useState vs useReducer

Choosing the Right State Management Hook

The Question

"When should I use useState vs useReducer? They both manage state, so what's the difference?"

This is one of the most fundamental questions in React. Both hooks manage state, but they solve different problems.

The Simple Answer

useState: For Simple State

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

Use when: Single value, simple updates

useReducer: For Complex State

```
const [state, dispatch] = useReducer(reducer, initialState);
```

Use when: Multiple related values, complex logic

useState: The Basics

What It Is

A simple hook that gives you a state value and a function to update it.

```
const [value, setValue] = useState(initialValue);
```

Simple Example

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

  return (
    <div>
```

```
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <button onClick={() => setCount(count - 1)}>Decrement</button>
      <button onClick={() => setCount(0)}>Reset</button>
    </div>
  );
}
```

Good for:

- ☑ Single piece of state
- ☑ Simple updates (set new value)
- ☑ Independent state variables
- ☑ Quick and straightforward

---

useReducer: The Basics

What It Is

A hook that manages state through a reducer function (like Redux).

```
const [state, dispatch] = useReducer(reducer, initialState);

// Reducer function
function reducer(state, action) {
  switch (action.type) {
    case 'ACTION_TYPE':
      return { ...state, /* changes */ };
    default:
      return state;
  }
}
```

Same Counter with useReducer

```
function 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;
  }
}
```

```
function Counter() {
  const [state, dispatch] = useReducer(reducer, { count: 0 });

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

Good for:

- ☑ Multiple related state values
- ☑ Complex update logic
- ☑ State transitions that depend on previous state
- ☑ Centralized state logic

When Things Get Complex

useState Starts to Struggle

```
// ❌ Managing form with multiple useState - Gets messy!
function AddNutritionEntry() {
  const [mealType, setMealType] = useState('breakfast');
  const [foodItems, setFoodItems] = useState('');
  const [calories, setCalories] = useState('');
  const [protein, setProtein] = useState('');
  const [carbs, setCarbs] = useState('');
  const [fats, setFats] = useState('');
  const [notes, setNotes] = useState('');
  const [isLoading, setIsLoading] = useState(false);
  const [error, setError] = useState(null);

  // 9 different state setters!
  // Hard to reset: setMealType('breakfast'); setFoodItems(''); ...
  // Hard to track: Which ones changed? What's the full state?
}
```

Problems:

- 9 separate `useState` calls
- Resetting form requires 9 function calls
- No single source of truth
- Can't easily log "current state"
- State updates can be out of sync

useReducer Handles It Better

```
// ☑ Managing complex state with useReducer - Clean!
const initialState = {
  mealType: 'breakfast',
  foodItems: '',
  calories: '',
  protein: '',
  carbs: '',
  fats: '',
  notes: '',
  isLoading: false,
  error: null,
};

function formReducer(state, action) {
  switch (action.type) {
    case 'UPDATE_FIELD':
      return { ...state, [action.field]: action.value };
    case 'SET_LOADING':
      return { ...state, isLoading: action.payload };
    case 'SET_ERROR':
      return { ...state, error: action.payload, isLoading: false };
    case 'RESET_FORM':
      return initialState;
    default:
      return state;
  }
}

function AddNutritionEntry() {
  const [state, dispatch] = useReducer(formReducer, initialState);

  // Update any field: dispatch({ type: 'UPDATE_FIELD', field: 'calories', value:
'500' })
  // Reset form: dispatch({ type: 'RESET_FORM' })
  // Single state object, easy to log: console.log(state)
}
```

Benefits:

- One reducer, all state in one place
- Reset entire form with one dispatch
- Easy to log full state for debugging
- All updates go through reducer (predictable)

Real Example from Your App

Your NutritionProvider Uses useReducer

```
// types.ts
export type NutritionState = {
  entries: NutritionEntry[];
  isLoading: boolean;
  error: string | null;
};

export type NutritionAction =
  | { type: "SET_ENTRIES"; payload: NutritionEntry[] }
  | { type: "ADD_ENTRY"; payload: NutritionEntry }
  | { type: "UPDATE_ENTRY"; payload: NutritionEntry }
  | { type: "REMOVE_ENTRY"; payload: string }
  | { type: "SET_LOADING"; payload: boolean }
  | { type: "SET_ERROR"; payload: string | null };

// NutritionProvider.tsx
function nutritionReducer(state: NutritionState, action: NutritionAction):
NutritionState {
  switch (action.type) {
    case "SET_ENTRIES":
      return { ...state, entries: action.payload, error: null };

    case "ADD_ENTRY":
      return {
        ...state,
        entries: [action.payload, ...state.entries],
        error: null
      };

    case "UPDATE_ENTRY":
      return {
        ...state,
        entries: state.entries.map((entry) =>
          entry._id === action.payload._id ? action.payload : entry
        ),
        error: null,
      };

    case "REMOVE_ENTRY":
      return {
        ...state,
        entries: state.entries.filter((entry) => entry._id !== action.payload),
        error: null,
      };

    case "SET_LOADING":
      return { ...state, isLoading: action.payload };

    case "SET_ERROR":
      return { ...state, error: action.payload, isLoading: false };

    default:
      return state;
```

```
  }
}

export default function NutritionProvider({ children, initialEntries }) {
  const [state, dispatch] = useReducer(nutritionReducer, {
    entries: initialEntries ?? [],
    isLoading: false,
    error: null,
  });

  // Now you can dispatch actions:
  // dispatch({ type: "ADD_ENTRY", payload: newEntry });
  // dispatch({ type: "SET_LOADING", payload: true });
}
```

