AeroAspire - SDE Intern
Gokul Krishna S

Week 2 – Day 5 (October 4)

Questions/Reflections :

1. Explain the full data flow: user action → state → props → UI update.

   - User action: When a user interacts (clicks, types, etc.), an event handler (like onClick or onChange) is triggered in a React component.
   - State update: That handler typically calls a state updater like setState or useState's setter (e.g. setValue(newValue)). The internal state changes, and React schedules a re-render.
   - Props update: If the updated state needs to flow to child components, props are passed down from the parent holding state to its children.
   - UI update: Components re-render with new state and/or props, updating the UI automatically to reflect the most current data.
   - This unidirectional data flow—state in parent components, props into children—keeps React apps predictable and easier to debug.

2. What are common anti-patterns in React you noticed or want to avoid?

   - Directly mutating state: State must always be updated immutably using setters (setState or useState). Mutating directly leads to bugs and missed updates.
   - Deeply nested state: Excessively deep or complex state structures are hard to update efficiently and debug. Favor flatter, simpler state objects.
   - Not using keys in lists: Lists without unique key props can cause rendering bugs and poor performance.

- Overusing inline functions: Declaring functions inside render methods creates new function instances every render, hurting performance and causing unnecessary child updates.
- State derived from props or other state: Duplicating state or copying from props often leads to out-of-sync bugs. Derive values as needed instead.
- Heavy prop drilling: Passing props through many layers makes code hard to maintain. Use context or state management for shared/global data.
- Avoiding these patterns keeps code maintainable and performant.

3. If this app grows big, what architectural patterns would you use (e.g. component separation, state management tool)?

- Component separation: Break the UI into focused, reusable components (container/presentational or smart/dumb components). Keep stateful logic in container components, pure UI in presentational components.
- State management tools: For complex or widely shared state, use libraries like Redux, Zustand, or React Context to centralize and control state changes. This enables predictable flows and easier debugging.
- Hooks for logic sharing: Custom hooks promote reusability and separation of business logic from UI code.
- Separation of concerns: Organize code into logical folders by feature, type (UI, hooks, data services), and keep concerns separate.
- Modularization & lazy loading: Split code into modules and use React’s lazy and Suspense for optimal loading as app size grows.
- These patterns make big React apps easier to scale, maintain, test, and optimize for performance.