**Doctor Appointment Booking System**

**Team Members:**

**Team Leader : Govada Sankara Naga Shyam**

**Team Member: Goli Komal**

**Team Member: Goli Sai Charan Reddy**

**Team Member: Goli Subhash**

1. Project Overview

The Doctor Appointment Booking System is a full-stack web application developed using the MERN (MongoDB, Express.js, React, Node.js) stack. It facilitates the process of booking, managing, and reviewing doctor appointments online. The system includes functionalities for different user roles: standard users, doctors, and administrators, providing tailored interfaces and capabilities for each.

The project aims to streamline the traditional appointment booking process, offering a convenient and efficient platform for both patients and healthcare providers.

2. Features

The system supports the following key features:

* **User Authentication:** Secure registration and login functionality using JWT (JSON Web Tokens).
* **Role-Based Access Control:** Different functionalities and data access based on user roles (User, Doctor, Admin).
* **User Registration:** Public registration for standard users. Includes a specific logic allowing only the very first user to register as an administrator, while subsequent registrations selecting 'Admin' are rejected for security.
* **User Login:** Authenticated login for registered users (User, Doctor, Admin).
* **Doctor Application:** Standard users can apply to become doctors by submitting their professional details.
* **Admin Dashboard:** A dedicated section for administrators to manage users, doctor applications, and view all appointments.
* **Doctor Management (Admin):** Administrators can view pending doctor applications and approve or reject them. Approval grants the user doctor privileges.
* **User Management (Admin):** Administrators can view a list of all registered users.
* **Admin Creation (Admin):** Existing administrators can create new administrator accounts via a protected interface.
* **Doctor Listing (User):** Standard users can view a list of approved doctors available for booking.
* **Appointment Booking (User):** Standard users can book appointments with approved doctors by selecting a date/time and optionally uploading a medical document.
* **User Appointments (User):** Users can view their own list of booked appointments and their status.
* **Doctor Appointments (Doctor):** Doctors can view a list of appointments booked by patients with them.
* **Appointment Status Handling (Doctor):** Doctors can update the status of pending appointments (e.g., Approve, Reject).
* **Document Upload/Download:** Users can upload documents during booking, and doctors can download these documents.
* **Notifications:** Users and doctors receive notifications for relevant events (e.g., doctor application status, new appointment requests, appointment status updates).
* **Responsive Design:** Basic responsiveness implemented using React Bootstrap.

3. Technologies Used

**Frontend:**

* **React v18:** JavaScript library for building the user interface.
* **React Router DOM v6:** For handling navigation and routing within the application.
* **Vite:** Frontend build tool, providing a fast development environment.
* **React Bootstrap v2:** UI library for styling and responsive components.
* **Ant Design v5:** UI library used for specific components like Messages, Modals, Tabs, TimePicker, Badges, Row/Col (used alongside Bootstrap grid).
* **Axios:** HTTP client for making API requests to the backend.
* @mui/icons-material: React components for Material Design icons.

**Backend:**

* **Node.js:** JavaScript runtime environment.
* **Express.js v5:** Web application framework for building the API.
* **MongoDB:** NoSQL document database for data storage.
* **Mongoose v8:** Object Data Modeling (ODM) library for MongoDB and Node.js.
* **bcryptjs v3:** Library for hashing passwords securely.
* **jsonwebtoken v9:** For generating and verifying JWTs for authentication.
* **Multer v2:** Middleware for handling multipart/form-data (file uploads).
* **CORS v2:** Middleware to enable Cross-Origin Resource Sharing.
* **Dotenv v16:** For loading environment variables from a .env file.

4. Architecture

The project follows a standard **MERN stack** architecture:

* **Frontend (Client):** A React application running in the user's browser. It's responsible for presenting the UI and capturing user interactions. It communicates with the backend solely through HTTP requests (API calls).
* **Backend (Server):** A Node.js/Express application. It handles all server-side logic, interacts with the database, enforces security rules (authentication, authorization), and provides RESTful API endpoints.
* **Database:** MongoDB, accessed by the backend via Mongoose. It stores all application data in a document-based structure.

