useReducer Hook

- The useReducer hook is an alternative to the useState hook for managing more complex state logic in React components.
- It provides a way to centralize state updates into a single function (the reducer), making it easier to manage state transitions, especially when the state has multiple sub-values or complex update logic.
- The useReducer hook is one of the fundamental hooks in React, designed for managing state transitions in a structured and predictable way.
- It is inspired by the Redux-like pattern but operates locally within a component, allowing developers to centralize and streamline their state management logic.
- The useReducer hook embodies React's philosophy of component encapsulation and composability, offering a clean, scalable solution to managing complex state transitions.
- It is a powerful tool when used judiciously in applications requiring robust state management.
- The useReducer hook is particularly useful for:
    - Managing state that depends on previous state.
    - Handling multiple state transitions triggered by various actions.
    - Centralizing the state management logic in a single place.

Syntax

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

where,

- reducer: A pure function that takes the current state and an action as arguments and returns a new state.
- initialState: The starting value of the state.
- state: The current state of the component.
- dispatch: A function used to send actions to the reducer.

Core Concepts of useReducer

- **Reducer Function**
    - The reducer function is the core of useReducer. It is a pure function that takes the current state and an action as input and returns a new state.
    - It ensures state updates are predictable and free of side effects.
    - The reducer encapsulates the state transition logic, making it easier to test and maintain.

- **State Management Flow**
  - useReducer divides state management into clear steps:
  - An action is dispatched, representing an event or intent (e.g., incrementing a counter, toggling a checkbox).
  - The reducer function determines how the state should change based on the action.
  - The new state replaces the previous state, triggering a re-render of the component.

- **Actions**
  - Actions are objects that describe "what happened" or "what should be done." Typically, they include:
  - A type property to identify the action.
  - Optional payload data for additional context or parameters.
  - Actions decouple state transitions from specific UI events, improving flexibility and scalability.

- **Initial State**
  - useReducer requires an initial state value, which sets the baseline for state transitions.
  - This initial state can be simple (like a number or string) or complex (like an object or array).

- **Dispatch Function**
  - dispatch is the function provided by useReducer to trigger state updates.
  - It acts as a bridge between the UI and the reducer function, allowing components to request state changes.

When to Use useReducer

- **Complex State Logic:** useReducer is ideal for scenarios where the state is an object or array with multiple fields or where updates involve interdependent values.
- **Multiple State Transitions:** When the same state requires updates in response to various actions, useReducer helps centralize and simplify the logic.
- **Improved Readability:** For components with extensive state logic, separating the logic into a reducer makes the code more readable and maintainable.
- **State-Driven Applications:** Applications with a workflow-like behavior, such as forms, wizards, or state machines, benefit from the structured approach of useReducer.

Working Structure

- Define the initial state.
- Create a reducer function that handles various actions and updates the state accordingly.
- Use the useReducer hook to initialize state and get access to the state and dispatch.

Example: Counter App

```
import React, { useReducer } from 'react';
// Define the initial state
const initialState = { count: 0 };
// Create the reducer function
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:
      throw new Error(`Unknown action: ${action.type}`);
  }}
// Use useReducer in the component
function App() {
  const [state, dispatch] = useReducer(reducer, initialState);
  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>
  );
}
export default App;
```

Example: Todo List Application

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

// Initial State
const initialState = [];

//Reducer Function
function todoReducer(state, action) {
  switch (action.type) {
    case 'add':
      return [...state, { id: Date.now(), text: action.text, completed: false }];
    case 'toggle':
      return state.map(todo =>
        todo.id === action.id ? { ...todo, completed: !todo.completed } : todo
      );
    case 'delete':
      return state.filter(todo => todo.id !== action.id);
    default:
      throw new Error(`Unknown action: ${action.type}`);
  }
}

// Component
function App() {
  const [state, dispatch] = useReducer(todoReducer, initialState);
  const [text, setText] = useState('');

  const handleAddTodo = () => {
    dispatch({ type: 'add', text });
    setText(''); // Clear input field
  };

  return (
    <div>
      <h3>Todo List</h3>
      <input
        type="text"
```

```
            value={text}
            onChange={(e) => setText(e.target.value)}
            placeholder="Add a todo"
          />
          <button onClick={handleAddTodo}>Add Todo</button>

