-renders

When a state variable is updated, React will re-render the component and its children.

This can lead to performance issues if not managed properly.

Consider using memoization techniques like React.memo or useMemo to optimize re-renders.

Chetan Mahajan
@_chetanmahajan

❌ Avoid this

```jsx
function MyComponent({ data }) {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <ExpensiveComponent data={data} />
    </div>
  );
}
```

✅ Do this

```jsx
const MemoizedExpensiveComponent = React.memo(ExpensiveComponent);

function MyComponent({ data }) {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <MemoizedExpensiveComponent data={data} />
    </div>
  );
}
```

3. Ignoring the initial state

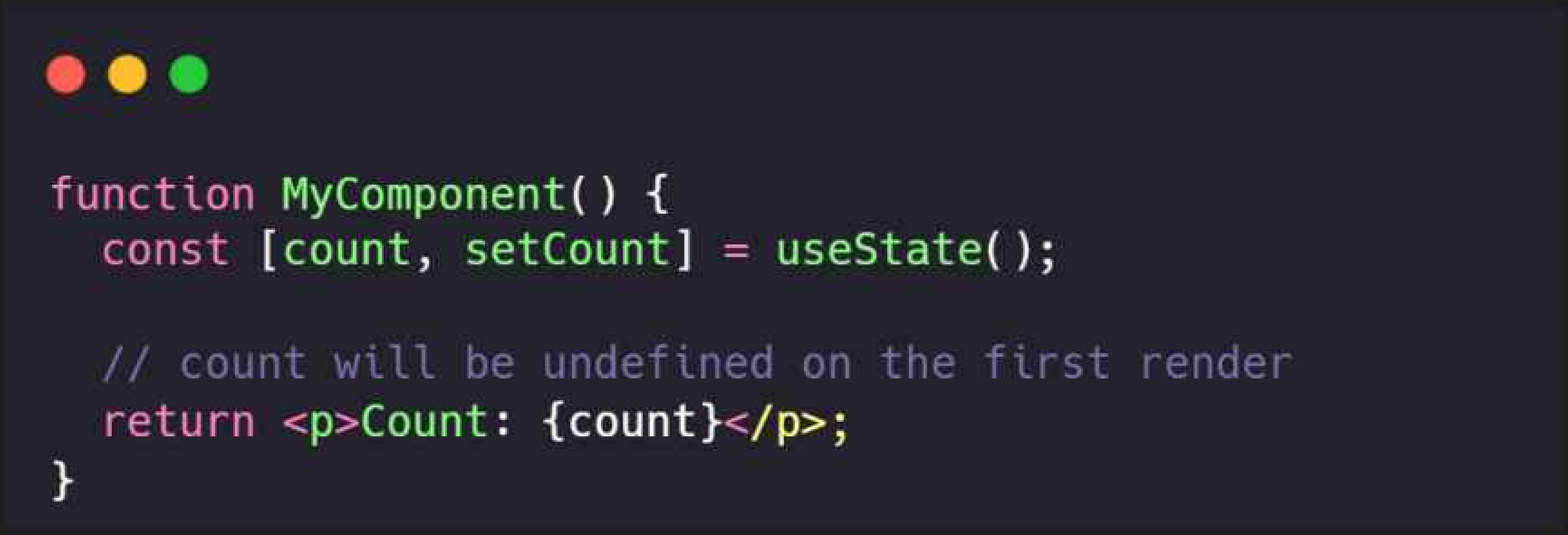
The initial state passed to useState is only used on the first render.

Subsequent updates will use the new state value.

Make sure to provide a meaningful initial state.

