### **Introduction**

**Project Title:** Book-A-Doctor-Using-MERN

**Team Members:**

The project was developed by a dedicated team of five members, each contributing their expertise to ensure its successful completion. Below is the list of team members and their respective roles:

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Register Number** | **Naan Mudhalvan ID** | **Role** |
| Udhaya Kumar A (Team Leader) | 412721104055 | 07C25EBEE884965A2DC44C962D801574 | Full-Stack Developer & Coordinator |
| Saviour V | 412721104043 | F12B1328970305A802B4161C3354151F | Backend Developer |
| Gokul K | 412721104014 | 45F858E0100F234B3EA4C47E5996D407 | Frontend Developer |
| Gokul J | 412721104013 | 27BFF489E0B9F12829243F2D871BE1E3 | Database Administrator |
| Ajay Y | 412721104003 | 531E5FF050FAEB29018458169E59F0A9 | UI/UX Designer |

**Project Overview:**

The *Book-A-Doctor-Using-MERN* project is a web-based platform aimed at simplifying the process of connecting patients with healthcare professionals. Built using the MERN stack (MongoDB, Express.js, React.js, and Node.js), the system ensures a seamless user experience for patients, doctors, and administrators. It facilitates appointment booking, doctor profile management, and efficient data handling while maintaining a secure and user-friendly interface.

### **Project Overview**

**Purpose:**

The *Book-A-Doctor-Using-MERN* project was developed to streamline and enhance the process of connecting patients with healthcare professionals. Its primary goal is to provide a centralized platform for managing doctor appointments, reducing the hassle of manual scheduling and ensuring a smoother healthcare experience. The system serves three distinct user roles—patients, doctors, and administrators—each with tailored functionalities to address their unique needs. This project leverages modern web development technologies to ensure scalability, maintainability, and an intuitive user experience.

**Goals:**

* Simplify the appointment booking process for patients.
* Provide doctors with an efficient way to manage their appointments and profiles.
* Enable administrators to oversee platform operations, including user and appointment management.
* Ensure secure data handling and authentication for all users.
* Create a modular, scalable application architecture using the MERN stack.

**Features:**

The application incorporates the following key features:

1. **User Registration and Login:**
   1. Secure registration and login for patients, doctors, and administrators.
   2. JWT-based authentication ensures secure access to the platform.
2. **Role-Based Dashboards:**
   1. **Patients:**
      1. View and search for doctors by specialization, location, or availability.
      2. Book appointments with doctors and track appointment statuses.
   2. **Doctors:**
      1. Update their profiles, including specialization, experience, working hours, and fees.
      2. View and manage their appointment schedules.
   3. **Administrators:**
      1. Manage the platform by overseeing user accounts and verifying doctor registrations.
      2. Monitor and update appointment statuses.
3. **Appointment Management:**
   1. Seamless booking and scheduling of appointments between patients and doctors.
   2. Status tracking for appointments (e.g., pending, confirmed, completed, canceled).
4. **Notifications System:**
   1. Real-time notifications for appointment confirmations, cancellations, and updates.
5. **Data Management:**
   1. **Patients:** Secure storage of user information, including notifications and appointment history.
   2. **Doctors:** Maintain detailed doctor profiles, including specializations and availability.
   3. **Appointments:** Centralized database for managing all appointment-related data.
6. **Document Uploads:**
   1. Support for patients to upload relevant documents when booking an appointment.
7. **Responsive User Interface:**
   1. A visually appealing, mobile-friendly UI for seamless interaction across devices.
8. **Secure Data Handling:**
   1. Implementation of bcrypt for password hashing.
   2. Data validation and error handling for robust backend operations.

This comprehensive system is designed to improve accessibility, efficiency, and overall satisfaction for all users involved in healthcare management.

### **Architecture**

**Frontend:**

The frontend of the *Book-A-Doctor-Using-MERN* project is built using **React**, focusing on a component-based architecture to ensure modularity, maintainability, and reusability.

* **Component Structure:**
* The application is divided into role-specific and common components:
  + **Common Components:** Shared across all users (e.g., Login.jsx, Register.jsx, Home.jsx).
  + **User-Specific Components:**
    - **Admin Components:** Handle tasks like managing users, doctors, and appointments (AdminUsers.jsx, AdminDoctors.jsx, AdminAppointments.jsx).
    - **Doctor Components:** Allow doctors to manage appointments and profiles (DoctorAppointments.jsx).
    - **User Components:** Enable patients to search for doctors, book appointments, and view schedules (DoctorList.jsx, UserAppointments.jsx).
* **State Management:**

Utilizes React's **Context API** or libraries like Redux to manage the application's state, ensuring efficient data flow between components.

