**BOOK A DOCTOR USING MERN**

**1.Introduction**

**Team Members:**

|  |  |
| --- | --- |
| **Team Member** | **Role** |
| Vishal Aakash H | Lead (Oversees and contributes to all MERN stack development) |
| Shree Badrinath S | Developed Frontend |
| R. Santhosh | Designed User Interface |
| Saravana Kumar E | Developed Backend |
| Praveen Kumar V | Test the Interface |

## **//11&9 need to attach**

**2. Project Overview**

**Purpose**

The purpose of the Book A Doctor platform is to simplify and streamline the process of booking medical appointments by connecting patients with verified doctors. Built using the MERN stack (MongoDB, Express.js, React.js, and Node.js), the project ensures both genuine doctor listings and efficient appointment management. By having an admin approval system for doctor applications, the platform prioritizes user safety and credibility. This approach makes healthcare more accessible to patients, empowering them to consult with trusted professionals at their convenience.

**Goals**

**• Accessibility:** Provide easy access to healthcare appointments across various devices, making it convenient for users to book a doctor anytime, anywhere.

**• Credibility:** Maintain a high standard of trustworthiness by having the admin approve doctors before they are listed, ensuring genuine healthcare providers are available.

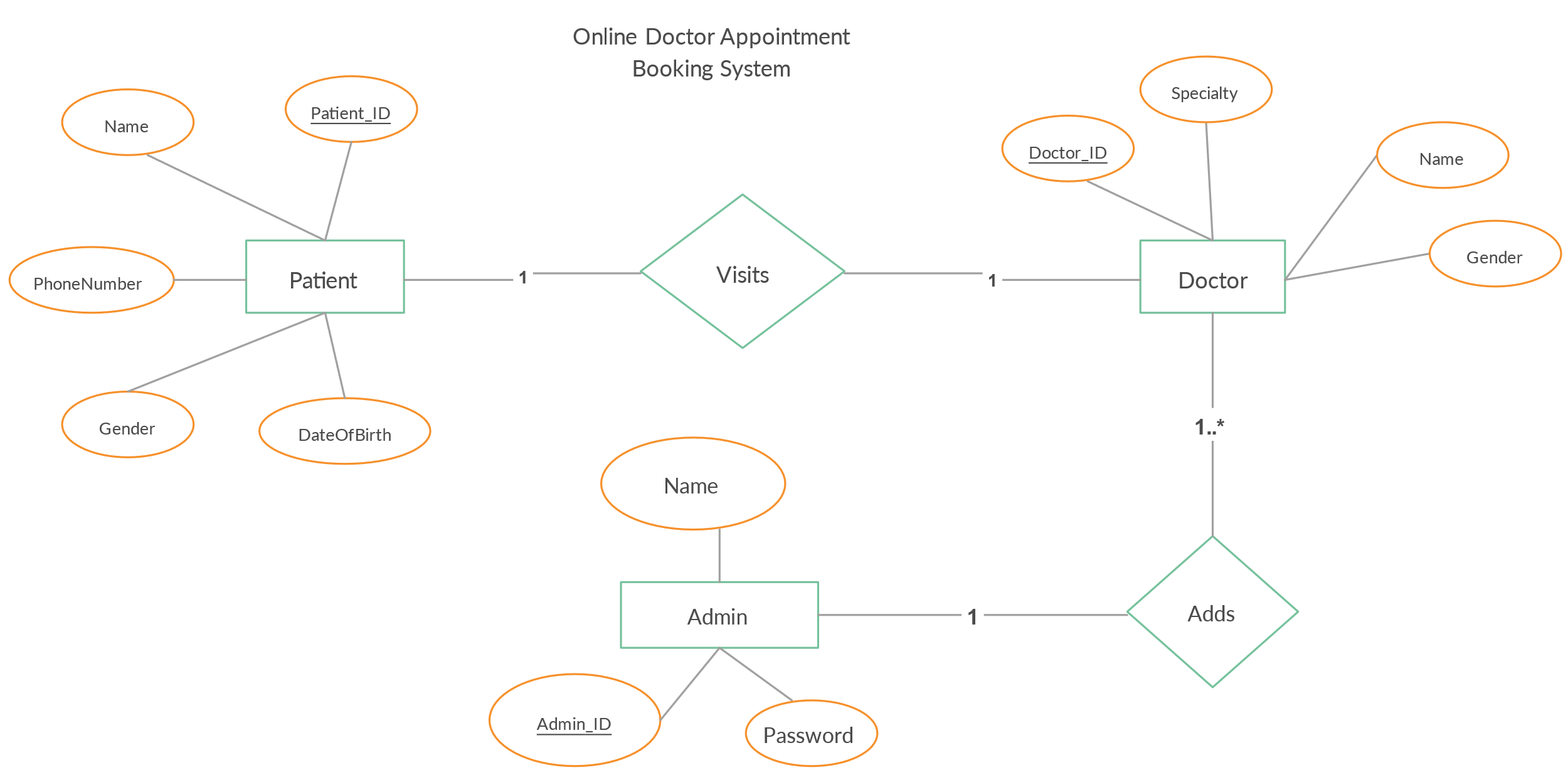
**• Efficiency:** Simplify the appointment scheduling process for users and doctors, ensuring a seamless experience.