The interaction flow is typically:

1. The user interacts with the **Frontend** (e.g., fills a form, clicks a button).
2. The Frontend makes an **API call** (HTTP request using Axios) to a specific endpoint on the **Backend**.
3. The Backend receives the request. **Middleware** (authMiddleware, Multer) might process the request first (e.g., verifying the token, handling file uploads).
4. A **Controller** function is executed, containing the main logic. It might interact with the **Database** (using Mongoose Schemas/Models) to read or write data.
5. The Controller prepares a response (data and status code).
6. The Backend sends the **response** back to the Frontend.
7. The Frontend receives the response, updates its state, and dynamically renders the UI based on the data received.

Protected routes and actions are secured by authMiddleware, which runs before the controller and verifies the user's identity and potentially their role based on the JWT token provided in the request headers. Role-specific logic is also implemented within the controllers.

5. Setup and Installation

To run the project locally, follow these steps:

**5.1 Prerequisites:**

* Node.js and npm (or Yarn) installed.
* A MongoDB Atlas account and cluster set up. Obtain your connection string. Ensure your current IP address is whitelisted in your Atlas Network Access settings.
* A code editor (like VS Code).

**5.2 Backend Setup:**

1. Navigate into the backend directory in your terminal.
2. Install dependencies: npm install
3. Create a .env file in the backend directory.
4. Add your MongoDB connection string and JWT secret key:

Generated env

MONGO\_DB=YOUR\_MONGODB\_ATLAS\_CONNECTION\_STRING

JWT\_KEY=a\_long\_random\_string\_for\_jwt\_signing

PORT=5000

content\_copydownload

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).Env

*Replace placeholders with your actual credentials.*

1. Create an uploads directory in the backend directory. This is where uploaded documents will be stored.
2. Start the backend server: node index.js
3. Confirm the server connects to MongoDB and starts listening on the specified port.

**5.3 Frontend Setup:**

1. Open a *new* terminal window and navigate into the frontend directory.
2. Install dependencies: npm install
3. Create a .env file in the frontend directory.
4. Add the URL of your backend API:

Generated env

VITE\_API\_URL=http://localhost:5000/api

content\_copydownload

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).Env

*Ensure the port matches the backend PORT.*

1. Start the frontend development server: npm run dev
2. Open the provided local URL (e.g., http://localhost:5173/) in your web browser.

**5.4 Creating the Initial Admin User:**

* The public registration form allows the *first* user to register as an admin.
* Go to the /register page in your browser.
* Fill out the form with the desired details for your first admin account.
* Select the "Admin" radio button.
* Submit the form. If no admin user exists in the database, this user will be created with type: 'admin'.
* If an admin *already* exists, you will receive an error message. In this case, you would need to create the first admin manually in the database (as described in previous explanations) if one wasn't created correctly through the form, or log in with the existing admin account.

6. Frontend Explanation

The frontend is built using React and structured into components within the src directory.

* **src/main.jsx:** The entry point, rendering the App component wrapped in BrowserRouter for routing and importing global styles (index.css, App.css, Bootstrap, Ant Design CSS).
* **src/App.jsx:** Configures the main routing for the application using react-router-dom. It includes logic to check if a user is logged in (by checking localStorage) and their type (userType) to render different components or redirect to the login page for protected routes (/adminhome, /userhome).
* **src/App.css:** Contains general styling for the application layout (main, layout, sidebar, content, header, body) and common elements.
* **src/index.css:** Provides basic CSS resets and default styles.
* **src/utils/axiosConfig.js:** Configures an Axios instance with the backend API base URL. It includes a request interceptor to automatically add the JWT token from localStorage to the Authorization header for every outgoing request, simplifying authenticated API calls in components. It also includes a basic response interceptor to handle 401 Unauthorized errors by clearing storage and forcing a page reload.