What Would This Look Like with useState?

```
// ❌ Would need multiple useState calls
function NutritionProvider({ children, initialEntries }) {
  const [entries, setEntries] = useState(initialEntries ?? []);
  const [isLoading, setIsLoading] = useState(false);
  const [error, setError] = useState(null);

  // Adding entry:
  const addEntry = (entry) => {
    setEntries([entry, ...entries]); // Update entries
    setError(null); // Clear error
    // Have to remember to update related state!
  };

  // Removing entry:
  const removeEntry = (id) => {
    setEntries(entries.filter(e => e._id !== id)); // Update entries
    setError(null); // Clear error
    // Easy to forget!
  };

  // More error-prone, more to remember
}
```

Side-by-Side Comparison

Simple Counter

useState	useReducer
5 lines of code	15 lines of code
Easier to understand	More boilerplate

useState	useReducer
☑ **Best choice here**	❌ Overkill

Complex Form

useState	useReducer
9+ useState calls	1 useReducer call
Hard to reset all fields	dispatch({ type: 'RESET' })
State can get out of sync	All changes through reducer
❌ Gets messy	☑ **Best choice here**

Your Nutrition State

useState	useReducer
3 useState calls	1 useReducer call
Must remember to clear error	Reducer handles it
Easy to miss related updates	Centralized logic
❌ More error-prone	☑ **Best choice here**

Decision Tree

Use useState When:

```
Is your state...
├─ A single primitive value? (number, string, boolean)
│  └─ ☑ USE useState
│
├─ A simple object or array?
│  └─ Are updates simple? (just set new value)
│     ├─ Yes → ☑ USE useState
│     └─ No → ⚠ CONSIDER useReducer
│
└─ Multiple related values?
   └─ ⚠ CONSIDER useReducer
```

Use useReducer When:

```
Does your state have...
├─ Multiple sub-values? (entries, loading, error)
│  └─ ☑ USE useReducer
│
```

```
├─ Complex update logic?
│  └─ ☑ USE useReducer
│
├─ Next state depends on previous?
│  └─ ☑ USE useReducer
│
└─ Many different ways to update?
   └─ ☑ USE useReducer
```

Common Patterns

Pattern 1: Toggle (useState is fine)

```
const [isOpen, setIsOpen] = useState(false);

<button onClick={() => setIsOpen(!isOpen)}>Toggle</button>
```

Pattern 2: Input Field (useState is fine)

```
const [name, setName] = useState('');

<input value={name} onChange={(e) => setName(e.target.value)} />
```

Pattern 3: Loading State with Data (useReducer is better)

```
// With useState - Can get out of sync!
const [data, setData] = useState(null);
const [isLoading, setIsLoading] = useState(false);
const [error, setError] = useState(null);

// With useReducer - Always in sync!
const [state, dispatch] = useReducer(dataReducer, {
  data: null,
  isLoading: false,
  error: null,
});
```

Pattern 4: List Operations (useReducer is better)

```
// With useState
const [items, setItems] = useState([]);
const addItem = (item) => setItems([...items, item]);
const removeItem = (id) => setItems(items.filter(i => i.id !== id));
```

```
const updateItem = (id, data) => setItems(items.map(i => i.id === id ? data : i));

// With useReducer - More organized!
const [state, dispatch] = useReducer(itemsReducer, { items: [] });
// dispatch({ type: 'ADD', payload: item });
// dispatch({ type: 'REMOVE', payload: id });
// dispatch({ type: 'UPDATE', payload: { id, data } });
```

Advantages Breakdown

useState Advantages

- ☑ **Less code** - Quick and simple
- ☑ **Easy to understand** - No reducer function
- ☑ **Good for beginners** - Straightforward API
- ☑ **Perfect for simple state** - One value, simple updates

useReducer Advantages

- ☑ **Predictable** - All updates through reducer
- ☑ **Testable** - Reducer is pure function
- ☑ **Organized** - Logic in one place
- ☑ **Scalable** - Easy to add new actions
- ☑ **Debuggable** - Can log every action
- ☑ **Complex updates** - Handle multiple state changes together

