**4) Hooks: React's Superpowers (Continued)**

**useReducer**

Now we’re talking about useReducer—React’s version of the manager who keeps track of everything happening in a complex office and delegates tasks like a boss. Think of it as Redux-lite but built directly into React.

We’ll build a **Music Playlist Manager** to demonstrate how useReducer manages state transitions like a pro.

**Algorithm/Steps:**

1. Define the **initial state** for your app.
2. Write a **reducer function** to manage state transitions based on actions.
3. Use the useReducer hook to initialize the reducer and state.
4. Dispatch actions to update the state based on events.

**Theory:**

* useReducer is a **state management hook** that is best suited for managing **complex state logic** involving multiple sub-values or actions.
* Think of it like a post office: You send an **action** (like posting a letter), the **reducer** processes it, and the **state** gets updated based on what’s inside the envelope.
* **Why use it?** When useState becomes chaotic, especially for state changes that depend on the previous state.
* **When does it shine?** Multi-step forms, chat application states, quiz navigation, or wizards with tons of actions and transitions.

**Code Example (Real-World Use Case): Music Playlist Manager**

// Example: Music Playlist Manager

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

// 1. Define Initial State

const initialState = [];

// 2. Reducer Function

const playlistReducer = (state, action) => {

  switch (action.type) {

    case 'ADD\_SONG':

      return [...state, { id: Date.now(), name: action.payload, liked: false }];

    case 'TOGGLE\_LIKE':

      return state.map((song) =>

        song.id === action.payload ? { ...song, liked: !song.liked } : song

      );

    case 'REMOVE\_SONG':

      return state.filter((song) => song.id !== action.payload);

    default:

      throw new Error('Unknown action type');

  }

};

const PlaylistManager = () => {

  const [playlist, dispatch] = useReducer(playlistReducer, initialState);

  const [songName, setSongName] = useState('');

  const addSong = () => {

    if (songName.trim()) {

      dispatch({ type: 'ADD\_SONG', payload: songName });

      setSongName('');

    }

  };

  return (

    <div style={{ padding: '20px' }}>

      <h1>Music Playlist Manager</h1>

      <input

        type="text"

        value={songName}

        onChange={(e) => setSongName(e.target.value)}

        placeholder="Add a song"

      />

      <button onClick={addSong}>Add Song</button>

      <ul>

        {playlist.map((song) => (

          <li key={song.id} style={{ margin: '10px 0' }}>

            <span

              style={{

                fontWeight: song.liked ? 'bold' : 'normal',

                cursor: 'pointer',

              }}

              onClick={() => dispatch({ type: 'TOGGLE\_LIKE', payload: song.id })}

            >

              {song.name}

            </span>

            <button

              onClick={() => dispatch({ type: 'REMOVE\_SONG', payload: song.id })}

              style={{ marginLeft: '10px' }}

            >

              Remove

            </button>

          </li>

        ))}

      </ul>

    </div>

  );

};

export default PlaylistManager;

**Explanation for Interviews:**

* **Why useReducer?** When state logic involves multiple steps or complex workflows—think managing chat application states, quiz navigation, or form wizards with validations.
* **Reducer Function?** It’s just a switch-case statement that takes the current state and an action, then spits out the next state.
* **Dispatch?** Think of it as firing an event—you describe what to do, and React does the rest.
* **Immutable State?** Notice how we never mutate the state directly; instead, we create a new array or object.

**Key Points for Interview:**

1. **When to Prefer useReducer Over useState:** Use it for state transitions with multiple actions or dependencies.
2. **Scalability Tip:** Combine it with useContext to create a global state management system without needing Redux.
3. **Debugging Tip:** Log actions and states inside the reducer to trace updates easily.
4. **Optimization Tip:** Use the lazy initialization feature of useReducer for performance optimization.