🔗 Connect with Me:
🌐 LinkedIn : https://www.linkedin.com/in/priya-bagde
🗁 GitHub : https://github.com/priya42bagde
💻 LeetCode : https://leetcode.com/priya42bagde/
🎥 YouTube Channel : https://youtube.com/channel/UCK1_Op30_pZ1zBs9l3HNyBw (Priya Frontend Vlogz)

useReducer:

What's useReducer:

- In React, there are two main hooks that we can use **for state management: useState and useReducer hooks.**
- useReducer Hook is an **alternative** to useState Hook and we use it when the React components **need to be optimized** or when the **next state value is dependent upon the previous state value**.
- useReducer Hook allows developers to **handle complex state manipulations (handling multiple states that rely on complex logic) & updates.** It is best used on more complex data, **specifically, arrays or objects.**
- With useReducer, you **can avoid passing down callbacks through different levels of your component**. Instead, useReducer allows you **to pass a provided dispatch function**, which in turn will improve performance for components that trigger deep updates.

Syntax: const [state, dispatch] = useReducer(reducer, initialState, init);

useReducer hook takes in **<u>three arguments</u>** including the **reducer function, the initial state object, and the third argument is init function** <u>to load the initial state lazily</u>. useReducer hook returns an array consisting of **the current state and the dispatch functional method**.

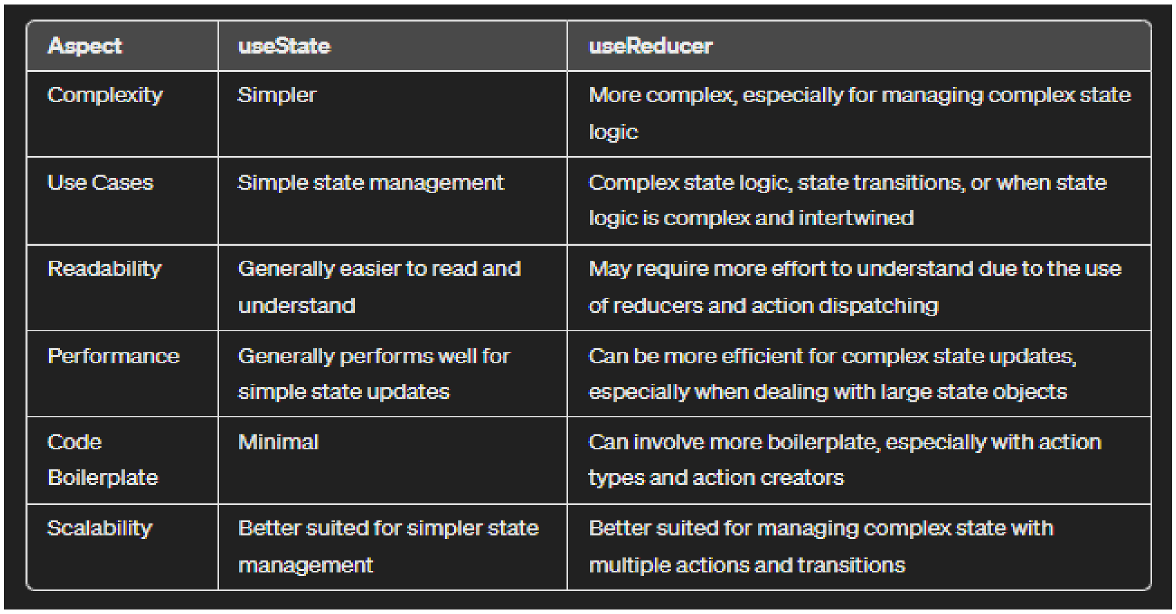
- state: It represents the **current state value** managed by the useReducer() hook. **Initially, it is set to the initialState** value provided.
- dispatch: In simpler terms, dispatching means **a request to update the state**. It is a **function** that is used **to send actions to the reducer**. An **<u>action</u>** object is an **object that describes how to update the state**. Typically, the action object has a **property "Type" and a payload** to be used by the reducer.
- reducer: The reducer is a **pure function**, that receives the **current state and an action as arguments** and **returns a new state**.
- initialState: The initial state is the **second argument** passed to the useReducer Hook, which represents the **default state**.
- init: It's an **optional** third argument and it's a **function, not just an array or object**. It **extracts the logic for calculating the initial state outside the reducer**. This is also handy for **resetting the state**. **If you don't pass a third argument to useReducer, it will take the second argument as the initial state.** React **saves the initial state once and ignores it on the next renders**. Although the result **<u>of init function is only used for the initial render,</u>** you're **<u>still calling this function on every render</u>**. To solve this, you may pass it as an init function to useReducer as the third argument instead. Notice that you're passing init, **<u>which is the function itself, and not init(),</u>** which is the result of calling it. This way, **<u>the initial state does not get re-created after initialization</u>**.