* **Routing:**
  + React Router is used to handle navigation between pages and ensure role-based access control.
  + Example routes: /login, /admin/dashboard, /user/appointments, /doctor/profile.
* **Styling:**
  + CSS modules and React-specific styling libraries are used for a responsive and visually appealing interface (App.css).

**Backend:**

The backend is developed using **Node.js** and **Express.js**, adhering to a RESTful architecture for managing API endpoints.

* **Server Setup:**
  + The index.js file serves as the entry point, initializing the Express server, connecting to the MongoDB database, and defining routes for different user roles.
  + Middleware such as authMiddleware.js ensures secure access to protected routes.
* **Modular Design:**
  + **Routes:** Organized into separate files for user, doctor, and admin functionalities (userRoutes.js, doctorRoutes.js, adminRoutes.js).
  + **Controllers:** Encapsulate the logic for handling requests, promoting a clean separation of concerns (userC.js, doctorC.js, adminC.js).
  + **Models:** Use Mongoose to define schemas for MongoDB collections, ensuring structured and validated data storage.
* **Security:**
  + JWT-based authentication to validate and authorize user requests.
  + Password hashing using bcrypt for secure credential storage.

**Database:**

The database layer uses **MongoDB**, with Mongoose as the ODM (Object-Document Mapping) library.

* **Schemas:**
  + **Users Collection (userModel.js):**

{   
 fullName: String,   
 email: String,   
 password: String,   
 phone: String,   
 type: { type: String, enum: ['user', 'admin'], default: 'user' },   
 notifications: [String],   
 isDoctor: { type: Boolean, default: false }   
}

* + **Doctors Collection (docModel.js):**

{   
 userId: mongoose.Schema.Types.ObjectId,   
 fullName: String,   
 email: String,   
 phone: String,   
 address: String,   
 specialization: String,   
 experience: Number,   
 fees: Number,   
 status: { type: String, enum: ['active', 'inactive'], default: 'inactive' },   
 workingHours: String   
}

* + **Appointments Collection (appointmentModel.js):**

{   
 userId: mongoose.Schema.Types.ObjectId,   
 doctorId: mongoose.Schema.Types.ObjectId,   
 userInfo: Object,   
 doctorInfo: Object,   
 appointmentDate: Date,   
 document: String,   
 status: { type: String, enum: ['pending', 'confirmed', 'completed', 'canceled'], default: 'pending' }   
}

* **Interactions:**
  + **Data Retrieval:** Efficient querying and filtering based on roles (e.g., finding available doctors, fetching user appointments).
  + **Data Relationships:** Collections are linked using ObjectId references for relational data, ensuring logical connections between users, doctors, and appointments.

This layered architecture ensures a clear separation of concerns, promoting scalability, maintainability, and secure handling of user data.

### **Setup Instructions**

#### **Prerequisites**

Before setting up the project, ensure the following dependencies are installed on your system:

1. **Node.js** (v14.0.0 or above)
   1. [Download Node.js](https://nodejs.org/)
   2. Verify installation:

node -v   
npm -v

1. **MongoDB** (v4.4 or above)
   1. [Download MongoDB](https://www.mongodb.com/try/download/community)
   2. Verify installation:

mongo --version

1. **Git**
   1. [Download Git](https://git-scm.com/downloads)
   2. Verify installation:

git --version

1. **Code Editor** (Recommended: Visual Studio Code)
   1. [Download VS Code](https://code.visualstudio.com/)

#### **Installation**

Follow these steps to set up the project locally:

1. **Clone the Repository**  
   Open a terminal and clone the project repository:

git clone <https://github.com/your-username/book-a-doctor-using-mern.git>   
cd book-a-doctor-using-mern

1. **Backend Setup**  
   Navigate to the backend folder:

cd backend

* 1. **Install Dependencies:**

npm install

* 1. **Environment Variables:**  
     Create a .env file in the backend directory and add the following variables:

PORT=5000   
MONGO\_URI=mongodb://localhost:27017/bookADoctorDB   
JWT\_SECRET=your\_jwt\_secret\_key

* 1. **Start the Backend Server:**

npm start

The backend server will run on <http://localhost:5000>.