**6.1 Common Components (src/components/common/)**

* **Home.jsx:** The landing page component (/). Displays introductory text and an image. Includes a navigation bar with links to Home, Login, and Register.
* **Login.jsx:** Provides the user login form (/login). Handles capturing email and password, sending a POST request to /api/user/login. On successful login, it stores the token and user data in localStorage and forces a page reload (window.location.reload()) to trigger the routing logic in App.jsx.
* **Register.jsx:** Provides the user registration form (/register). Captures user details including desired type ("User" or "Admin"). Sends a POST request to /api/user/register. Includes basic frontend validation and displays backend success/error messages (including the specific message if admin registration is attempted after the first admin is created). Redirects to login on success.
* **Notification.jsx:** Displays user notifications (/userhome/notification or /adminhome/notification). Fetches notification data from the user object stored in localStorage (updated via API calls when notifications are marked/deleted). Allows marking all unread as read and deleting all seen notifications by calling backend API endpoints. Uses Ant Design Tabs to switch between unread and seen notifications.

**6.2 Admin Components (src/components/admin/)**

* **AdminHome.jsx:** The main layout and hub for administrators (/adminhome). Displays a sidebar menu, header with admin name and notification count, and a main content area. Uses state (activeMenuItem) to control which specific admin sub-component is rendered in the content area. Fetches admin's user data from localStorage on mount.
* **AdminUsers.jsx:** Renders a table listing all registered users in the system. Fetches data from /api/admin/getallusers.
* **AdminDoctors.jsx:** Renders a table listing all doctor applications/profiles (pending, approved, rejected). Fetches data from /api/admin/getalldoctors. Provides buttons to Approve/Reject pending applications, which call backend endpoints and refresh the list.
* **AdminAppointments.jsx:** Renders a table listing *all* appointments in the system. Fetches data from /api/admin/getallAppointmentsAdmin.
* **AdminRegister.jsx:** Provides a form for an *existing* admin to create *new* admin user accounts. This component is rendered within AdminHome when the "Create Admin" menu item is selected. Sends data to the protected /api/admin/registeradmin endpoint.

**6.3 User Components (src/components/user/)**

* **UserHome.jsx:** The main layout and hub for standard users and doctors (/userhome). Displays a sidebar menu, header with user/doctor name and notification count, and a main content area. Uses state (activeMenuItem) to control the rendered sub-component. Dynamically shows/hides the "Apply doctor" menu item based on the user's isdoctor status. Fetches user data and the initial list of approved doctors on mount.
* **ApplyDoctor.jsx:** Provides a form for a standard user to apply to become a doctor. This component is rendered within UserHome if the user is not already a doctor and selects "Apply doctor". Sends the application details to /api/user/registerdoc. Requires the authenticated user's ID (passed as a prop).
* **DoctorList.jsx:** Renders a card displaying details for a single approved doctor. This is rendered as part of the "Home" view in UserHome. Includes a "Book Now" button which triggers a modal.
* **UserAppointments.jsx:** Renders a table listing appointments. This component is rendered within UserHome when "Appointments" is selected. It dynamically shows the user's booked appointments (fetching from /api/user/getuserappointments) if they are a standard user, or appointments booked with them (fetching from /api/doctor/getdoctorappointments) if they are a doctor. It also includes action buttons for doctors to handle appointment status and a download link for patient documents.

7. Backend Explanation

The backend is an Express.js application structured logically into directories.