Chetan Mahajan
@_chetanmahajan

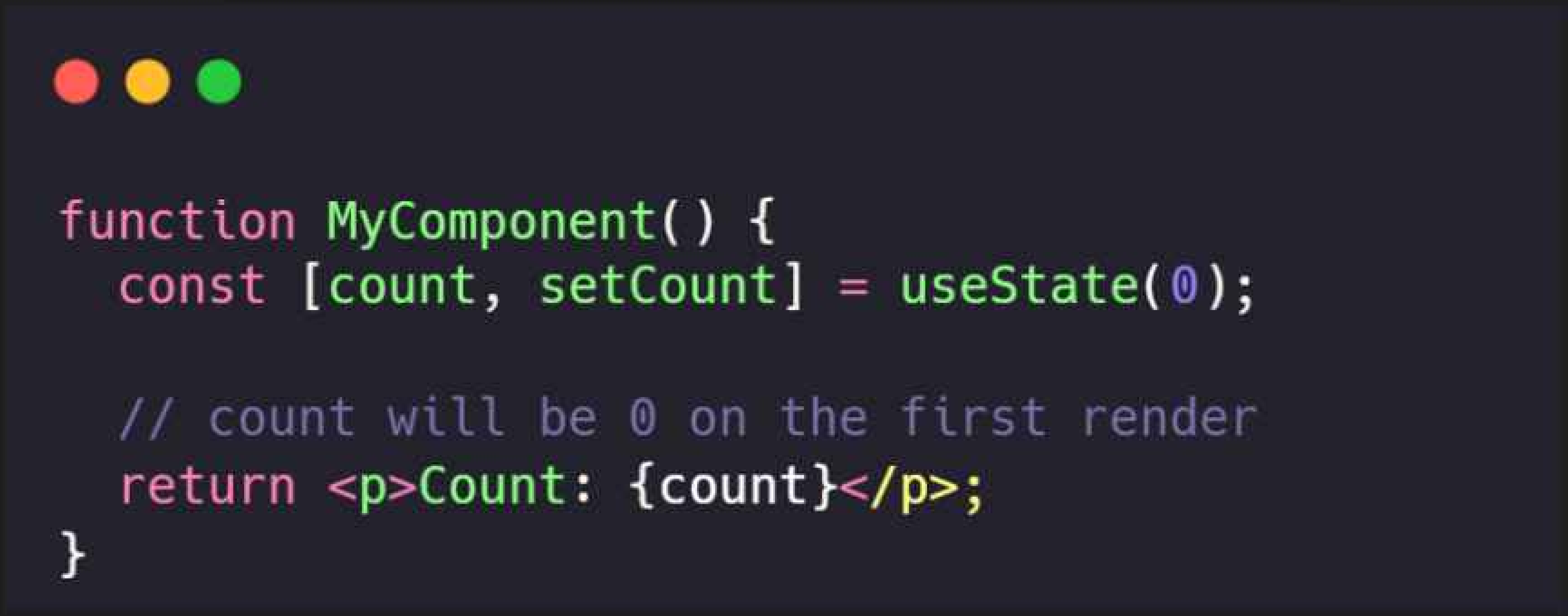
Avoid this



```
function MyComponent() {
  const [count, setCount] = useState();

  // count will be undefined on the first render
  return <p>Count: {count}</p>;
}
```

Do this



```
function MyComponent() {
  const [count, setCount] = useState(0);

  // count will be 0 on the first render
  return <p>Count: {count}</p>;
}
```

4. Mixing state management strategies

Avoid mixing useState with other state management libraries like Redux or MobX.

This can lead to confusion and make the codebase harder to maintain.

Choose a single state management strategy and stick to it.

Avoid this

```
function MyComponent() {
  const [count, setCount] = useState(0);
  const dispatch = useDispatch();

  // Mixing useState and Redux
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <button onClick={() => dispatch(increment())}>Increment (Redux)
</button>
    </div>
  );
}
```

Do this

```
function MyComponent() {
  const count = useSelector((state) => state.count);
  const dispatch = useDispatch();

  // Using only Redux
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => dispatch(increment())}>Increment</button>
    </div>
  );
}
```

Save this for later.

If you like it repost to share it!