Introduction to Reducers/reducer function:

- Reducers are a way **to reduce the component's state logic to a single function that is stored outside the component.** You **can place all of your state logic from different functions here**. It **simplifies** your component and reduces complexity by separating state logic into its own function. Now, **rather than invoking the handler functions, we'll now dispatch an action**.
- Having two **parameters**: **a state object and an action object**. The **state object includes the current state** that will be modified, and the **action object describes the operations that will be performed in the current state.**
- It **returns the updated state object**, which is then **applied to the original state object**, and the component is rerendered to show the latest changes, just like in the useState hook.
- <u>Keep in mind that you must **return** something from the reducer. **If no actions were found, return the default state. Otherwise, if nothing was returned, the current state value will be set to undefined**.</u>

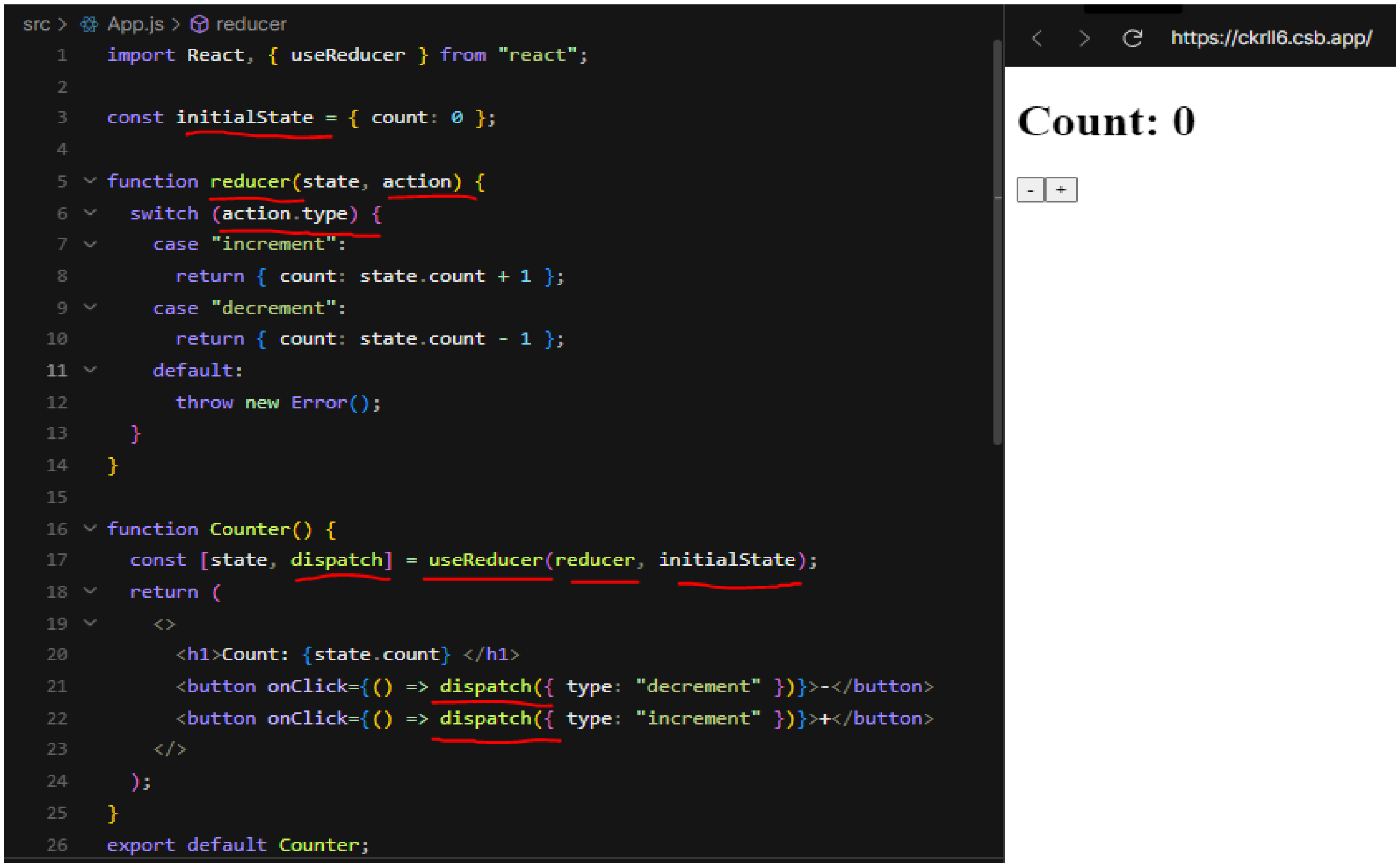
- A function is considered **pure**, if it adheres to the following rules:
  1.The function always returns the same output if the same arguments are passed in.
  2.The function does not produce any side-effects (doesn't modify variables outside of their scope, mutate state, or perform any I/O operations).