**• User Engagement:** Ensure that both patients and doctors have interactive and efficient interfaces tailored to their roles.

**Features**

* **User-Friendly Interface:** A simple and intuitive design for users, doctors, and admins, making it easy to navigate and use the platform.
* **Doctor Application & Approval:** Doctors can apply to join the platform, and the admin reviews and approves these applications to ensure legitimacy and quality.
* **Appointment Management:** Users can book appointments with doctors. Doctors have the ability to approve or reschedule appointments based on availability.
* **Admin Dashboard:** The admin dashboard allows for the management of doctor applications, user appointments, and general platform operations, ensuring a secure and efficient environment.
* **Notification System:** Automated notifications and reminders for both users and doctors, helping keep track of appointments and approvals.
* **Doctor Profile:** Detailed profiles for doctors, including their specialization, qualifications, and experience, making it easier for users to choose the right doctor.
* **Scalability:** Designed to handle an increasing number of users, doctors, and appointments as the platform grows.
* **Admin Dashboard**: Admins have control over platform operations, including course monitoring, user management, and issue resolution.

**3. Architecture**



**Frontend Architecture (React)**

The frontend is developed using React, focusing on creating a responsive and user-friendly interface for both students and instructors. Key elements include:

* **Component-Based Structure**: React components for user registration, course browsing, course management, payment, and access to interactive elements.
* **State Management**: Context API or a state management library to manage user authentication, appointment status, and application approval.
* **Routing**: React Router for navigating between pages like application listings, booking options, and user dashboards.
* **Interactivity**: Interactive elements such as discussion forums and live webinars, utilizing third-party libraries or WebSocket for real-time communication.

**Backend Architecture (Node.js & Express.js)**

The backend, built with Node.js and Express.js, serves as the core of the application, managing requests and interacting with the database.

* **API Design**: RESTful APIs to handle user registration, course management, payment processing, and course content access.
* **Authentication & Authorization**: JWT (JSON Web Tokens) for secure user authentication and role-based access control for administrators, teachers, and students.
* **Appointment Management:** API endpoints for doctor registration, appointment scheduling, approval tracking, and managing appointment statuses.
* **Doctor Approval System:** Admin functionalities to review and approve doctor applications, ensuring that only verified and qualified doctors are listed on the platform.

**Database Schema and Interactions (MongoDB)**

The database is structured to handle various entities such as users, courses, enrollments, and progress tracking.

* **User Schema**: Stores user details, role (admin, doctor, or user), and contact information.
* **Doctor Schema**: Includes doctor details, specialization, qualifications, and approval status.
* **Appointment Schema**: Tracks appointment details, including user ID, doctor ID, appointment date, time, and status.
* **Approval Tracking**: Monitors the status of doctor applications and user appointments, ensuring that doctors and appointments are verified and managed appropriately.

**4. Setup Instructions**

**Prerequisites:**

1. **React.js:** A JavaScript library for building user interfaces.

* Install: npx create-react-app book-a-doctor to initialize your React project.

1. **Node.js & npm**: Required for server-side JavaScript.

* Install: [Node.js](https://nodejs.org/en/download/)

1. **Express.js**: Web framework for backend APIs.

* Install: npm install express

1. **MongoDB**: NoSQL database for storing app data.

* Install: [MongoDB](https://www.mongodb.com/try/download/community)

1. **CORS**: A middleware package to allow cross-origin requests from the frontend to the backend.

* Run npm install cors to handle cross-origin requests between your React frontend and Express backend..

1. **Mongoose**: ODM for MongoDB in Node.js.

* Install: npm install mongoose

**Installation Steps**

1. **Create React App**:

* git clone <repo-url>
* cd frontend
* npm install
* cd ../backend
* npm install

1. **Set Up Backend (Express + MongoDB)**:

* Create a backend folder, initialize Node.js, and install Express:
* cd backend
* npm init -y
* npm install express mongoose

1. **Start Servers**:

**Frontend**: In the React project folder:

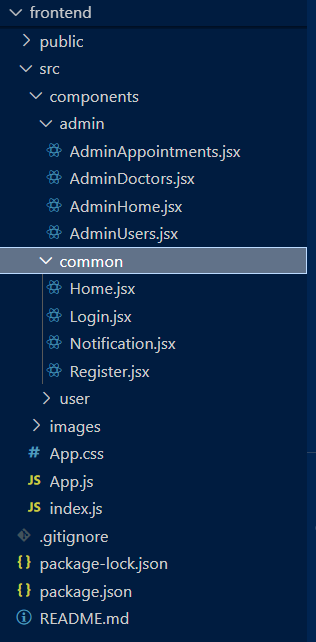
* npm start
* (Access at <http://localhost:3000>)