          <ul>
            {state.map(todo => (
              <li key={todo.id}>
                <span
                  style={{
                    textDecoration: todo.completed ? 'line-through' : 'none',
                    cursor: 'pointer',
                  }}
                  onClick={() => dispatch({ type: 'toggle', id: todo.id })}
                >
                  {todo.text}
                </span>
                <button onClick={() => dispatch({ type: 'delete', id: todo.id })}>
                  Delete
                </button>
              </li>
            ))}
          </ul>
        </div>
    );
}

export default App;
```

Differences Between useReducer and useState

useReducer and useState are both React hooks used for managing component state. While they serve similar purposes, they differ significantly in terms of complexity, use cases, and how state updates are managed.

Features	useState	useReducer
Purpose	Designed for managing simple and independent state variables.	Suitable for managing complex state logic, particularly when state updates depend on previous state or involve multiple interrelated variables.
Syntax Simplicity	Very simple and intuitive to use; often used inline within a component.	More verbose as it requires defining a reducer function and dispatching actions.
State Representation	Best for states represented as **primitives** (numbers, strings, booleans) or simple objects.	Ideal for states represented as **complex objects** or arrays, especially when the state structure evolves based on multiple actions.
Update Mechanism	Updates the state directly using a setter function returned by useState.	Updates the state through a **reducer function**, which centralizes all state update logic.
Action Handling	No action type or payload concept; directly update state by calling the setter function with a new value.	Uses **actions** (objects) to describe the type of update and optionally include payload for more structured updates.
Logic Location	State update logic is often embedded within the component or event handlers.	State update logic is centralized in the reducer function, making it easier to maintain and debug.
Scalability	Becomes harder to manage and less readable with increasing state complexity or interrelated state transitions.	Scales better with complexity as all state transitions are handled systematically through the reducer.
Performance	Suitable for smaller components or components with simple state.	May introduce slight overhead for small state, but performs well for larger or more complex state logic.
Code Readability	Cleaner for simple state transitions but can get cluttered when handling multiple states.	Cleaner for complex logic as state transitions are consolidated in the reducer function.

Testing	Harder to isolate state logic for testing since it is typically inline within the component.	Easier to test as the reducer function is a standalone, pure function.
Debugging	Debugging can be straightforward for simple state but harder for interrelated state changes as logic is scattered across the component.	Debugging is simpler for complex state since all state transitions are centralized in the reducer function.
Examples of Use Cases	- Simple counters - Toggling booleans - Form inputs - Component visibility toggles.	- Complex forms or Todo lists - State machines - Workflow applications - Nested or interdependent state updates.
Dispatch Mechanism	No dispatch function; directly updates state using the setter function.	Provides a dispatch function to trigger state transitions by sending actions to the reducer.
Predictability	State transitions are implicit and determined by the setter function.	State transitions are explicit, predictable, and determined by the reducer logic.
Error Handling	Errors in state logic may be harder to identify if state update logic is scattered.	Errors are easier to trace as all logic is centralized in the reducer.
Learning Curve	Easier for beginners due to simplicity and straightforward usage.	Requires a deeper understanding of reducers, actions, and immutability, making it more challenging for beginners.

Advantages of useReducer

- **Predictable State Updates:** Since reducers are pure functions, state transitions are deterministic and easier to debug.
- **Centralized Logic:** All state transitions are consolidated in the reducer, eliminating scattered update logic.
- **Scalability:** useReducer simplifies scaling components with increasing state complexity by organizing logic into discrete action-handling blocks.
- **Testing:** Reducers are isolated from UI logic, making them straightforward to test independently.
- **Decoupled Actions:** The use of actions decouples "what happened" (dispatching actions) from "what to do" (reducer logic), increasing flexibility.

Complex State Management Using useReducer

When managing complex state, useReducer provides a structured approach to handle objects, arrays, nested state updates, and performance optimizations while adhering to immutability best practices.

- **Managing Objects and Arrays in the State:** The useReducer hook is well-suited for handling states represented as objects or arrays. This is especially useful when the state has multiple properties or when dealing with lists of items.

Example: Managing a Form with Multiple Fields

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

// Reducer function
const formReducer = (state, action) => {
 switch (action.type) {
  case 'update':
   return { ...state, [action.field]: action.value };
  case 'reset':
   return action.payload; // Reset to initial state
  default:
   throw new Error(`Unknown action: ${action.type}`);
 }
};

const initialState = {
                        name: '',
                        email: '',
                        password: '',
};

const App = () => {
 const [formState, dispatch] = useReducer(formReducer, initialState);

 return (
  <div>
   <input
```

```
        type="text"
        value={formState.name}
        onChange={(e) =>
          dispatch({ type: 'update', field: 'name', value: e.target.value })
        }
        placeholder="Name"
      />
      <input
        type="email"
        value={formState.email}
        onChange={(e) =>
          dispatch({ type: 'update', field: 'email', value: e.target.value })
        }
        placeholder="Email"
      />
      <input
        type="password"
        value={formState.password}
        onChange={(e) =>
          dispatch({ type: 'update', field: 'password', value: e.target.value })
        }
        placeholder="Password"
      />
      <button onClick={() => dispatch({ type: 'reset', payload: initialState })}>
        Reset
      </button>
      <pre>{JSON.stringify(formState, null, 2)}</pre>
    </div>
  );
};

export default App;
```

- **Nested State Updates:** Handling deeply nested objects or arrays often involves immutability concerns. Use a utility function like immer or manual techniques to update state without mutating it.