1. **Frontend Setup**  
   Navigate to the frontend folder:

cd ../frontend

* 1. **Install Dependencies:**

npm install

* 1. **Environment Variables:**  
     If necessary, create a .env file in the frontend directory. Example:

REACT\_APP\_API\_URL=http://localhost:5000/api

* 1. **Start the Frontend Server:**

npm start

The frontend application will run on <http://localhost:3000>.

1. **Run MongoDB:**  
   Ensure the MongoDB server is running locally or on a hosted service:
2. **Database Initialization (Optional):**  
   Seed the database with initial data (if available):
   1. Add sample data from backend/users.json or other scripts provided in the repository.
   2. Run:

#### **Testing the Application**

1. Open a browser and navigate to the frontend: <http://localhost:3000>.
2. Perform API tests or manual testing by registering users, adding doctors, and booking appointments.

### **Folder Structure**

#### **Client (Frontend)**

The frontend is organized into folders and files to ensure modularity, scalability, and maintainability. Built using **React**, it follows a component-based architecture.

frontend/

├── public/

│ ├── index.html # Main HTML file to mount the React app

│ └── manifest.json # Web application metadata

├── src/

│ ├── components/

│ │ ├── admin/ # Admin-specific components

│ │ │ ├── AdminAppointments.jsx

│ │ │ ├── AdminDoctors.jsx

│ │ │ └── AdminUsers.jsx

│ │ ├── common/ # Common components used across the app

│ │ │ ├── Home.jsx

│ │ │ ├── Login.jsx

│ │ │ └── Register.jsx

│ │ ├── user/ # User-specific components

│ │ │ ├── DoctorAppointments.jsx

│ │ │ ├── DoctorList.jsx

│ │ │ ├── UserAppointments.jsx

│ │ │ └── UserHome.jsx

│ │ └── App.js # Root component that defines routing and app-wide layout

│ ├── App.css # Global styles for the app

│ ├── index.js # Entry point of the React app

│ └── package.json # Project metadata and dependencies

**Description of Key Folders:**

* **public/**: Contains the index.html file, which serves as the entry point for the app.
* **src/**: Main application source folder containing React components and the entry point index.js.
  + **components/**: Divided into subdirectories for admin, user, and common functionalities. This modular approach keeps user roles and shared functionalities separate.
  + **App.js**: Implements routing to navigate between pages such as login, home, and role-based dashboards.

#### **Server (Backend)**

The backend is structured to handle API requests, manage database interactions, and serve static files. Built with **Node.js** and **Express.js**, it follows a modular approach.

backend/

├── config/

│ └── connectToDB.js # MongoDB connection logic

├── controllers/

│ ├── adminC.js # Controller for admin operations

│ ├── doctorC.js # Controller for doctor operations

│ └── userC.js # Controller for user operations

├── middlewares/

│ └── authMiddleware.js # Middleware for JWT-based authentication

├── routes/

│ ├── adminRoutes.js # Routes for admin API endpoints

│ ├── doctorRoutes.js # Routes for doctor API endpoints

│ └── userRoutes.js # Routes for user API endpoints

├── schemas/

│ ├── appointmentModel.js # Mongoose schema for appointments

│ ├── docModel.js # Mongoose schema for doctors

│ └── userModel.js # Mongoose schema for users

├── uploads/ # Directory for uploaded files

├── .gitignore # Git ignore file

├── hashPasswords.js # Script to hash passwords using bcrypt

├── index.js # Entry point of the backend application

├── package.json # Project metadata and dependencies

└── users.json # Example user data for seeding the database

**Description of Key Folders:**

* **config/**: Contains the logic to establish a MongoDB connection.
* **controllers/**: Business logic for handling requests. Controllers are role-specific: adminC.js (Admin), doctorC.js (Doctors), and userC.js (Users).
* **middlewares/**: Contains reusable middleware, such as authMiddleware.js for JWT authentication.
* **routes/**: Defines the RESTful API routes for user, doctor, and admin. Each file maps HTTP methods to controller functions.
* **schemas/**: Mongoose schemas for defining and managing database models (Users, Doctors, and Appointments).
* **uploads/**: A directory for storing uploaded files (e.g., documents for appointments).

**Main Entry Point:**

* **index.js**: Initializes the Express server, connects to MongoDB, configures middlewares, and sets up API routes.

### **Running the Application**

To run the application locally, follow these steps to start both the **frontend** and **backend** servers:

#### **1. Start the Frontend Server**

Navigate to the client directory and use the following commands:

cd frontend   
npm install # Installs all dependencies for the React application  
npm start # Starts the React development server

Once the frontend server starts, it will usually run on <http://localhost:3000> (default port). If this port is occupied, React will prompt you to use another port.