**Backend**: In the backend folder:

Npm run dev

**5. Folder Structure**

**Client Side (**React Frontend Structure)



* **node\_modules**: Contains the dependencies required for the React app to function. These are installed through npm and specified in the package.json file.
* **public**: Holds static files like index.html, which serves as the entry point for the React app. Any static assets accessible to the public can be placed here.
* **src**: The main source folder for the React app, containing:
* **Assets**: A directory for storing images, icons, or other static assets required by the application.
* **Components**: Contains the different components used across the app, divided into three subfolders:
  + **Admin**: Components used for admin functionalities, including AdminHome.jsx and AdminDoctors.jsx.
  + **Common**: Shared components accessible to various users, such as Notification.jsx, Home.jsx, and authentication components (Login.jsx, Register.jsx).
  + **User**: Components specific to user roles:
    - **User**: Components like ApplyDoctor.jsx, DoctorList.jsx, and userHome.jsx, meant for student functionalities.
* **App.css**: Contains global CSS styling for the application.
* **App.jsx**: The main component that sets up routing and renders the main application layout.
* **index.css**: Holds global styles that apply to the entire application.
* **App.js**: The entry point for rendering the React app.
* **Other Files**:
* **.gitignore**: Lists files and folders to ignore in Git version control.
* **eslint.config.js**: Configuration file for ESLint, which helps in maintaining code quality.

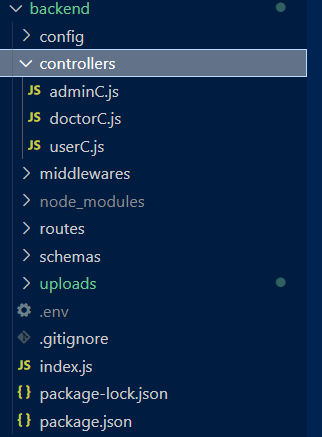
**Server Side**

1. **Config**: Contains configuration files, such as Connect.js, which handles the database connection setup.
2. **Controllers**: Contains the controller files, which define the logic for handling requests:

* **adminC.js**: Defines functions to handle admin-related operations.
* **doctorC.js**: Defines functions to handle user-related operations.

1. **Middlewares**: Holds middleware functions, like authMiddleware.js, which manage tasks such as authentication and authorization.
2. **Routers**: Contains route files that define the endpoints for different parts of the application:

* **adminRoutes.js**: Defines the routes for admin functionalities.
* **userRoutes.js**: Defines the routes for user functionalities.

****

1. **Schemas**: Stores Mongoose schema definitions for various collections in MongoDB:

* **appointmentModel.js**: Defines the schema for appointements.
* **docModel.js**: Defines the schema for doctors.
* **userModel.js**: Defines the schema for users.

1. **uploads**: Likely used to store uploaded files, though the exact purpose isn't specified in the structure.
2. **Other Files**:

* **.env**: Environment configuration file to store sensitive information, such as database credentials.
* **.gitignore**: Lists files and folders to ignore in Git version control.
* **index.js**: The main entry point for the Node.js application.
* **package.json** and **package-lock.json**: Files listing project dependencies and other metadata.

**6. Run the Application**

1. **Start the Frontend Server (React)**

* Navigate to the frontend directory and run the following command:

**cd frontend**

**npm start**

This command starts the react development server for the React frontend, typically accessible at http://localhost:3000 (or another port if 3000 is occupied).

1. **Start the Backend Server (Node.js + Express)**

Open a new terminal window, navigate to the backend directory named as Server, and run the following command:

**cd backend**

**npm run dev**

**7. API Documentation**

**1. User Login**

* **Endpoint**: /api/user/login
* **Method**: POST
* **Description**: Authenticates a user and generates a JWT token upon successful login.

**Request Parameters**:

* email (string): The email of the user.
* password (string): The password of the user.

**Example :**

**{**

**"email": "user@example.com",**

**"password": "password123"**

**}**

**Response**:

* success (boolean): Indicates whether the login was successful.
* message (string): A message providing additional information about the result of the login attempt.
* token (string): The JWT token generated for the user (if successful).
* userData (object): The user's data (if successful).

**{**

**"success": true,**

**"message": "Login successful",**

**"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",**

**"userData": {**

**"id": "12345",**

**"name": "John Doe",**

**"email": "user@example.com"**

**}**

**}**

**Usage Example:**

**axiosInstance.post('http://localhost:5000/api/user/login', data)**

**.then((res) => {**

**if (res.data.success) {**

**alert(res.data.message)**

**localStorage.setItem("token", res.data.token);**

**localStorage.setItem("user", JSON.stringify(res.data.userData));**

**navigate('/dashboard')**

**setTimeout(() => {**

**window.location.reload()**

**}, 1000)**

**} else {**

**alert(res.data.message)**

**}**

**})**

**.catch((err) => {**

**if (err.response && err.response.status === 401) {**

**alert("User doesn't exist");**

**}**

**navigate("/login");**

**});**

**8. Authentication**

Authentication in the Online Learning Platform (OLP) is managed using **JSON Web Tokens (JWT)**. JWT is used to securely verify the identity of users during login and ensure that only authorized users can access protected resources.

**Authentication Flow:**

* When a user logs in with their credentials (email and password), the server verifies the credentials against the database.
* If the credentials are correct, the server generates a JWT that contains the user’s information and sends it back to the client.
* The client stores this token (typically in localStorage or a cookie) for use in subsequent requests.

**Authorization:**

Authorization controls access to different resources based on the user's role or privileges.

