4 `useState` Mistakes You Should Avoid in React🚫

Jayanth babu S · Follow

Published in Level Up Coding · 5 min read · Jan 26, 2024

4 useState Mistakes in in React

Introduction

React.js has become a cornerstone of modern web development, with its unique approach to managing state within components. One common hook, `useState`, is fundamental but often misused. Understanding and avoiding these common mistakes is crucial for both beginners and experienced developers aiming to create efficient and bug-free applications.

This blog will dive into four critical mistakes to avoid when using `useState` in React. Let's enhance our React skills together!

Before diving in, explore more in-depth articles on web development at my personal website:

Program With Jayanth
Program With Jayanth is a dedicated space for aspiring and seasoned front-end developers. Here, Jayanth shares...
programwithjayanth.com

Mistake 1: Forgetting to Consider the Previous State 😨

When working with React's `useState` hook, a common mistake is not taking into account the most recent state when updating it. This oversight can lead to unexpected behaviors, particularly when you're dealing with rapid or multiple state updates.

❌ Understanding the Issue

Let's imagine you're building a counter in React. Your goal is to increase the count each time a button is clicked. A straightforward approach might be to simply add 1 to the current state value. However, this can be problematic.

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

const CounterComponent = () => {
  const [counter, setCounter] = useState(0);

  const incrementCounter = () => {
    setCounter(counter + 1); // Might not always work as expected
  };

  return (
    <div>
      <p>Counter: {counter}</p>
      <button onClick={incrementCounter}>Increment</button>
    </div>
  );
};

export default CounterComponent;
```

In the above code, `incrementCounter` updates the counter based on its current value. This seems straightforward but can lead to issues. React might batch multiple `setCounter` calls together, or other state updates might interfere, resulting in the `counter` not being updated correctly every time.

✅ The Correction:

To avoid this issue, use the functional form of the `setCounter` method. This version takes a function as its argument, which React calls with the most recent state value. This ensures that you're always working with the latest value of the state.

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

const CounterComponent = () => {
  const [counter, setCounter] = useState(0);

  const incrementCounter = () => {
    setCounter(prevCounter => prevCounter + 1); // Correctly updates based on th
  };
```

```
    return (
      <div>
        <p>Counter: {counter}</p>
        <button onClick={incrementCounter}>Increment</button>
      </div>
    );
  };

  export default CounterComponent;
```

In this corrected code, `incrementCounter` uses a function to update the state. This function receives the most recent state (`prevCounter`) and returns the updated state. This approach is much more reliable, especially when updates happen rapidly or multiple times in a row.

Mistake 2: Neglecting State Immutability 🧊

❌ Understanding the Issue

In React, state should be treated as immutable. A common mistake is directly mutating the state, especially with complex data structures like objects and arrays.

Consider this faulty approach with a stateful object:

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

  const ProfileComponent = () => {
```

```
  const [profile, setProfile] = useState({ name: 'John', age: 30 });

  const updateAge = () => {
    profile.age = 31; // Directly mutating the state
    setProfile(profile);
  };

  return (
    <div>
      <p>Name: {profile.name}</p>
      <p>Age: {profile.age}</p>
      <button onClick={updateAge}>Update Age</button>
    </div>
  );
};

export default ProfileComponent;
```

This code incorrectly mutates the `profile` object directly. Such mutations don't trigger re-renders and lead to unpredictable behaviors.

✅ The Correction:

Always create a new object or array when updating state to maintain immutability. Use the spread operator for this purpose.

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

const ProfileComponent = () => {
  const [profile, setProfile] = useState({ name: 'John', age: 30 });

  const updateAge = () => {
    setProfile({...profile, age: 31}); // Correctly updating the state
  };
  return (
    <div>
      <p>Name: {profile.name}</p>
      <p>Age: {profile.age}</p>
      <button onClick={updateAge}>Update Age</button>
    </div>
  );
};

export default ProfileComponent;
```

In the corrected code, `updateAge` uses the spread operator to create a new `profile` object with the updated age, preserving state immutability.

Mistake 3: Misunderstanding Asynchronous Updates ⏳

❌ Understanding the Issue

React's state updates via `useState` are asynchronous. This often leads to confusion, especially when multiple state updates are made in quick succession. Developers might expect the state to change immediately after a `setState` call, but in reality, React batches these updates for performance reasons.

Let's look at a common scenario where this misunderstanding can cause problems

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

```
const AsyncCounterComponent = () => {
  const [count, setCount] = useState(0);

  const incrementCount = () => {
    setCount(count + 1);
    setCount(count + 1);
    // Developer expects count to be incremented twice
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={incrementCount}>Increment Count</button>
    </div>
  );
};

export default AsyncCounterComponent;
```

In this example, the developer intends to increment the `count` twice. However, due to the asynchronous nature of state updates, both `setCount` calls are based on the same initial state, resulting in the `count` being incremented only once.

✅ The Correction:

To handle asynchronous updates correctly, use the functional update form of `setCount`. This ensures that each update is based on the most recent state.

