Series 3: State Management & Context — structured for interview prep with:

- 🔹 **Main Topics Highlighted**
- 💬 **Interview-Style Answers**
- 📌 **Code Examples**
- 🔁 **Real-World Analogies**

🔹 Series 3: State Management & Context

✅ 1. What is "Lifting State Up" in React?

💬 **Interview Answer:**
"Lifting state up" means moving shared state to the nearest common parent of two or more child components that need access to it. This avoids duplicate state and improves consistency.

📌 **Code Example:**

```
function Parent() {
  const [text, setText] = useState('');
  return (
    <>
      <InputBox value={text} onChange={setText} />
      <DisplayBox value={text} />
    </>
  );
}
```

🔁 **Analogy:**
It's like putting shared snacks on a central table so all kids can reach them instead of giving each kid their own.

✅ 2. What is Prop Drilling and Why is it a Problem?

💬 **Interview Answer:**
Prop Drilling occurs when data is passed through many nested components just to reach a deeply nested child. It clutters component trees and makes them harder to manage.

📌 **Code Example:**

```
<Parent>
  <Child>
    <GrandChild data={userData} />
  </Child>
</Parent>
```

🔁 **Analogy:**
Like whispering a message through 4 people — messy and error-prone.

✅ 3. What is the Context API?

💬 **Interview Answer:**
The Context API provides a way to share values like theme, user info, or language across the component tree without manually passing props at every level.

📌 **Code Example:**

```
const ThemeContext = createContext('light');

function App() {
  return (
    <ThemeContext.Provider value="dark">
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function Toolbar() {
  const theme = useContext(ThemeContext);
  return <div className={theme}>Theme is {theme}</div>;
}
```

🔁 **Analogy:**
Context is like a Wi-Fi network — accessible to anyone connected, no need to plug in directly.

✅ 4. When to Use Context API vs Lifting State Up?

💬 **Interview Answer:**
Use **lifting state** when only a few components need to share data.
Use **Context** when many nested components need the same data, or when avoiding prop drilling improves code clarity.

✅ 5. How Does `useContext` Work?

💬 **Interview Answer:**
`useContext` is a hook that lets functional components access values from a React Context. It returns the current context value, based on the nearest `<Provider>` above in the tree.

📌 **Code Example:**

```
const lang = useContext(LanguageContext);
```

🔁 **Analogy:**
Like asking, “What language are we speaking here?” — and you get the answer instantly from your surroundings.

✅ 6. What are the Drawbacks of Context API?

💬 **Interview Answer:**

- Re-renders all components that consume the context when the value changes.
- Not ideal for frequently updating data (like typing text).
- Best for static or low-frequency global data (theme, locale, auth).

✅ 7. What Alternatives Exist for Global State?

💬 **Interview Answer:** For complex state, alternatives like **Redux**, **Zustand**, or **Recoil** offer better scalability, performance, and separation of logic.

Use Case	Suggested Tool
Global theming	Context API
Auth/user info	Context API
App-wide state	Zustand / Redux
Server state	React Query / SWR

✅ 8. What is Zustand (as a lightweight alternative to Redux)?

💬 **Interview Answer:**
Zustand is a small, fast state-management library for React. It avoids boilerplate, supports hooks directly, and works well for shared state.

📌 **Code Example:**

```
import create from 'zustand';

const useStore = create((set) => ({
  count: 0,
  increment: () => set((state) => ({ count: state.count + 1 })),
}));
```

🔁 **Analogy:**
Like using a smart notebook instead of a full spreadsheet app — simple, focused, efficient.

✅ 9. How Do You Prevent Unnecessary Re-renders with Context?

💬 **Interview Answer:**
Split context into smaller pieces, memoize values with `useMemo`, and use selectors (with libraries like Zustand or `use-context-selector`) to isolate updates.

✅ 10. When Should You Avoid Context?

💬 **Interview Answer:**
Avoid using Context for:

- High-frequency updates (like mouse position, form typing)
- Deep trees with large re-render scope
- Situations where performance is critical