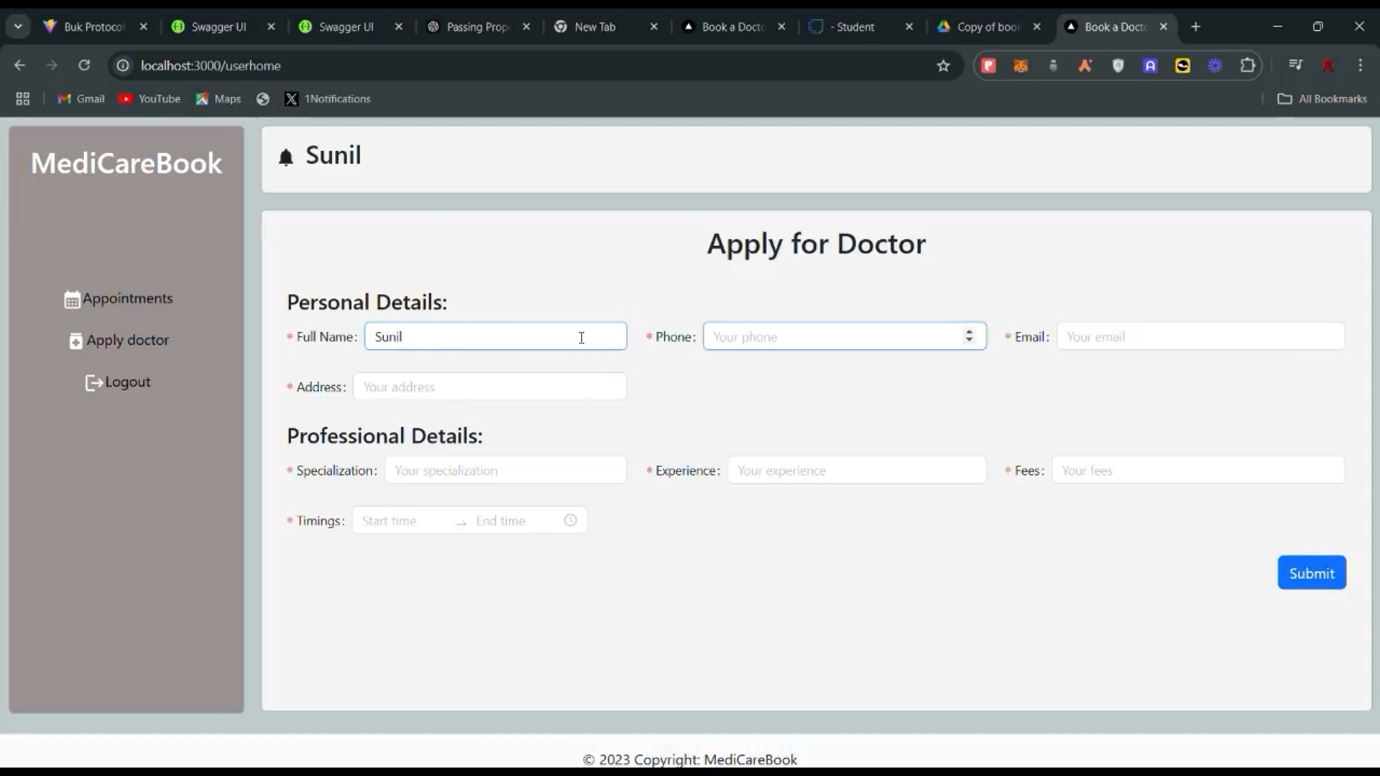
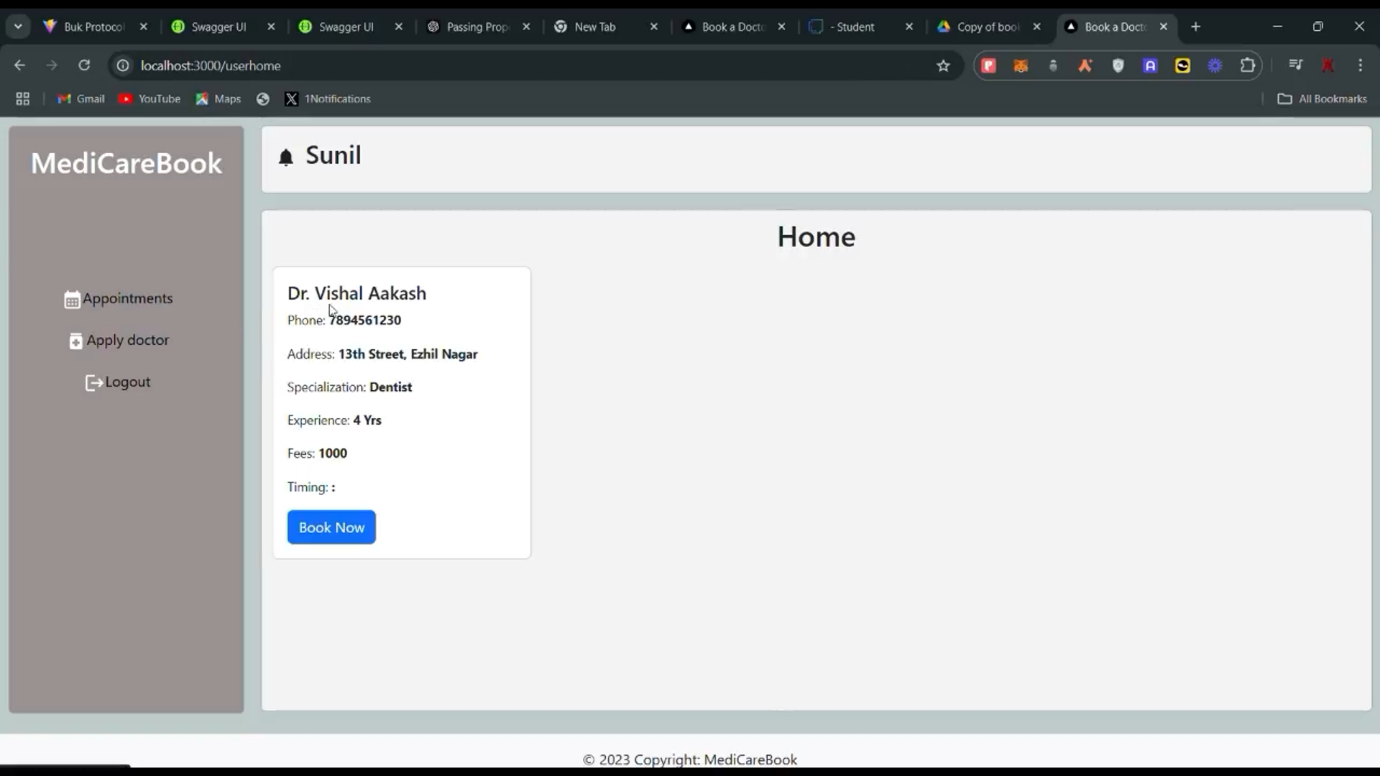
