

Managing Complex State with useReducer



Cory House

REACT CONSULTANT AND TRAINER

@housecor reactjsconsulting.com



Agenda



Why useReducer?

- Pure functions
- useState vs useReducer

Convert from useState to useReducer



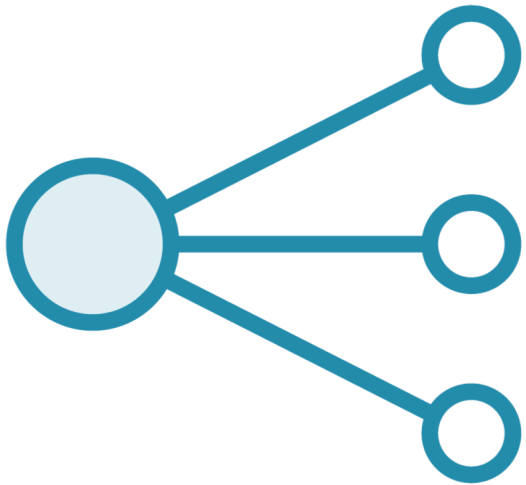
```
const initialState = {count: 0};

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 (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}
```



Why useReducer?



Extract



Reuse



Unit test



Scalability

```
function add(x, y) {  
    return x + y;  
}
```



Always returns same output for a given input



Composable and reusable



Easy to understand and test in isolation

What's a Pure Function?

Depends only on arguments

Doesn't mutate arguments

Has no side-effects

Returns a new value



Demo



Implement useReducer

- Switch from useState to useReducer



useState vs useReducer

Good default → **useState**

Easy to implement for most scenarios

Easy to learn

useReducer ← Switch to as needed

More scalable for complex scenarios:

- Many complex state transitions
- Multiple sub-values
- Next state depends on the previous one

Reason about state in isolation

Testable in isolation

Reusable



Summary



Switched from useState to useReducer

- Handles state as a pure function
- Can unit test
- Can reuse
- Scales well
- Can mix with useState

Consider for complex state

Next up: Context (global data, funcs)