* **index.js:** The main server file. It sets up the Express app, connects to the MongoDB database (config/connectToDb.js), applies essential middleware (express.json, cors), serves static files from the uploads directory (express.static), defines the base paths for different route groups (app.use('/api/user', ...)), and includes a basic global error handler. Finally, it starts the server.
* **config/connectToDb.js:** Contains the Mongoose connection setup to MongoDB using the connection string from environment variables.
* **middlewares/authMiddleware.js:** A crucial middleware function used on protected routes. It extracts the JWT token from the Authorization header, verifies it, fetches the corresponding user from the database (excluding the password and using .lean() for performance), attaches the user object (req.user) to the request, and calls next() to allow the request to proceed to the controller. It sends 401/500 responses if authentication fails.
* **schemas/\*.js:** Defines the Mongoose schemas (models) that shape the data stored in MongoDB.
  + **userModel.js:** Defines the structure for user documents (fullName, email, password, phone, type, notification, seennotification, isdoctor). Includes validations (required, unique email), password selection exclusion, and default values.
  + **docModel.js:** Defines the structure for doctor profile documents (userId, fullName, email, phone, address, specialization, experience, fees, status, timings). Links to the user model via userId. Includes validations (required fields, unique userId and email, number types for experience/fees, enum for status).
  + **appointmentModel.js:** Defines the structure for appointment documents (userId, doctorId, date, document, status). Links to user and doctor models. Stores document information (filename, path). Includes enum for status and default. *Does NOT store full userInfo/doctorInfo objects, relying on population.*
* **controllers/\*.js:** Contains the logic executed for each API endpoint. Controllers receive the req (request) and res (response) objects. They perform validation, interact with Mongoose models, implement business logic, and send the final HTTP response.
  + **userC.js:** Handles user-related logic:
    - registerController: Handles public user registration. Includes the logic to allow only the first 'admin' registration. Hashes passwords, creates user documents, and saves them.
    - loginController: Handles user login. Finds user, compares password, generates JWT token, and sends back token and user data.
    - authController: Used by authMiddleware to return the authenticated user's data.
    - docController: Handles a standard user applying to become a doctor. Creates a pending doctor profile and notifies the admin.
    - getallnotificationController: Marks user's unread notifications as read and moves them to seen.
    - deleteallnotificationController: Clears user's seen notifications.
    - getAllDoctorsControllers: Fetches a list of *approved* doctors for standard users to browse.
    - appointmentController: Handles user booking an appointment. Creates a pending appointment, handles document upload via Multer, and notifies the doctor.
    - getAllUserAppointments: Fetches appointments booked *by* the authenticated user.
  + **adminC.js:** Handles administrator-related logic:
    - getAllUsersControllers: Fetches all user accounts.
    - getAllDoctorsControllers: Fetches all doctor applications/profiles.
    - updateDoctorStatusController (used by /getapprove and /getreject routes): Updates a doctor's status (approved/rejected), updates the linked user's isdoctor flag, and notifies the user.
    - displayAllAppointmentController: Fetches *all* appointments in the system. Populates user and doctor names.
    - registerAdminController: Handles creating a *new* admin user. **Protected endpoint**, only usable by existing admins. Hashes password, creates the admin user document.
  + **doctorC.js:** Handles doctor-related logic:
    - updateDoctorProfileController: Allows an authenticated doctor to update their own profile details.
    - getAllDoctorAppointmentsController: Fetches appointments booked *with* the authenticated doctor.
    - handleStatusController: Allows an authenticated doctor to change the status of one of their appointments (approved/rejected) and notifies the patient.
    - documentDownloadController: Allows an authenticated doctor to securely download a document associated with one of their appointments. Includes path validation and file streaming.
* **routes/\*.js:** Defines the API endpoints (paths and HTTP methods) and links them to the appropriate middleware (authMiddleware, multer) and controller functions.
  + userRoutes.js: Contains public routes (/register, /login) and authenticated user routes (/getuserdata, /registerdoc, /getalldoctorsu, /getappointment, /getallnotification, /deleteallnotification, /getuserappointments). Configures Multer for appointment document uploads on /getappointment.
  + adminRoutes.js: Contains authenticated admin routes (/getallusers, /getalldoctors, /getapprove, /getreject, /getallAppointmentsAdmin, /registeradmin). Uses authMiddleware on all routes.
  + doctorRoutes.js: Contains authenticated doctor routes (/updateprofile, /getdoctorappointments, /handlestatus, /getdocumentdownload). Uses authMiddleware on all routes.