useState vs useReducer: useReducer is very similar to useState, but it lets you **move the state update logic from event handlers into a single function outside of your component**.



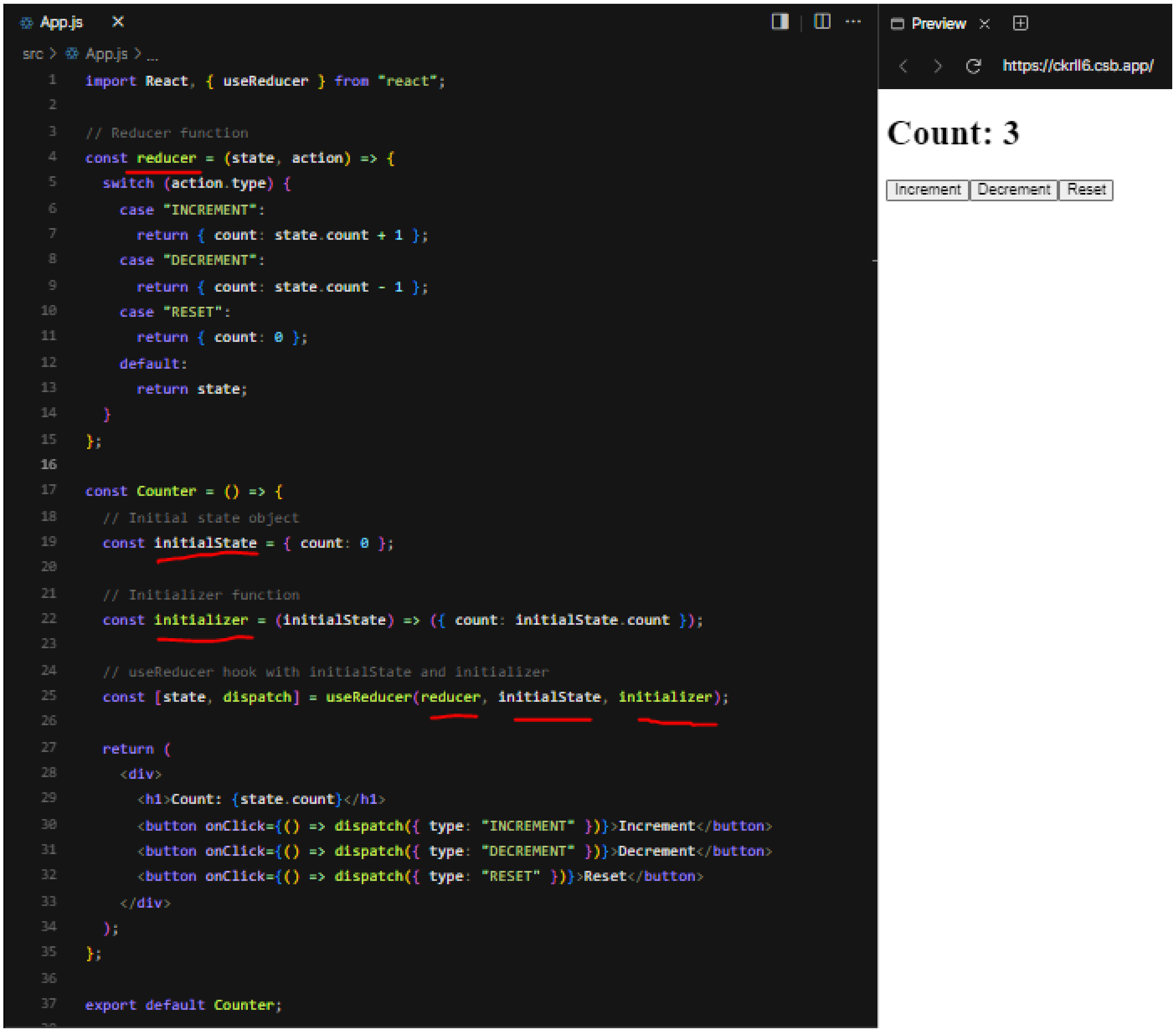
Aspect	useState	useReducer
Complexity	Simpler	More complex, especially for managing complex state logic
Use Cases	Simple state management	Complex state logic, state transitions, or when state logic is complex and intertwined
Readability	Generally easier to read and understand	May require more effort to understand due to the use of reducers and action dispatching
Performance	Generally performs well for simple state updates	Can be more efficient for complex state updates, especially when dealing with large state objects
Code Boilerplate	Minimal	Can involve more boilerplate, especially with action types and action creators
Scalability	Better suited for simpler state management	Better suited for managing complex state with multiple actions and transitions