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

const AsyncCounterComponent = () => {
  const [count, setCount] = useState(0);
```

```
    const incrementCount = () => {
      setCount(prevCount => prevCount + 1);
      setCount(prevCount => prevCount + 1);
      // Now each update correctly depends on the most recent state
    };
    // Optional: Use useEffect to see the updated state
    useEffect(() => {
      console.log(count); // 2
    }, [count]);

    return (
      <div>
        <p>Count: {count}</p>
        <button onClick={incrementCount}>Increment Count</button>
      </div>
    );
  };

  export default AsyncCounterComponent;
```

In the above code, each call to `setCount` uses the most recent value of the state, ensuring accurate and sequential updates. This approach is crucial for operations that depend on the current state, especially when multiple state updates occur in quick succession.

Mistake 4: Misusing State for Derived Data 📊

❌ Understanding the Issue

A frequent error is using state for data that can be derived from existing state or props. This redundant state can lead to complex and error-prone code.

For example:

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

const GreetingComponent = ({ name }) => {
  const [greeting, setGreeting] = useState(`Hello, ${name}`);

  return (
    <div>{greeting}</div>
  );
};

export default GreetingComponent;
```

Here, `greeting` state is unnecessary as it can be derived directly from `name`.

✅ The Correction:

Instead of using state, derive data directly from existing state or props.

```
import React from 'react';

const GreetingComponent = ({ name }) => {
  const greeting = `Hello, ${name}`; // Directly derived from props

  return (
    <div>{greeting}</div>
  );
};

export default GreetingComponent;
```

In the corrected code, `greeting` is computed directly from the `name` prop, simplifying the component and avoiding unnecessary state management.

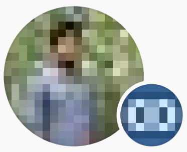
Conclusion 🚀

Effectively using the `useState` hook in React is crucial for building reliable and efficient applications. By understanding and avoiding common mistakes —like neglecting the previous state, mismanaging state immutability, overlooking asynchronous updates, and avoiding redundant state for derived data—you can ensure smoother and more predictable component behavior. Keep these insights in mind to enhance your React development journey and create more robust applications.

Enjoyed this article? For more in-depth discussions and insights on web development, visit my personal blog at Program With Jayanth.

Happy coding!!

Technology React JavaScript Programming Software Development



Written by Jayanth babu S

Follow

2.4K Followers · Writer for Level Up Coding

https://programwithjayanth.com/ I am a Senior Front-End Engineer with years of experience and specialized in Front-End Web technologies.

More from Jayanth babu S and Level Up Coding

Jayanth babu S in JavaScript in Plain English

JavaScript Interview: Can You Stop or Break a forEach Loop? 🛑

Introduction

4 min read · Dec 28, 2023

1.4K 24

Somnath Singh in Level Up Coding

The Era of High-Paying Tech Jobs is Over

The Death of Tech Jobs.

✦ · 14 min read · Apr 1, 2024

2.3K 90

Tirendaz AI in Level Up Coding

How to Use ChatGPT in Daily Life?

Save time and money using ChatGPT

9 min read · Apr 4, 2023

6.2K 110

Jayanth babu S in Stackademic

How to Change the Default Port Number in React: React Tips

Introduction 🌟

3 min read · Nov 17, 2023

166

See all from Jayanth babu S

See all from Level Up Coding

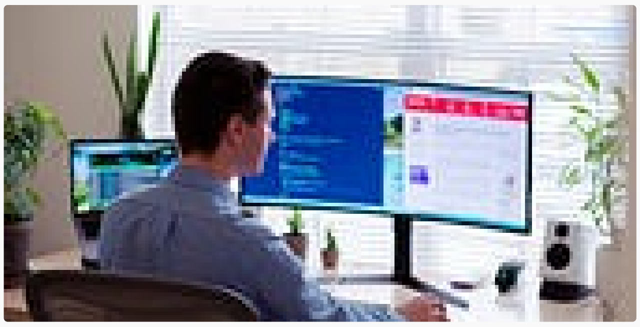
Reed Barger

📚 React Libraries You Should Use In 2024

I'm going to show you all of the libraries that I would recommend you use in 2024 to build...

9 min read · Feb 27, 2024

710 4



Hammad Hassan

I Tried Upwork For 30 Days

I challenged myself to earn $100 in the first month, and it turned out earning more than...

✦ · 4 min read · Feb 17, 2024

3.9K 86

Lists

General Coding Knowledge
20 stories · 1092 saves

Stories to Help You Grow as a Software Developer
19 stories · 961 saves

Coding & Development
11 stories · 549 saves

ChatGPT prompts
47 stories · 1385 saves

Kristiyan Velkov in Stackademic

5 React Interview Questions. I bet the senior front-end developer...

React JS top questions for the senior Front-end developers — Advanced features,...

✦ · 7 min read · Feb 20, 2024

104 2

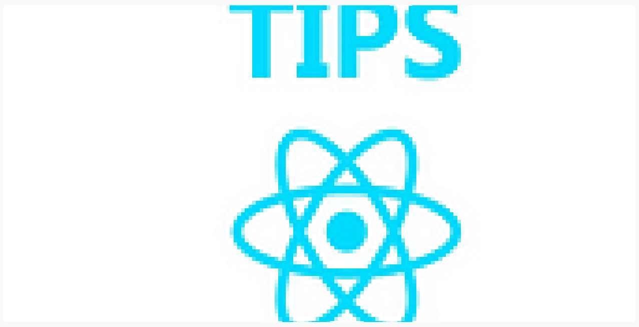
Techmade

JavaScript Concepts for Senior Software Engineer Interviews

Ready to level up your JavaScript skills? You're on an exciting path! JavaScript reigns...

✦ · 5 min read · Jan 3, 2024

529 2



Khushi_developer

10 Expert Performance Tips Every Senior JS React Developer Should...

As a senior Javascript React developer, consistently improving the performance of...

Dorian Szafranski

NEXT.js is the REAL MONSTER

Hi guys! Today I gonna talk about NEXT.js as a greatest monster of frameworks like React,...

3 min read · Dec 5, 2023

841 6

5 min read · Jan 29, 2024

414 13

See more recommendations