Week 17: DocAppoint - Advanced System Design & Database Schema

Day 1: Requirement Specification for Healthcare Platforms

- **Problem Statement**: The project addresses the inefficiency of manual appointment booking by providing a real-time, 24/7 digital interface.
- **User Personas**: The system is architected to support three distinct roles: Admins (clinic management), Doctors (schedule management), and Patients (booking and history).
- **Functional Requirements**: Core features include doctor profile customization, specialization-based searching, time-slot availability logic, and booking status tracking.
- **Non-Functional Requirements**: Prioritizing data privacy, horizontal scalability via MongoDB, and high-performance responses using the Node.js event-driven model.
- **Technology Stack**: Utilizing the MERN (MongoDB, Express, React, Node) stack to ensure a unified JavaScript development environment from frontend to backend.

Day 2: NoSQL Data Modeling and Relationship Design

- **Schema Flexibility**: Leveraging MongoDB’s dynamic schemas to store diverse medical specializations without rigid table structures.
- **Doctors Collection**: Documents include specialization, experience, fees, and an array of availableSlots containing unique timestamps.
- **Users Collection**: Stores identity data including hashed passwords, emails, and roles (Doctor/Patient) to control system access.
- **Appointments Collection**: A mapping document linking patientID to doctorID, containing fields for appointmentDate, timeSlot, and status (Pending/Confirmed/Cancelled).
- **Data Consistency**: Implementing logic to ensure that once a slot is booked, it is marked as unavailable in the doctor's document to prevent conflicts.

Day 3: Application Architecture and State Hierarchy

- **State Types (Appendix B)**: Categorizing data into "Session State" for user logins and "Communication State" for API request status (loading/success/error).
- **One-Way Data Flow**: Following React's unidirectional data flow, passing doctor data from parent containers to child profile cards via props.
- **Virtual DOM Optimization**: Ensuring that only specific time-slots re-render when a user selects them, maintaining high UI performance.
- **Control State**: Managing the visibility of modals for appointment confirmation and the active state of filter buttons (e.g., filtering by "Cardiologist").
- **Lifting State Up**: Moving the search query state to a common ancestor to allow the search bar to filter the doctor list displayed in the gallery.

Day 4: Backend Infrastructure with Node.js and Express

- **Server Setup**: Initializing the Node.js environment and installing the Express framework to handle healthcare-related API requests.
- **Non-Blocking I/O**: Utilizing Node's event-driven architecture to handle multiple patient bookings simultaneously without server lag.
- **Modular Routing**: Setting up dedicated route files for /api/doctors, /api/users, and /api/appointments to maintain a clean codebase.
- **Middleware Integration**: Implementing express.json() for parsing patient data and cors() for frontend-backend communication.
- **Environment Variables**: Securing the MongoDB URI and private keys in .env files to follow industry security standards.

Day 5: Frontend Scaffolding and Routing (React Router)

- **Project Initialization**: Scaffolding the React application with a structure optimized for large-scale projects like DocAppoint.
- **Single Page Application (SPA)**: Implementing react-router to allow patients to navigate from the home page to booking without refreshing the browser.

- **Route Parameters**: Using dynamic paths like /doctor/:id to fetch and display the specific profile of a doctor based on the URL.
- **Navigation Components**: Creating a persistent NavBar using <NavLink> to highlight the active section (e.g., "My Appointments").
- **Private Routes**: Developing a wrapper to redirect unauthenticated users to the login page if they attempt to access the appointment booking screen.