Example:



```
import React, { useReducer } from "react";

const initialState = { count: 0 };

function reducer(state, action) {
  switch (action.type) {
    case "increment":
      return { count: state.count + 1 };
    case "decrement":
      return { count: state.count - 1 };
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      <h1>Count: {state.count} </h1>
      <button onClick={() => dispatch({ type: "decrement" })}>-</button>
      <button onClick={() => dispatch({ type: "increment" })}>+</button>
    </>
  );
}
export default Counter;
```

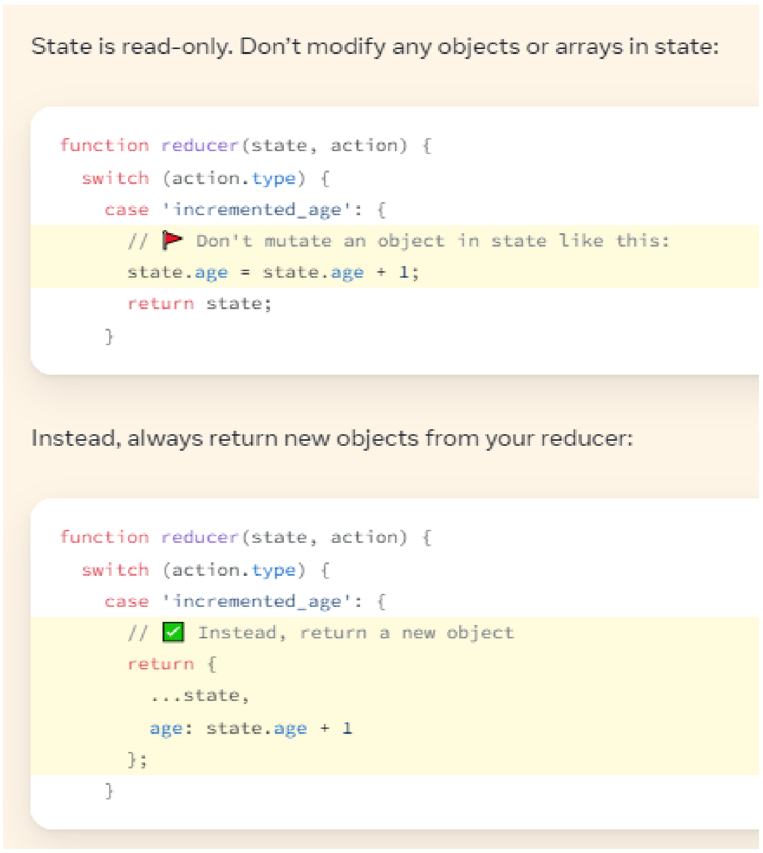
```
Count: 0
```



```
import React, { useReducer } from "react";

// Reducer function
const reducer = (state, action) => {
  switch (action.type) {
    case "INCREMENT":
      return { count: state.count + 1 };
    case "DECREMENT":
      return { count: state.count - 1 };
    case "RESET":
      return { count: 0 };
    default:
      return state;
  }
};

const Counter = () => {
  // Initial state object
  const initialState = { count: 0 };

  // Initializer function
  const initializer = (initialState) => ({ count: initialState.count });

  // useReducer hook with initialState and initializer
  const [state, dispatch] = useReducer(reducer, initialState, initializer);

  return (
    <div>
      <h1>Count: {state.count}</h1>
      <button onClick={() => dispatch({ type: "INCREMENT" })}>Increment</button>
      <button onClick={() => dispatch({ type: "DECREMENT" })}>Decrement</button>
      <button onClick={() => dispatch({ type: "RESET" })}>Reset</button>
    </div>
  );
};

export default Counter;
```