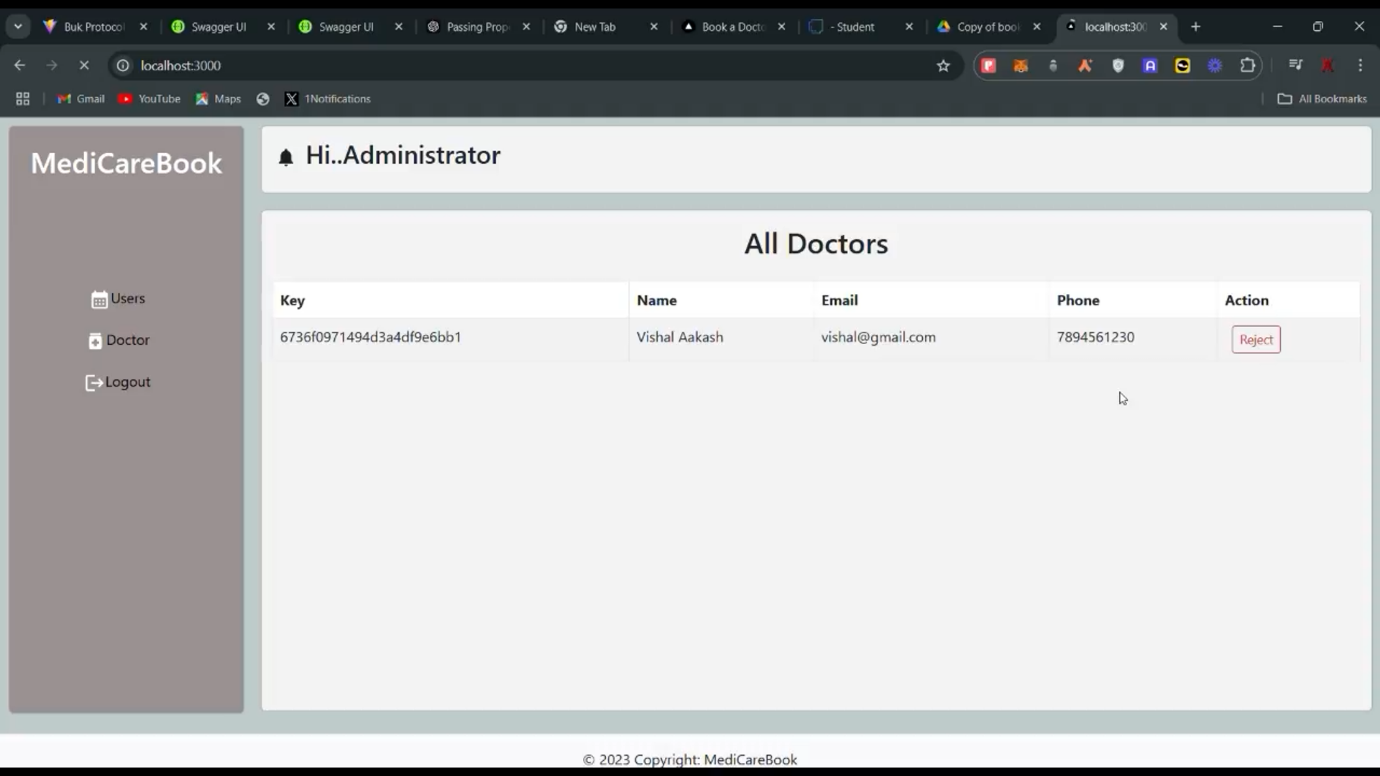