---

Migration Example

Starting with useState

```
// Simple at first
function TodoList() {
  const [todos, setTodos] = useState([]);

  const addTodo = (text) => {
    setTodos([...todos, { id: Date.now(), text, done: false }]);
  };

  return <div>...</div>;
}
```

Growing Complexity

```
// Getting messy...
function TodoList() {
```

```
  const [todos, setTodos] = useState([]);
  const [filter, setFilter] = useState('all');
  const [isLoading, setIsLoading] = useState(false);
  const [error, setError] = useState(null);

  const addTodo = (text) => {
    setTodos([...todos, { id: Date.now(), text, done: false }]);
  };

  const toggleTodo = (id) => {
    setTodos(todos.map(t => t.id === id ? { ...t, done: !t.done } : t));
  };

  const deleteTodo = (id) => {
    setTodos(todos.filter(t => t.id !== id));
  };

  // Getting complicated!
}
```

Refactored to useReducer

```
// Much cleaner!
function todoReducer(state, action) {
  switch (action.type) {
    case 'ADD_TODO':
      return {
        ...state,
        todos: [...state.todos, { id: Date.now(), text: action.payload, done:
false }],
      };
    case 'TOGGLE_TODO':
      return {
        ...state,
        todos: state.todos.map(t =>
          t.id === action.payload ? { ...t, done: !t.done } : t
        ),
      };
    case 'DELETE_TODO':
      return {
        ...state,
        todos: state.todos.filter(t => t.id !== action.payload),
      };
    case 'SET_FILTER':
      return { ...state, filter: action.payload };
    case 'SET_LOADING':
      return { ...state, isLoading: action.payload };
    case 'SET_ERROR':
      return { ...state, error: action.payload };
    default:
      return state;
```

```
  }
}

function TodoList() {
  const [state, dispatch] = useReducer(todoReducer, {
    todos: [],
    filter: 'all',
    isLoading: false,
    error: null,
  });

  // Clean dispatch calls
  // dispatch({ type: 'ADD_TODO', payload: text });
  // dispatch({ type: 'TOGGLE_TODO', payload: id });
}
```

Testing Perspective

Testing useState

```
// Hard to test - tied to component
test('adds todo', () => {
  const { getByText, getByLabelText } = render(<TodoList />);
  // Have to interact with UI to test state logic
});
```

Testing useReducer

```
// Easy to test - just a function!
test('adds todo', () => {
  const initialState = { todos: [] };
  const action = { type: 'ADD_TODO', payload: 'Buy milk' };
  const newState = todoReducer(initialState, action);

  expect(newState.todos).toHaveLength(1);
  expect(newState.todos[0].text).toBe('Buy milk');
});
```

Reducer is a pure function:

- Same input → same output
- No side effects
- Easy to test without component

The Transition Point

Start with useState

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

Consider useReducer when you find yourself:

- ❌ Having 3+ related useState calls
- ❌ Updating multiple state values together
- ❌ Copying complex update logic between components
- ❌ Having bugs from state getting out of sync
- ❌ Wanting to log state changes for debugging

Real-World Analogies

useState = Light Switch

- On or off
- Simple toggle
- One action, one state change
- Perfect for simple control

useReducer = Control Panel

- Multiple settings
- Complex interactions
- Settings affect each other
- Centralized control
- Perfect for complex systems

The Big Takeaway

> **useState and useReducer aren't competing—they're complementary.**
>
> - **useState** for simple, independent state
> - **useReducer** for complex, related state
>
> **Start simple with useState. Refactor to useReducer when complexity grows.**

Quick Reference

```
// useState - Simple state
const [value, setValue] = useState(initial);
setValue(newValue);

// useReducer - Complex state
```

```
const [state, dispatch] = useReducer(reducer, initialState);
dispatch({ type: 'ACTION', payload: data });

// Reducer function
function reducer(state, action) {
  switch (action.type) {
    case 'ACTION':
      return { ...state, /* changes */ };
    default:
      return state;
  }
}
```

Summary for Your Presentation

Key Points:

1. **useState** = Simple, single values
2. **useReducer** = Complex, multiple related values
3. **Both are valid** - Choose based on complexity
4. **Start simple** - Refactor when needed
5. **useReducer benefits** - Predictable, testable, scalable

Demo Opportunity:

- Show simple form field with useState (appropriate)
- Show NutritionProvider with useReducer (necessary)
- Explain why each choice was made
- Show reducer testing (pure function)

When in Doubt

Ask yourself:

1. **Is this state simple?** → useState
2. **Are these values related?** → useReducer
3. **Is update logic complex?** → useReducer
4. **Will I need to test this logic?** → useReducer
5. **Am I just getting started?** → useState (refactor later if needed)

Remember: You can always refactor from useState to useReducer as your needs grow!

Further Reading

- React: useState
- React: useReducer
- When to use useReducer

- Kent C. Dodds: Should I useState or useReducer?