Key Points:

- The **dispatch function only updates the state variable for the next render**. If you read the state variable after calling the dispatch function, you will still get the old value that was on the screen before your call.
- React checks the difference between the new and the current state to determine whether the state has been updated. If the **new value you provide is identical to the current state**, as determined by **an Object.is comparison**, React **will skip re-rendering the component and its children**. This is an **optimization**. React may still need to call your component before ignoring the result, but it shouldn't affect your code.
- **Do not mutate the current state directly.**


State is read-only. Don't modify any objects or arrays in state:

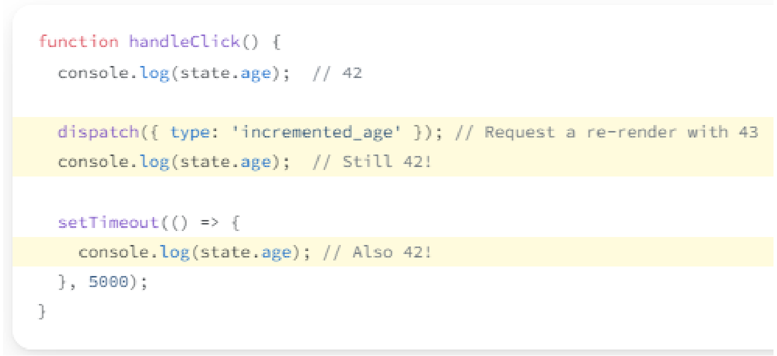
```
function reducer(state, action) {
  switch (action.type) {
    case 'incremented_age': {
      // 🚩 Don't mutate an object in state like this:
      state.age = state.age + 1;
      return state;
    }
```

Instead, always return new objects from your reducer:

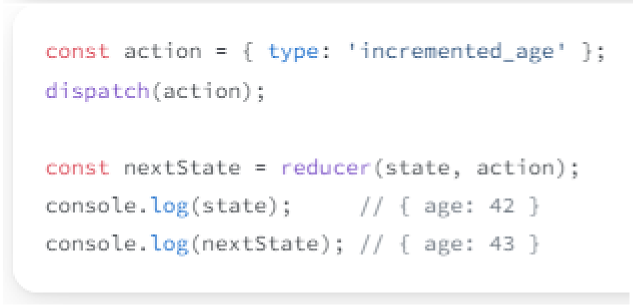
```
function reducer(state, action) {
  switch (action.type) {
    case 'incremented_age': {
      // ✅ Instead, return a new object
      return {
        ...state,
        age: state.age + 1
      };
    }
```

Common Mistakes Can Happen:

1. **<u>I've dispatched an action, but logging gives me the old state value:</u>** This is because states behaves like a **snapshot**. Updating state **requests another render with the new state value** but **does not affect the state variable in your already-running event handler**. If you need to guess the next state value, you can calculate it **manually by calling the reducer** yourself:



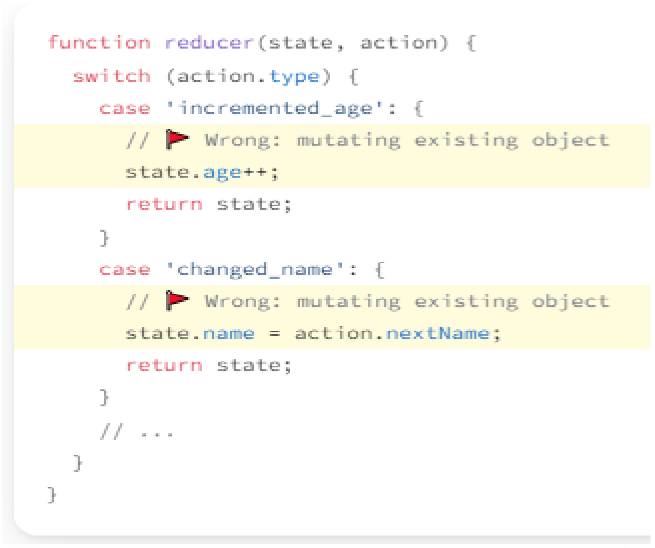
```
function handleClick() {
  console.log(state.age);  // 42

  dispatch({ type: 'incremented_age' }); // Request a re-render with 43
  console.log(state.age);  // Still 42!

  setTimeout(() => {
    console.log(state.age); // Also 42!
  }, 5000);
}
```



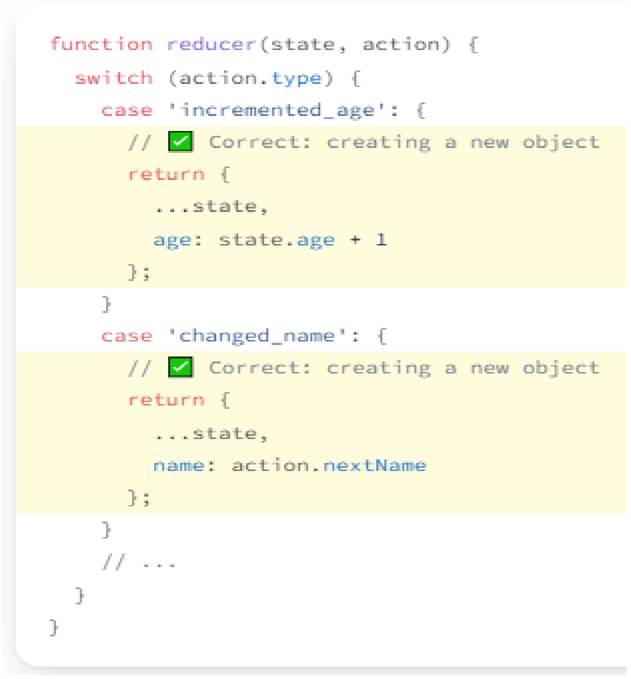
```
const action = { type: 'incremented_age' };
dispatch(action);

const nextState = reducer(state, action);
console.log(state);     // { age: 42 }
console.log(nextState); // { age: 43 }
```

2. **<u>I've dispatched an action, but the screen doesn't update:</u>** React will ignore your update **if the next state is equal to the previous state,** as determined by an Object.is comparison. This usually happens **when you change an object or an array in state directly**. You **mutated an existing state object and returned it, so React ignored the**

update. To fix this, you need to ensure that you're always updating objects in state and updating arrays in state instead of mutating them.

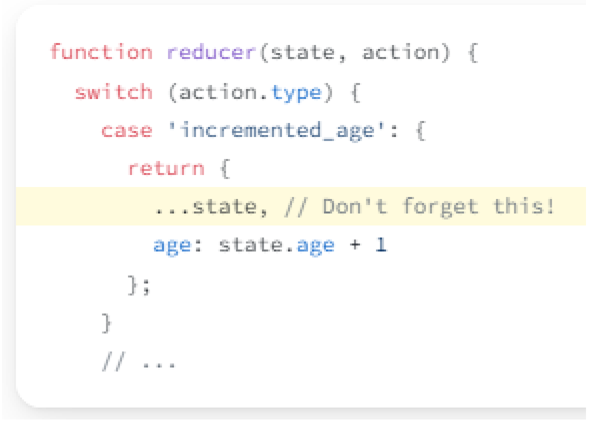


```
function reducer(state, action) {
  switch (action.type) {
    case 'incremented_age': {
      // 🚩 Wrong: mutating existing object
      state.age++;
      return state;
    }
    case 'changed_name': {
      // 🚩 Wrong: mutating existing object
      state.name = action.nextName;
      return state;
    }
    // ...
  }
}
```