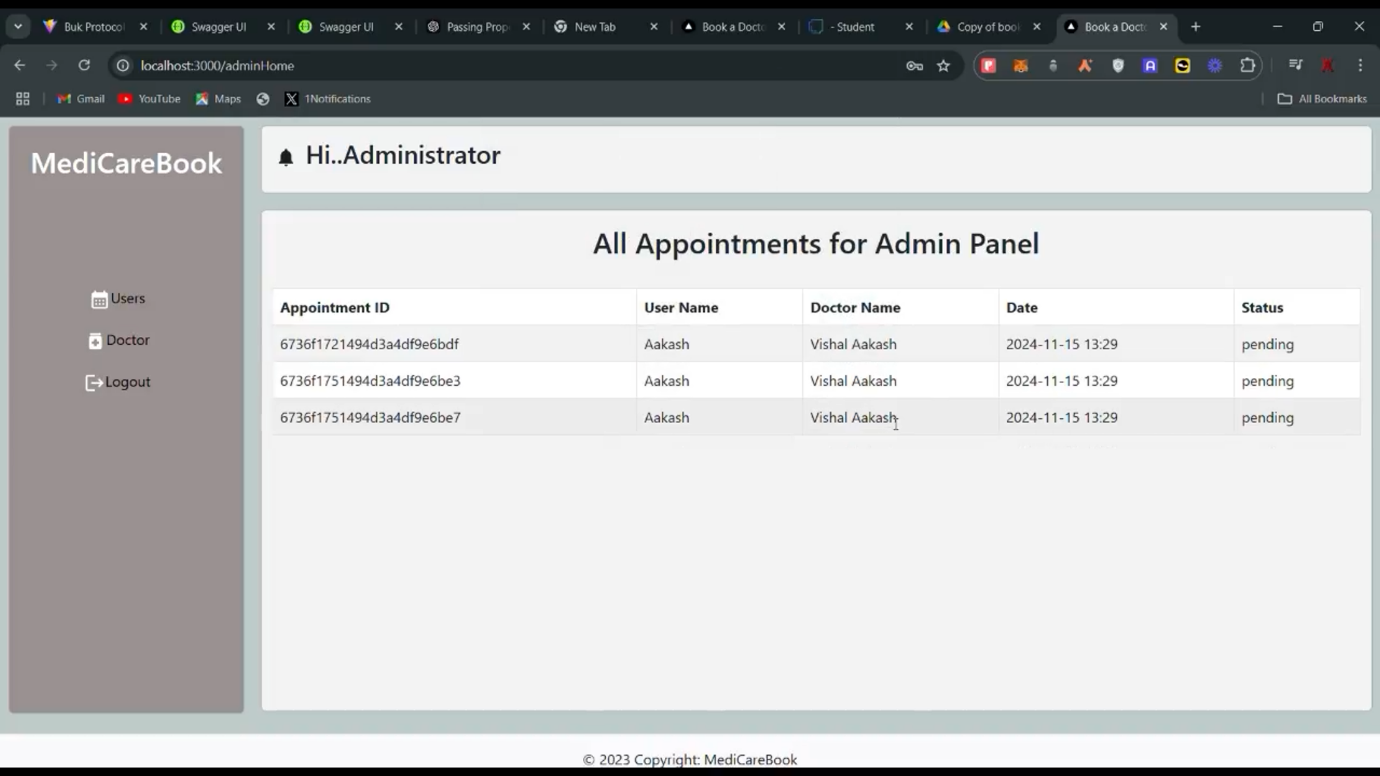
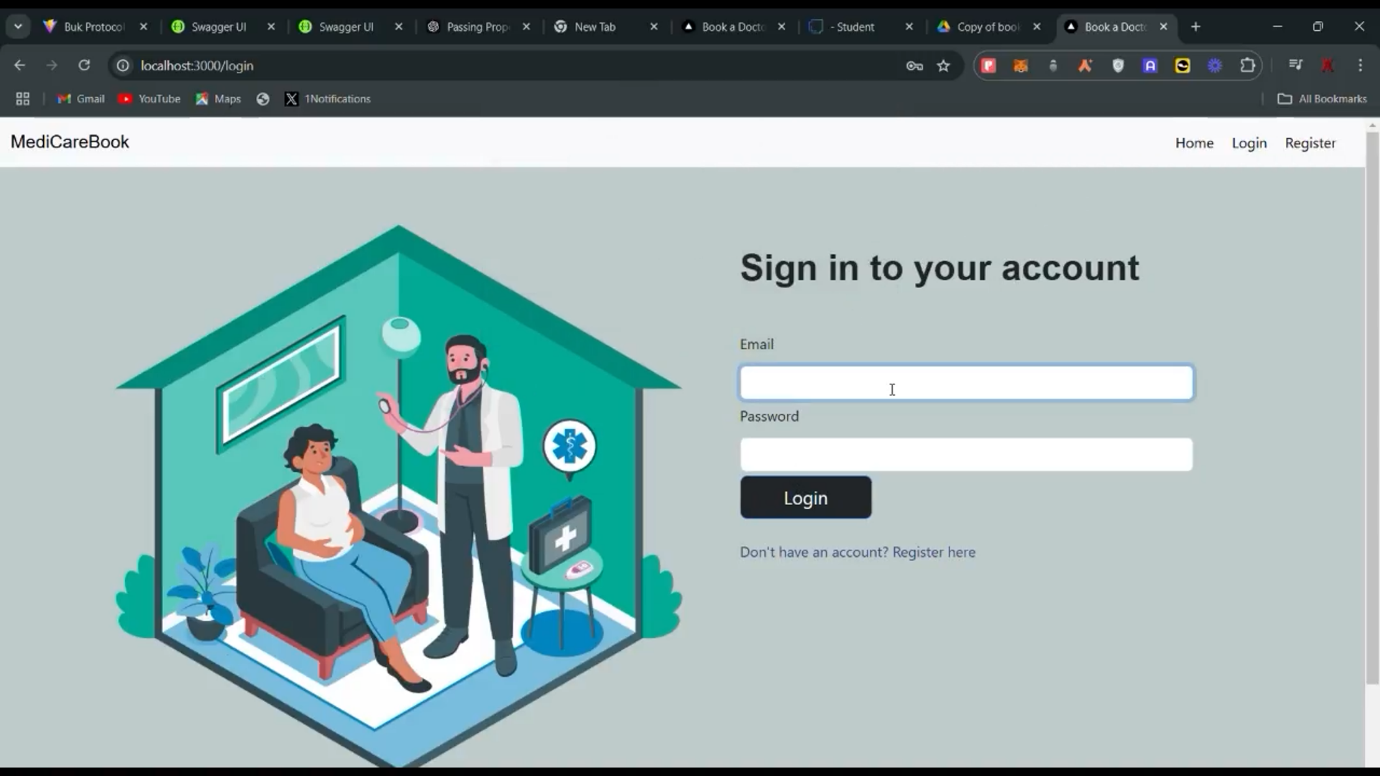
