REACT HOOKS Mistakes

React Hooks Mistake - 1

Using State When You Don't Need It

Mistake: Using state for values that don't change over time can lead to unnecessary re-renders, which impacts performance.

```
// Mistake: using state for a constant value
const MyComponent = () => {
  const [greeting, setGreeting] = useState("Hello, World!");
  return <h1>{greeting}</h1>;
};
```

Correct Approach: Only use useState when a value needs to change over time. For static or unchanging values, use constants or variables outside of state.

```
// Correction: using a constant for unchanging values
const MyComponent = () => {
  const greeting = "Hello, World!";
  return <h1>{greeting}</h1>;
};
```

React Hooks Mistake - 2

Not Using Function Version of useState

Mistake: When setting state based on the previous state, not using the function version can lead to bugs, especially when the state updates rapidly or in response to user input.

```
const Counter = () => {
  const [count, setCount] = useState(0);
  const increment = () => {
    setCount(count + 1); // This might cause issues if setCount is called multiple times
  };
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
};
```

Correct Approach: When updating state based on the current state, always use the function version of setState, like setCount((prevCount) => prevCount + 1).

```
const Counter = () => {
  const [count, setCount] = useState(0);
  const increment = () => {
    setCount((prevCount) => prevCount + 1); // Correct way to use the previous state
  };
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
};
```

React Hooks Mistake - 3

Expecting Immediate State Updates

Mistake: React's useState is asynchronous. If you expect state to update immediately, your code might behave unexpectedly.

```
const Counter = () => {
  const [count, setCount] = useState(0);
  const handleClick = () => {
    setCount(count + 1);
    console.log(count); // This will log the old value, not the updated count
  };
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={handleClick}>Increment</button>
    </div>
  );
};
```

Correct Approach: To see the updated value of state after it changes, use useEffect to run code that depends on state updates.

```
const Counter = () => {
  const [count, setCount] = useState(0);
  const handleClick = () => {
    setCount(count + 1);
  };
  useEffect(() => {
    console.log(count); // Logs the updated count whenever count changes
  }, [count]);
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={handleClick}>Increment</button>
    </div>
  );
};
```

React Hooks Mistake - 4

Unnecessary useEffect

Mistake: Adding useEffect for code that could run outside of an effect can lead to dependency issues, additional renders, and bugs.

```
const MyComponent = ({ message }) => {
  const [displayMessage, setDisplayMessage] = useState("");
  useEffect(() => {
    setDisplayMessage(message);
  }, [message]); // useEffect is unnecessary here
  return <p>{displayMessage}</p>;
};
```

Correct Approach: Only use useEffect for side effects (e.g., fetching data, subscribing to events), not for directly setting or passing props/state in the render.

```
const MyComponent = ({ message }) => {
  // Directly render the message, no useEffect needed
  return <p>{message}</p>;
};
```

React Hooks Mistake - 5

Referential Equality Mistakes

Mistake: Using objects or arrays directly in the dependency array causes the effect to re-run on every render because objects and arrays in JavaScript are reference-checked, not value-checked.

```
const MyComponent = () => {
  const [data, setData] = useState([]);
  useEffect(() => {
    // fetch or process data
  }, [{}]); // Mistake: {} creates a new object on every render
};
```

Correct Approach: Avoid putting newly created objects or arrays directly in the dependency array. Use useMemo or constants where possible.

```
const MyComponent = () => {
  const [data, setData] = useState([]);
  useEffect(() => {
    // fetch or process data
  }, []); // Use an empty dependency array to run only once
};
```

React Hooks Mistake - 6

Not Aborting Fetch Requests

Mistake: Not cleaning up fetch requests when a component unmounts can lead to memory leaks and unwanted behavior, especially with asynchronous requests.

```
const MyComponent = () => {
  useEffect(() => {
    fetch("https://api.example.com/data")
      .then((response) => response.json())
      .then((data) => console.log(data));
  }, []);
  return <div>Fetching Data...</div>;
};
```

Correct Approach: Use AbortController to cancel fetch requests when the component unmounts, which prevents memory leaks.

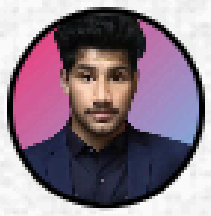
```
const MyComponent = () => {
  useEffect(() => {
    const controller = new AbortController(); // Create a controller
    fetch("https://api.example.com/data", { signal: controller.signal })
      .then((response) => response.json())
      .then((data) => console.log(data))
      .catch((error) => {
        if (error.name === "AbortError") {
          console.log("Fetch aborted");
        }
      });
    return () => controller.abort(); // Abort fetch on component unmount
  }, []);
  return <div>Fetching Data...</div>;
};
```

For

React Interview Questions

You can find link in above post

Happy Learning



Shubham Gupta

Sharing Insights on Cracking Tech Interviews