Day 17: Context API – State Management and Sharing Data Across Components

1. Introduction to Context API

React's **Context API** helps manage state globally, without the need to manually pass props down multiple component levels (known as "prop drilling"). It's ideal when multiple components need access to the same data.

Example: Imagine you have a logged-in user's data (like name, email) that you want to use in the Navbar, Dashboard, and Footer. Passing this through every component would be messy — Context API solves this problem.

---

2. Why We Use Context API

- To **avoid prop drilling** (passing props through multiple levels).
- To **share state globally** across unrelated components.
- To **manage app-wide data** such as theme, authentication, or language settings.

---

3. Core Components of Context API

1. `createContext()` – Creates a context object.
2. `Provider` – Makes the data available to child components.
3. `useContext()` – Used by child components to consume data.

---

4. Example: Creating and Using Context

Let's create a simple **UserContext** to share user info across components.

```
import React, { createContext, useContext, useState } from 'react';

// Step 1: Create Context
const UserContext = createContext();

// Step 2: Create a Provider Component
function UserProvider({ children }) {
  const [user, setUser] = useState({ name: 'Rehan Abbasi', role: 'Software
Engineer' });
  return (
    <UserContext.Provider value={{ user, setUser }}>
      {children}
    </UserContext.Provider>
  );
}

// Step 3: Create a component that consumes the Context
function Navbar() {
```

```
  const { user } = useContext(UserContext); // access data from context
  return <h2>Welcome, {user.name}! 👋</h2>;
}

// Step 4: Another component to update data
function UpdateUser() {
  const { setUser } = useContext(UserContext);
  return (
    <button onClick={() => setUser({ name: 'Boss', role: 'Full Stack Dev' })}
>
      Update User Info
    </button>
  );
}

// Step 5: Combine everything
function App() {
  return (
    <UserProvider>
      <Navbar />
      <UpdateUser />
    </UserProvider>
  );
}

export default App;
```

---

5. How It Works

- `UserContext` is created with `createContext()`.
- `UserProvider` wraps all components that need access to user data.
- `Navbar` and `UpdateUser` consume the data using `useContext(UserContext)`.
- When you click the **Update User Info** button, both components using the context automatically update.

---

6. When to Use Context API

👈🏻Global state like **theme (dark/light)**, **authentication**, or **language**. 👈🏻Sharing user data or settings across multiple pages. 👈Avoid using for small local state (use `useState` instead).

---

7. Benefits

- Cleaner and scalable code.
- Avoids prop drilling.
- Better structure for global data.

---

8. Exercise – Practice Task

👉**Task:** Create a small app using Context API that manages theme mode.

Requirements: 1. Create a `ThemeContext`. 2. Store a value for the theme (e.g., `light` or `dark`). 3. Create a toggle button to switch themes. 4. Use the theme value to change the background color of the app.

Bonus: Save the selected theme in local storage so it persists after reload.

👈**End of Day 17 – You have learned Context API for global state management!**