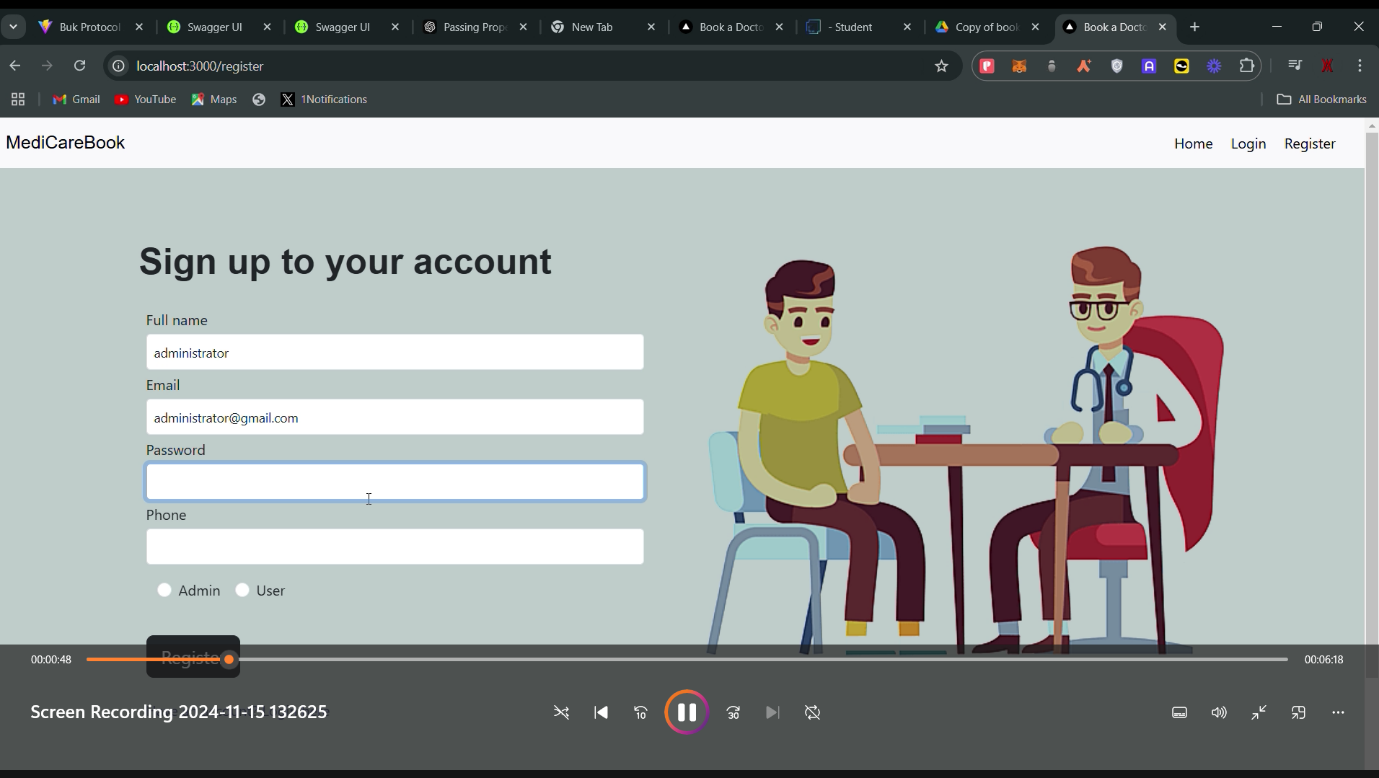
**Authorization Flow:**