#### **2. Start the Backend Server**

Navigate to the server directory and use the following commands:

cd backend   
npm install # Installs all dependencies for the Node.js application  
npm start # Starts the Express backend server

The backend server typically runs on <http://localhost:5000> (default port). You can customize this port by editing the environment variables or index.js file.

#### **Concurrent Execution**

To simplify the process of starting both servers, you can use tools like concurrently. Add it as a dependency to your project using:

npm install -g concurrently

Then, execute the following command from the root project directory to start both servers simultaneously:

concurrently "npm start --prefix frontend" "npm start --prefix backend"

### **API Documentation**

The **backend API** provides RESTful endpoints for managing users, doctors, and appointments. Below is a detailed list of the available endpoints, categorized by their functionalities.

### **User Endpoints**

#### **1. Register User**

* **Endpoint**: /api/user/register
* **Method**: POST
* **Description**: Registers a new user.
* **Request Body**:

{  
 "fullName": "John Doe",  
 "email": "[johndoe@example.com](mailto:johndoe@example.com)",  
 "password": "securepassword",  
 "phone": "1234567890"  
}

* **Response**:

{  
 "success": true,  
 "message": "User registered successfully."  
}

#### **2. Login User**

* **Endpoint**: /api/user/login
* **Method**: POST
* **Description**: Authenticates a user and provides a JWT token.
* **Request Body**:

{  
 "email": "[johndoe@example.com](mailto:johndoe@example.com)",  
 "password": "securepassword"  
}

* **Response**:

{  
 "success": true,  
 "token": "jwt-token-here",  
 "user": {  
 "id": "user-id",  
 "fullName": "John Doe",  
 "email": "[johndoe@example.com](mailto:johndoe@example.com)",  
 "type": "user",  
 "isDoctor": false  
 }  
}

#### **3. Get User Notifications**

* **Endpoint**: /api/user/notifications
* **Method**: GET
* **Headers**:
  + Authorization: Bearer <jwt-token>
* **Description**: Fetches notifications for the authenticated user.
* **Response**:

{  
 "success": true,  
 "notifications": [  
 {  
 "message": "Your appointment has been confirmed.",  
 "date": "2024-11-18T10:00:00Z"  
 }  
 ]  
}

### **Doctor Endpoints**

#### **1. Register as Doctor**

* **Endpoint**: /api/doctor/register
* **Method**: POST
* **Description**: Registers a user as a doctor.
* **Request Body**:

{  
 "userId": "user-id",  
 "fullName": "Dr. Jane Smith",  
 "email": "[drsmith@example.com](mailto:drsmith@example.com)",  
 "phone": "9876543210",  
 "specialization": "Cardiology",  
 "experience": 10,  
 "fees": 500,  
 "workingHours": "9 AM - 5 PM",  
 "address": "123 Clinic Street"  
}

* **Response**:

{  
 "success": true,  
 "message": "Doctor profile created successfully."  
}

#### **2. Get Doctor Appointments**

* **Endpoint**: /api/doctor/appointments
* **Method**: GET
* **Headers**:
  + Authorization: Bearer <jwt-token>
* **Description**: Fetches appointments for the authenticated doctor.
* **Response**:

{  
 "success": true,  
 "appointments": [  
 {  
 "id": "appointment-id",  
 "userInfo": {  
 "fullName": "John Doe",  
 "email": "[johndoe@example.com](mailto:johndoe@example.com)"  
 },  
 "appointmentDate": "2024-11-20T10:00:00Z",  
 "status": "Confirmed"  
 }  
 ]  
}

### **Admin Endpoints**

#### **1. Get All Users**

* **Endpoint**: /api/admin/users
* **Method**: GET
* **Headers**:
  + Authorization: Bearer <jwt-token>
* **Description**: Fetches all registered users.
* **Response**:

{  
 "success": true,  
 "users": [  
 {  
 "id": "user-id",  
 "fullName": "John Doe",  
 "email": "[johndoe@example.com](mailto:johndoe@example.com)",  
 "type": "user",  
 "isDoctor": false  
 }  
 ]  
}

#### **2. Approve Doctor**

* **Endpoint**: /api/admin/approve-doctor/:doctorId
* **Method**: PATCH
* **Headers**:
  + Authorization: Bearer <jwt-token>
* **Description**: Approves a doctor profile.
* **URL Parameters**:
  + doctorId (string): ID of the doctor to be approved.
* **Response**:

{  
 "success": true,  
 "message": "Doctor approved successfully."  
}

### **Appointment Endpoints**

#### **1. Create Appointment**

* **Endpoint**: /api/appointment/create
* **Method**: POST
* **Headers**:
  + Authorization: Bearer <jwt-token>
* **Description**: Creates a new appointment.
* **Request Body**:

{  
 "userId": "user-id",  
 "doctorId": "doctor-id",  
 "appointmentDate": "2024-11-20T10:00:00Z",  
 "document": "document-url"  
}

* **Response**:

{  
 "success": true,  
 "message": "Appointment created successfully."  
}

#### **2. Update Appointment Status**

* **Endpoint**: /api/appointment/status/:appointmentId
* **Method**: PATCH
* **Headers**:
  + Authorization: Bearer <jwt-token>
* **Description**: Updates the status of an appointment.
* **URL Parameters**:
  + appointmentId (string): ID of the appointment.
* **Request Body**:

{  
 "status": "Cancelled"  
}

* **Response**:

{  
 "success": true,  
 "message": "Appointment status updated successfully."  
}

### **Authentication**

#### **Authentication and Authorization Overview**

The project employs **JWT (JSON Web Token)** for secure authentication and authorization. This method ensures that users, doctors, and admins access only their respective resources and functionalities.

#### **Authentication Process**

1. **User Registration**:
   1. A new user registers with their details. Passwords are hashed using **bcrypt** before storage.
2. **User Login**:
   1. Upon successful login, a JWT is issued containing user-specific details like user ID, role, and authentication status.
3. **Token Storage**:
   1. The JWT is stored on the client side in **localStorage** or **httpOnly cookies** (depending on implementation preferences).

#### **Authorization Process**

1. **Protected Routes**:
   1. Specific API endpoints are protected using middleware that verifies the JWT token.
2. **Middleware Functionality**:
   1. The middleware checks the Authorization header for the token.
   2. If the token is valid, the request proceeds.
   3. If invalid or missing, the request is rejected with an **unauthorized** error.
3. **Role-Based Access Control (RBAC)**:
   1. Depending on the user role (e.g., admin, doctor, regular user), the application determines which resources and actions are accessible.

#### **Token Example**

JWT payload typically includes:

{  
 "id": "user-id",  
 "email": "[user@example.com](mailto:user@example.com)",  
 "role": "admin",  
 "iat": 1696585571,  
 "exp": 1696595571  
}

### **User Interface**

Here’s an overview of the UI components and features, complemented by screenshots:

#### **Login and Registration Pages**

* **Features**:
  + Form validation for user inputs.
  + Error messages for invalid credentials.

#### **Dashboard Views**

1. **User Dashboard**:
   1. Shows appointments, notifications, and profile settings.
   2. Allows booking new appointments.
2. **Doctor Dashboard**:
   1. Displays a schedule of appointments.
   2. Provides options to accept/reject appointments.
3. **Admin Dashboard**:
   1. Includes user and doctor management panels.
   2. Tools for approving doctor registrations.

### **Testing**

#### **Testing Strategy**

The project incorporates **manual testing** and **automated testing** for functionality, security, and performance.

#### **Testing Tools**

1. **Postman**:
   1. Used to test API endpoints for correct responses and error handling.
2. **Jest**:
   1. For unit testing of backend functionality, such as controllers and middleware.
3. **Cypress**:
   1. For end-to-end testing of the React frontend.

#### **Test Scenarios**

1. **API Testing**:
   1. Test all endpoints with valid and invalid inputs.
   2. Ensure proper status codes (200, 401, 403, 404) are returned.
2. **Frontend Testing**:
   1. Validate form inputs and field-level validation.
   2. Test UI components for rendering correctness.
3. **Integration Testing**:
   1. Check interaction between frontend and backend, such as booking an appointment or updating status.
4. **Role-Specific Tests**:
   1. Ensure users, doctors, and admins have access only to their functionalities.