Example: Nested Todo List with Subtasks

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

const reducer = (state, action) => {
  switch (action.type) {
    case 'add_task':
      return {
        ...state,
        tasks: [...state.tasks, { id: Date.now(), title: action.payload, subtasks: [] }],
      };
    case 'add_subtask':
      return {
        ...state,
        tasks: state.tasks.map((task) =>
          task.id === action.taskId
            ? {
                ...task,
                subtasks: [...task.subtasks, { id: Date.now(), title: action.payload }],
              }
            : task
        ),
      };
    default:
      throw new Error(`Unknown action: ${action.type}`);
  }
};
const initialState = {  tasks: [],       };

const App = () => {
  const [state, dispatch] = useReducer(reducer, initialState);
  const [taskTitle, setTaskTitle] = useState('');
  const [subTaskTitle, setSubTaskTitle] = useState('');
  const [taskId, setTaskId] = useState(null);
```

```
  return (
    <div>      <input        type="text"        value={taskTitle}
       onChange={(e) => setTaskTitle(e.target.value)}
       placeholder="New Task"
     />
     <button onClick={() => dispatch({ type: 'add_task', payload: taskTitle
})}>
       Add Task
     </button>
     {state.tasks.map((task) => (
       <div key={task.id}>
         <h3>{task.title}</h3>
         <input           type="text"           placeholder="New Subtask"
          onChange={(e) => {
            setSubTaskTitle(e.target.value);
            setTaskId(task.id);
          }}
         />
         <button
          onClick={() =>
            dispatch({ type: 'add_subtask', taskId: taskId, payload: subTaskTitle
})
          }
         >
          Add Subtask
         </button>
         <ul>
          {task.subtasks.map((subtask) => (
            <li key={subtask.id}>{subtask.title}</li>
          ))}
         </ul>
       </div>
     ))}
    </div>
  );
};
export default App;
```

- **Avoiding State Mutation with Immutability Best Practices:** When updating state, always ensure the original state is not modified. This is crucial for React to correctly detect changes and re-render components.

  Techniques to Maintain Immutability

  - **Using the Spread Operator:** Spread the existing state (...state) and make updates to specific fields.
  - **Avoid Direct Modifications:** Never directly modify arrays or objects (e.g., avoid state.push or state[key] = value).
  - **Immutability Libraries:** Libraries like immer or immutable.js simplify immutability management.

- **Optimizing Performance in State Updates:** Performance optimization can be achieved by minimizing unnecessary state updates and avoiding deep copying of unchanged parts of the state.

  Best Practices

  - **Action-Based State Updates:** Only dispatch actions when necessary. Avoid redundant updates.
    Example: Dispatching only when user input is valid.
  - **Memoizing State:** Use React.memo or useMemo to prevent re-renders of unaffected components. (to be discussed soon).
  - **Batching Updates:** React automatically batches updates in event handlers. Avoid splitting updates unnecessarily.
  - **Selector Functions:** Use selectors to compute derived state, reducing unnecessary state complexity.

Handling Side Effects with useReducer

useReducer focuses on state management, but side effects such as API calls, subscriptions, or logging are often handled using useEffect. Combining these hooks provides a clean separation of concerns: useReducer manages the state, and useEffect performs side effects based on state changes or dispatched actions.

Combining useReducer with useEffect: When using useReducer alongside useEffect:

- useReducer handles state transitions based on actions.
- useEffect listens for state changes or specific actions and performs side effects.
- Action types can include those specifically intended to trigger side effects, ensuring a predictable flow.

Example: API Calls in useReducer-Managed Components

```
import React, { useReducer, useEffect } from 'react';

// Define the initial state
const initialState = {
  data: null,
  loading: false,
  error: null,
};

// Define the reducer function
const reducer = (state, action) => {
  switch (action.type) {
    case 'fetch_start':
      return { ...state, loading: true, error: null };
    case 'fetch_success':
      return { ...state, loading: false, data: action.payload, error: null };
    case 'fetch_failure':
      return { ...state, loading: false, error: action.payload };
    default:
      throw new Error(`Unknown action: ${action.type}`);
  }
};
```

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

  // Function to trigger data fetching
  const fetchData = (url) => {
    dispatch({ type: 'fetch_start' });
    fetch(url)
      .then((response) => {
        if (!response.ok) {          throw new Error('Network response is not ok');        }
        return response.json();
      })
      .then((data) => {        dispatch({ type: 'fetch_success', payload: data });      })
      .catch((error) => {       dispatch({ type: 'fetch_failure', payload: error.message });
});
  };
  // Side effect to perform an initial data fetch
  useEffect(() => {    fetchData('https://jsonplaceholder.typicode.com/posts');  }, []);

  return (
    <div>
      <h1>Data Fetching with useReducer and useEffect</h1>
      {state.loading && <p>Loading...</p>}
      {state.error && <p style={{ color: 'red' }}>Error: {state.error}</p>}
      {state.data && (
        <ul>
          {state.data.slice(0, 5).map((item) => (<li key={item.id}>{item.title}</li> ))}
        </ul>
      )}
      <button onClick={() =>
fetchData('https://jsonplaceholder.typicode.com/posts')}>
        Refetch Data
      </button>
    </div>
  );
};

export default App;
```