* For each subsequent request to a protected route (e.g., accessing a course or dashboard), the client sends the JWT in the Authorization header of the HTTP request.
* The server validates the JWT and checks for its expiration. If the token is valid, the server grants access to the requested resource.
* Role-based authorization is applied by including roles within the JWT (e.g., admin, instructor, student), allowing the backend to check if a user has the correct permissions to perform a specific action or access a resource.
* If the token is expired or invalid, the server returns an error (e.g., 401 Unauthorized or 403 Forbidden).

**Token Expiration:**

JWTs have an expiration time, after which the token becomes invalid. When the token expires, users are required to re-authenticate or refresh their token to regain access.

**9. User Interface**

****

**10. Testing**

**Testing Strategy**: The testing strategy for the Online Learning Platform (OLP) combines manual and automated testing methods to ensure that all aspects of the platform function correctly and deliver a high-quality user experience. The primary focus is on functional and user experience testing, which ensures that both the frontend and backend work seamlessly together.

1. **Unit Testing**:

* **Purpose**: To test individual components or functions in isolation and ensure they work as expected.
* **Tool Used**: **Jest**

Jest is used to perform unit testing on React components. It helps verify that each component is rendered correctly, responds to user interactions, and handles the state as expected.

1. **End-to-End (E2E) Testing**:

* **Purpose**: To test the entire application flow from the user's perspective, ensuring that the platform works as intended when users interact with it.
* **Tool Used**: **Cypress**

Cypress is employed for end-to-end testing. It simulates real-world user interactions, such as signing up, enrolling in a course, and navigating through the learning platform, to ensure all critical workflows are functioning smoothly.

1. **Performance Testing**:

* **Purpose**: To evaluate the platform's performance under various conditions, ensuring it handles traffic and user load efficiently.
* **Tool Used**: **Google Lighthouse**

Google Lighthouse is used to assess performance metrics, such as load times, responsiveness, and other crucial user experience factors.

**11. Demo**

Watch the demo video of the Book A Doctor application to see how it works in real-time. The video provides an overview of key features, including user registration, doctor approval by the admin, appointment booking, and how doctors manage and approve appointments. This demo will give you a comprehensive understanding of the platform’s functionality and user experience.

Drive Link: <https://drive.google.com/drive/u/3/folders/1Cju4slrYkqn7n3jr26gj4Thq3BbMPri1>

**12. Known Issues**

1. **Authentication Token Expiry**
   * **Issue**: Users may be logged out unexpectedly due to token expiry.
   * **Impact**: Frequent re-logins required, causing inconvenience.
   * **Solution**: Adjust token expiry settings and implement a token refresh mechanism to maintain user sessions seamlessly.
2. **Doctor Approval Delays**
   * **Issue**: Admin approval for doctor applications may take longer than expected.
   * **Impact**: Doctors may experience delays in getting listed on the platform.
   * **Solution**: Set up automated reminders for the admin or optimize the approval workflow.
3. **Appointment Scheduling Conflicts**
   * **Issue**: Double bookings or appointment overlaps may occur if multiple users book simultaneously.
   * **Impact**: Doctors may have difficulty managing their schedules effectively.
   * **Solution**: Implement real-time availability checks and locking mechanisms during the booking process.
4. **Email Notification Delays**
   * **Issue**: Delayed or missing email notifications for appointment confirmations and updates.
   * **Impact**: Users and doctors may miss important information, leading to confusion.
   * **Solution**: Verify email configuration and optimize the SMTP server settings to ensure timely notifications.
5. **Appointment Status Syncing**
   * **Issue**: Appointment status updates may not reflect immediately across all parts of the application.
   * **Impact**: Users and doctors may see outdated information.
   * **Solution**: Implement efficient database syncing and refresh mechanisms to keep appointment statuses up-to-date.

**13. Future Enhancements**

1. **User Analytics**: Track user engagement, appointment trends, and the most frequently consulted doctors for better insights.
2. **Advanced Search and Filtering**: Enable search by doctor specialization, location, availability, and user reviews.
3. **Real-time Notifications**: Alert users and doctors about appointment confirmations, reminders, and important updates.
4. **Mobile App**: Develop iOS and Android apps to allow users to book and manage appointments conveniently on the go.
5. **Doctor Rating and Review** System: Allow users to rate and review doctors based on their experience to improve the quality of service.
6. **Multi-language Support**: Localize the application to support multiple languages and cater to a global user base.
7. **Video Consultation Feature**: Add a secure video call option for users to consult with doctors remotely.
8. **Appointment Reminders**: Implement automated reminders via SMS and email to reduce missed appointments.
9. **Enhanced Doctor Profile**: Include detailed profiles with verified qualifications, experience, and specialization.
10. **Accessibility Features**: Add support for screen readers, high contrast modes, and adjustable font sizes to make the app more inclusive.
11. **Integration with Health Records**: Allow users to upload and manage their medical records securely for doctors to review before appointments.
12. **Appointment History**: Provide a detailed history of past appointments and allow easy re-booking with the same doctor.
13. **Doctor Availability Calendar**: Enhance scheduling by showing real-time availability and providing better appointment management tools.
14. **Admin Dashboard**: Improve with advanced data visualization, doctor approval statistics, and user activity monitoring.
15. **Content Management System (CMS):** Create a CMS for admins to manage doctor information, appointment policies, and content updates easily.
16. **Automated Follow-up Messages**: Send post-appointment follow-ups for user feedback and doctor recommendations.