8. API Endpoints Reference

Here is a list of the main API endpoints:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Method** | **Path** | **Description** | **Auth Required?** | **Role Required?** | **Request Body** | **Response Data (on success)** |
| POST | /api/user/register | Register a new user (User or First Admin). | No | N/A | fullName, email, password, phone, type | message, success |
| POST | /api/user/login | Log in a user. | No | N/A | email, password | message, success, token, userData |
| POST | /api/user/getuserdata | Get authenticated user's data. | Yes | Any authenticated | (Empty or {}) | success, data (user object without password) |
| POST | /api/user/registerdoc | Apply to become a doctor. | Yes | User | { doctor: { ...doctor profile fields }, userId: userId } | success, message |
| GET | /api/user/getalldoctorsu | Get a list of approved doctors. | Yes | User | (None) | message, success, data (array of approved doctors) |
| POST | /api/user/getappointment | Book an appointment with a doctor. Includes document upload. | Yes | User | doctorId, date, document (file upload) | message, success |
| GET | /api/user/getuserappointments | Get appointments booked by the authenticated user. | Yes | User | (None, userId as query param handled by controller from req.user) | message, success, data (array of user appointments with doctor name) |
| POST | /api/user/getallnotification | Mark all user's unread notifications as read. | Yes | Any authenticated | (Empty or {}) | success, message, data (updated user object) |
| POST | /api/user/deleteallnotification | Delete all user's seen notifications. | Yes | Any authenticated | (Empty or {}) | success, message, data (updated user object) |
| GET | /api/admin/getallusers | Get a list of all users. | Yes | Admin | (None) | message, success, data (array of users) |
| GET | /api/admin/getalldoctors | Get a list of all doctor applications/profiles. | Yes | Admin | (None) | message, success, data (array of doctor profiles) |
| POST | /api/admin/getapprove | Approve a doctor application. | Yes | Admin | doctorId, status ("approved") | message, success, data (updated doctor profile) |
| POST | /api/admin/getreject | Reject a doctor application. | Yes | Admin | doctorId, status ("rejected") | message, success, data (updated doctor profile) |
| GET | /api/admin/getallAppointmentsAdmin | Get a list of all appointments. | Yes | Admin | (None) | success, message, data (array of all appointments with names) |
| POST | /api/admin/registeradmin | Create a new admin user account. | Yes | Admin | fullName, email, password, phone | message, success |
| POST | /api/doctor/updateprofile | Update authenticated doctor's profile. | Yes | Doctor | { ...doctor profile fields to update } | success, data (updated doctor profile), message |
| POST | /api/doctor/getdoctorappointments | Get appointments booked with the authenticated doctor. | Yes | Doctor | (Empty or {}) | message, success, data (array of doctor appointments) |
| POST | /api/doctor/handlestatus | Update status of an appointment belonging to the authenticated doctor. | Yes | Doctor | appointmentId, status ("approved" or "rejected"), userid (patient's ID for notification) | success, message |
| GET | /api/doctor/getdocumentdownload | Download a patient document for a specific appointment. | Yes | Doctor | (None, appointId as query param) | File stream (Blob) |

9. Database Schema

The database consists of three main collections, managed by Mongoose schemas:

**9.1 User Model (userModel.js)**

Represents a user account (standard user, doctor, or admin).

* \_id: ObjectId (Automatically generated unique ID)
* fullName: String, Required, Trimmed, First letter capitalized.
* email: String, Required, Unique, Lowercase, Trimmed.
* password: String, Required, select: false (Excluded by default in queries).
* phone: String, Required, Trimmed.
* type: String, Required, Enum: ['user', 'admin'], Default: 'user'. Defines the user's role.
* notification: Array of Objects (e.g., { type: String, message: String, data?: Object, onClickPath?: String }), Default: []. Stores unread notifications.
* seennotification: Array of Objects, Default: []. Stores read notifications.
* isdoctor: Boolean, Default: false. Indicates if this user also has a doctor profile and privileges.
* createdAt: Date (Automatically added if timestamps: true)
* updatedAt: Date (Automatically added if timestamps: true)

**9.2 Doctor Model (docModel.js)**

Represents a doctor's professional profile. Linked to a user account.

* \_id: ObjectId (Automatically generated unique ID)
* userId: ObjectId, Required, Unique, Ref: 'user'. Links the doctor profile to a specific user account.
* fullName: String, Required, Trimmed, First letter capitalized. Doctor's name (might be same as user's).
* email: String, Required, Unique, Lowercase, Trimmed. Doctor's professional email.
* phone: String, Required, Trimmed. Doctor's professional phone.
* address: String, Required, Trimmed. Doctor's clinic/practice address.
* specialization: String, Required, Trimmed.
* experience: Number, Required, Minimum 0. Years of experience.
* fees: Number, Required, Minimum 0. Consultation fees.
* status: String, Required, Enum: ['pending', 'approved', 'rejected'], Default: 'pending'. Approval status of the doctor application.
* timings: Object (or [String]), Required. Doctor's availability times (e.g., { start: 'HH:mm', end: 'HH:mm' } or ['HH:mm', 'HH:mm']).
* createdAt: Date (Automatically added if timestamps: true)
* updatedAt: Date (Automatically added if timestamps: true)

**9.3 Appointment Model (appointmentModel.js)**

Represents a single appointment booking.