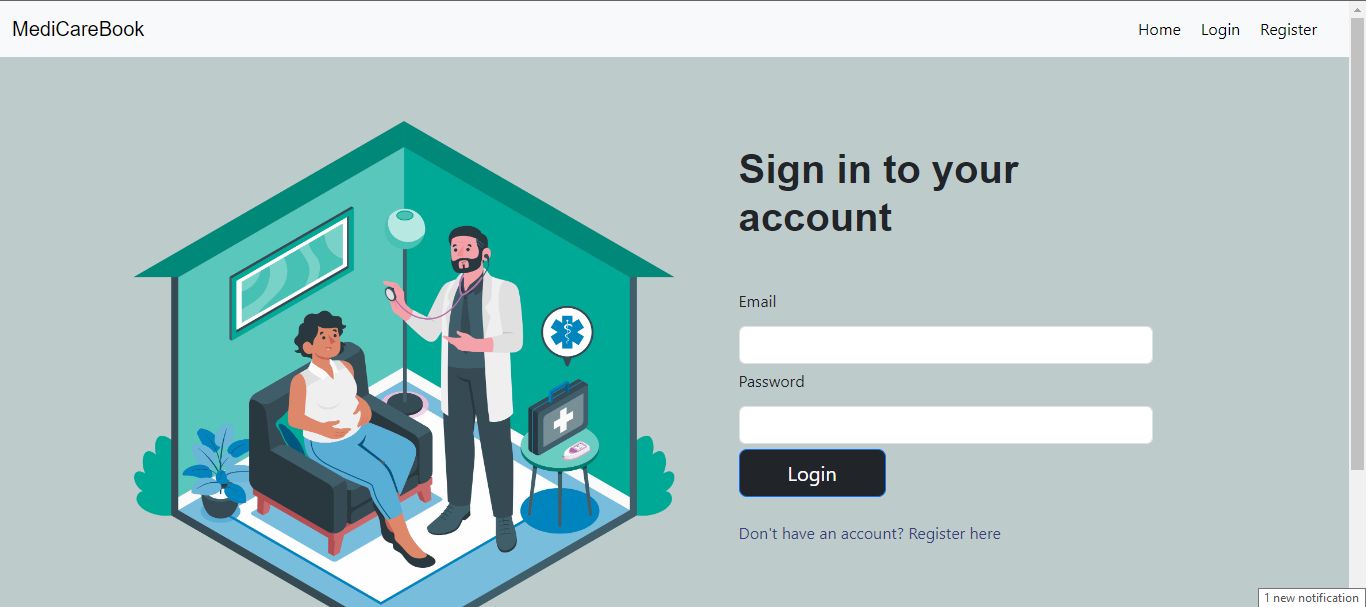
#### **Example Test Case**

|  |  |  |  |
| --- | --- | --- | --- |
| **Scenario** | **Input** | **Expected Output** | **Result** |
| User Registration | Valid email, password | "User registered successfully" | Pass |
| Invalid Login | Incorrect credentials | "Invalid email or password" | Pass |
| Book Appointment (Unauthorized) | No token in request header | "Unauthorized access" | Pass |