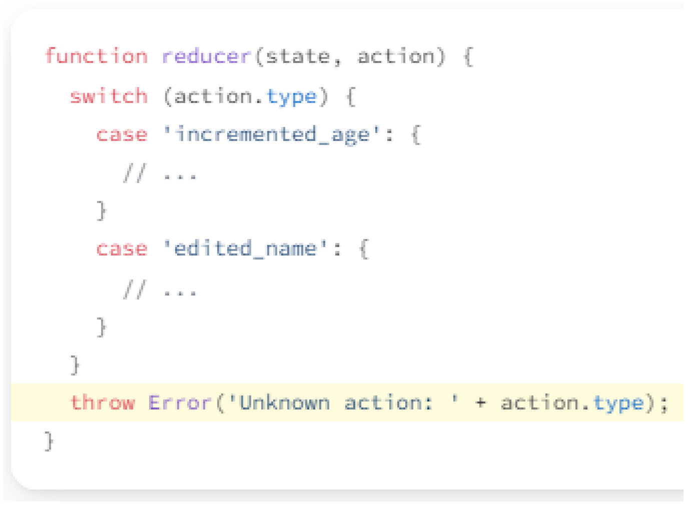
```
function reducer(state, action) {
  switch (action.type) {
    case 'incremented_age': {
      // ✅ Correct: creating a new object
      return {
        ...state,
        age: state.age + 1
      };
    }
    case 'changed_name': {
      // ✅ Correct: creating a new object
      return {
        ...state,
        name: action.nextName
      };
    }
    // ...
  }
}
```

3. **A part of my reducer state becomes undefined after dispatching**: Make sure that every case **branch copies all of the existing fields when returning the new state**. **Without ...state** above, the returned next state would only contain the age field and nothing else.



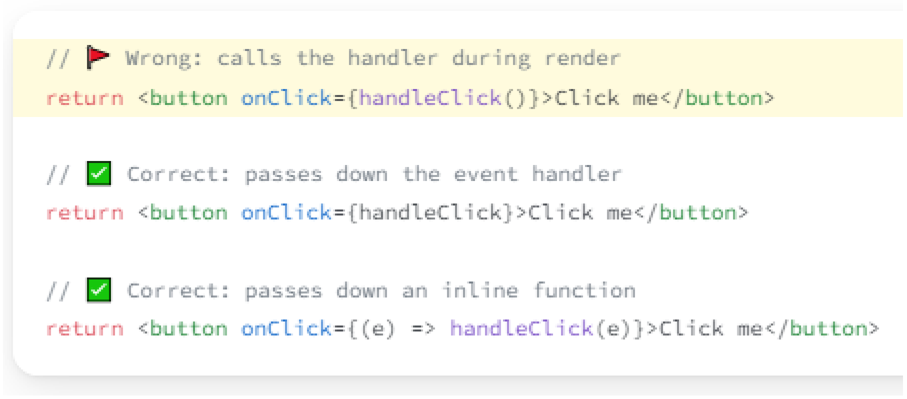
```
function reducer(state, action) {
  switch (action.type) {
    case 'incremented_age': {
      return {
        ...state, // Don't forget this!
        age: state.age + 1
      };
    }
    // ...
```

4. **My entire reducer state becomes undefined after dispatching:** If your state unexpectedly becomes undefined, you’re **likely forgetting to return state in one of the cases**, or your **action type doesn’t match any of the case statements. To find why, throw an error outside the switch**.



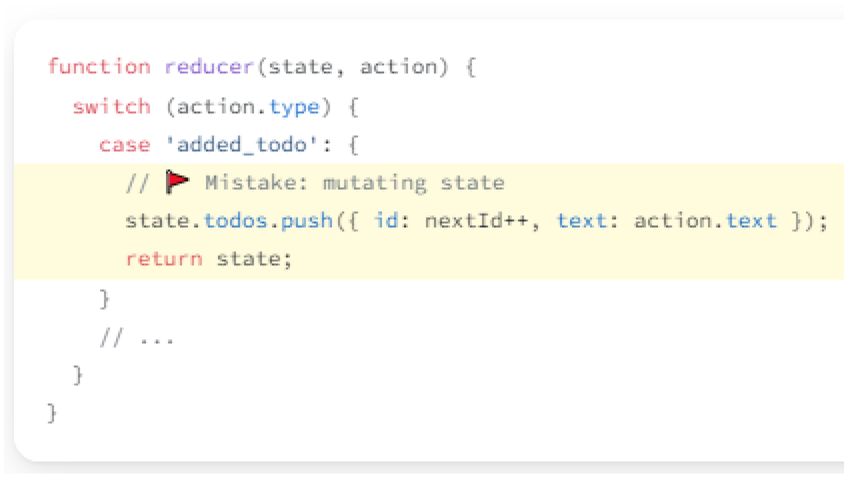
```
function reducer(state, action) {
  switch (action.type) {
    case 'incremented_age': {
      // ...
    }
    case 'edited_name': {
      // ...
    }
  }
  throw Error('Unknown action: ' + action.type);
}
```