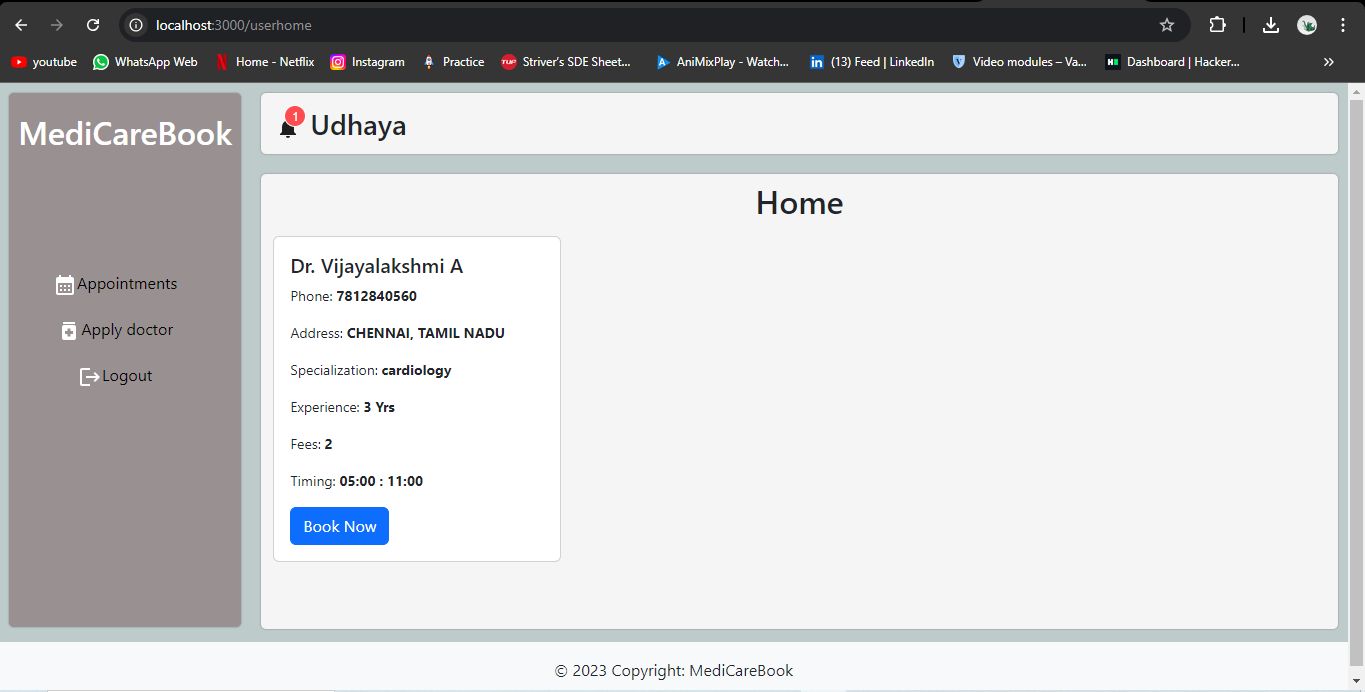
### **Screenshots or Demo**

#### **Screenshots**

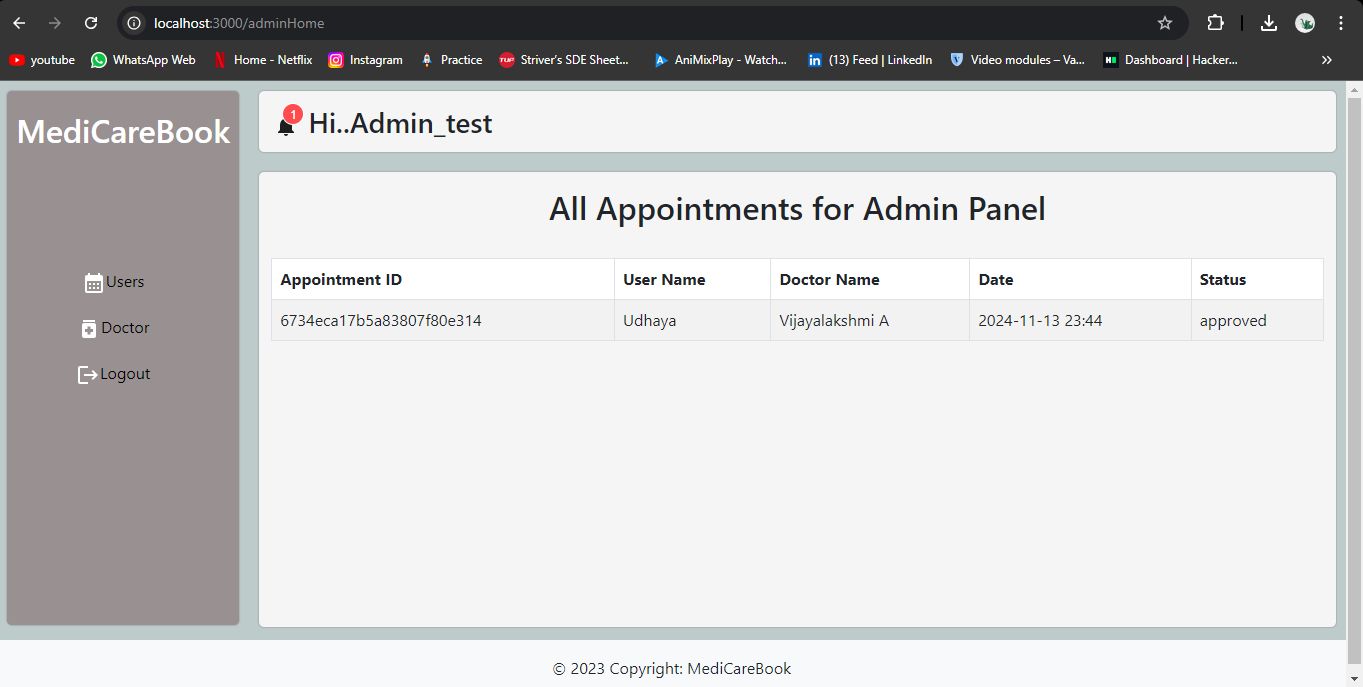
1. **Login Page**:



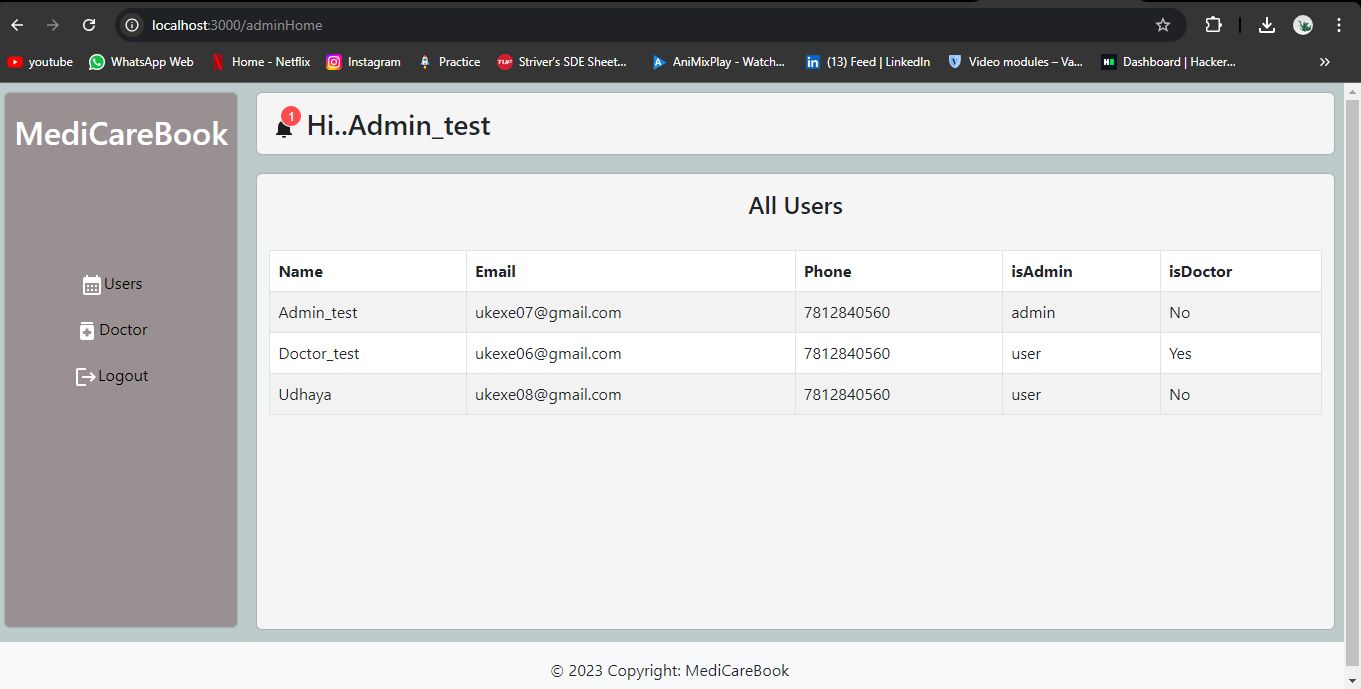
1. **User Dashboard**:



1. **Doctor Management Panel**:



1. **Admin Dashboard**:



#### **Demo Link**

[Click here to view the live demo](https://drive.google.com/file/d/1FZ9XmVKr2UPNkrgzPca7KITvg1bRecT8/view?usp=sharing)

### **Known Issues**

#### **Current Bugs**

1. **Token Expiration Handling**:
   1. Issue: The application doesn’t notify the user when the JWT token expires.
   2. Temporary Fix: Users must log in again manually.
2. **Doctor Availability Updates**:
   1. Issue: Changes in working hours may not immediately reflect for users due to cache delays.
   2. Workaround: Refresh the page to retrieve updated data.
3. **Cross-Browser Compatibility**:
   1. Issue: Minor layout inconsistencies on older versions of Safari.
   2. Planned Fix: Update CSS to ensure broader compatibility.

### **Future Enhancements**

#### **Short-Term Improvements**

1. **Improved Notifications**:
   1. Real-time push notifications using **Socket.IO** for appointment updates.
2. **User Profile Customization**:
   1. Add features like uploading profile pictures and updating personal information.
3. **Localization Support**:
   1. Multi-language support for better accessibility.

#### **Long-Term Goals**

1. **AI-Powered Doctor Recommendations**:
   1. Integrate a recommendation engine to suggest doctors based on user history and preferences.
2. **Analytics Dashboard**:
   1. Provide insights for doctors and admins, such as appointment trends and user demographics.
3. **Mobile Application**:
   1. Develop native Android and iOS applications for broader accessibility.
4. **Integration with Healthcare APIs**:
   1. Connect with external healthcare systems to sync user medical records and appointment histories.