5. I’m getting an error: “Too many re-renders”: React limits the **number of renders to prevent an infinite loop**. Typically, this means that **you’re unconditionally dispatching an action during render**, so your component enters a loop: render, dispatch (which causes a render), render, dispatch (which causes a render), and so on. Very often, this is caused by a **mistake in specifying an event handler**. This is specific to **dispatch function call responsible for the error.**

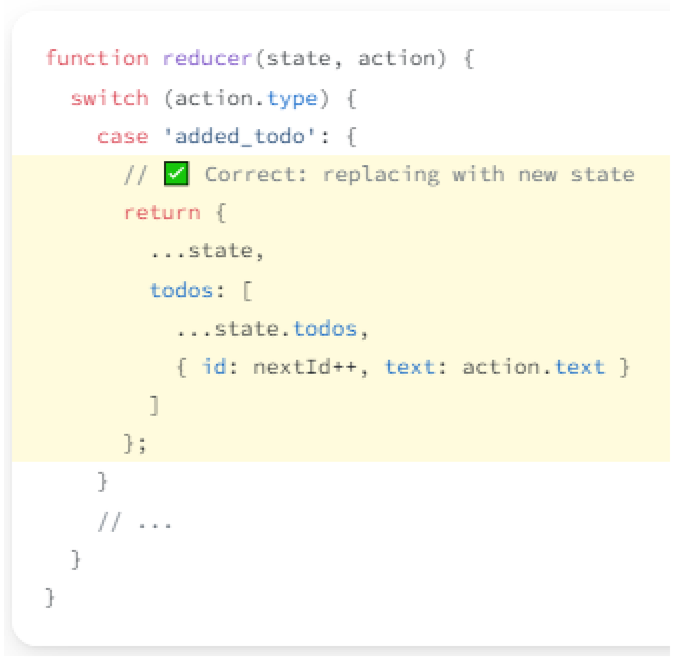


```
// 🚩 Wrong: calls the handler during render
return <button onClick={handleClick()}>Click me</button>

// ✅ Correct: passes down the event handler
return <button onClick={handleClick}>Click me</button>

// ✅ Correct: passes down an inline function
return <button onClick={(e) => handleClick(e)}>Click me</button>
```

6. **My reducer or initializer function runs twice:** In **Strict Mode**, React will call your **reducer and initializer functions twice**. This development-only behavior helps you keep components pure. React uses the result of one of the calls, and ignores the result of the other call. As long as your component, initializer, and reducer functions are pure, this shouldn’t affect your logic. However, if they are accidentally impure, this helps you notice the mistakes. For example, this impure reducer function mutates an array in state, Because React calls your reducer function twice, you’ll see the todo was added twice, so you’ll know that there is a mistake. In this example, you can fix the mistake by replacing the array instead of mutating it. Now that this **reducer function is pure**, calling it an extra time doesn’t make a difference in behavior. This is why React calling it twice helps you find mistakes. **Only component, initializer, and reducer functions need to be pure**. **Event handlers don’t need to be pure, so React will never call your event handlers twice.**



```
function reducer(state, action) {
  switch (action.type) {
    case 'added_todo': {
      // 🚩 Mistake: mutating state
      state.todos.push({ id: nextId++, text: action.text });
      return state;
    }
    // ...
  }
}
```



```
function reducer(state, action) {
  switch (action.type) {
    case 'added_todo': {
      // ✅ Correct: replacing with new state
      return {
        ...state,
        todos: [
          ...state.todos,
          { id: nextId++, text: action.text }
        ]
      };
    }
    // ...
  }
}
```