* \_id: ObjectId (Automatically generated unique ID)
* userId: ObjectId, Required, Ref: 'user'. The user (patient) who booked the appointment.
* doctorId: ObjectId, Required, Ref: 'doctor'. The doctor the appointment is with.
* date: String, Required. The date and time of the appointment (stored as a string, e.g., "YYYY-MM-DDTHH:mm").
* document: Object (Optional). Stores information about an uploaded document.
  + filename: String. The name of the file.
  + path: String. The relative path to the stored file on the server (e.g., /uploads/timestamp-filename.pdf).
* status: String, Required, Enum: ['pending', 'approved', 'rejected'], Default: 'pending'. The current status of the appointment.
* createdAt: Date (Automatically added if timestamps: true)
* updatedAt: Date (Automatically added if timestamps: true)

10. User Flows

**10.1 Standard User Flow:**

1. **Registration:** Go to /register, select "User", fill details, submit.
2. **Login:** Go to /login, enter email/password, submit. Redirected to /userhome.
3. **Browse Doctors:** On /userhome, the default view is a list of approved doctors.
4. **Book Appointment:** Click "Book Now" on a doctor card, select date/time, upload document (optional), submit modal.
5. **View Appointments:** Click "Appointments" in the sidebar on /userhome. See a table of appointments booked.
6. **Apply as Doctor:** If not already a doctor, click "Apply doctor" in the sidebar on /userhome, fill form, submit. Waits for admin approval.
7. **Notifications:** Click the notification icon in the header to view new (unread) and read notifications. Can mark read or delete.

**10.2 Doctor Flow:**

1. **Become a Doctor:** Starts as a standard user, applies, gets approved by admin (user's isdoctor becomes true).
2. **Login:** Log in via /login. Redirected to /userhome. User Home sidebar menu dynamically adjusts (e.g., "Apply doctor" might be hidden or replaced with "Profile").
3. **View Appointments:** Click "Appointments" in the sidebar on /userhome. See a table of appointments booked *with them*.
4. **Handle Status:** In the appointment table, for 'pending' appointments, click "Approve" or "Reject".
5. **Download Documents:** In the appointment table, click the document link to download the patient's uploaded file.
6. **Notifications:** Receive notifications for new appointment requests and view them via the header icon.

**10.3 Administrator Flow:**

1. **Initial Creation:** First admin created manually in the database.
2. **Login:** Log in via /login. Redirected to /adminhome.
3. **Manage Users:** Click "Users" in the sidebar on /adminhome to view all users.
4. **Manage Doctors:** Click "Doctors" in the sidebar on /adminhome to view all doctor applications (pending, approved, rejected) and approve/reject pending ones. Receives notifications for new applications.
5. **Manage Appointments:** Click "Appointments" in the sidebar on /adminhome to view all appointments in the system.
6. **Create Admins:** Click "Create Admin" in the sidebar on /adminhome, fill form, submit to create a new admin user account.
7. **Notifications:** Receive notifications for new doctor applications and view them via the header icon.

11. Future Improvements / Enhancements

* **Improved State Management:** Implement React Context API or a library like Redux Toolkit for better global state management (user authentication status, notifications) across components, replacing direct localStorage checks and window.location.reload().
* **More Robust Doctor Approval:** Add more details to the admin's view of doctor applications, potentially allowing editing before approval.
* **Doctor Profile Page:** Create a dedicated page for doctors to view and manage their profile details.
* **Appointment Management Features:** Allow users to cancel appointments (with rules), allow doctors to reschedule, implement time slot availability logic.
* **Calendar View:** Display doctor availability and booked appointments in a calendar format.
* **Search and Filtering:** Add search bars and filters for doctors, users, and appointments.
* **User Profiles:** Allow users to view and edit their own profiles.
* **Password Reset:** Implement forgotten password functionality.
* **Email Notifications:** Integrate an email service to send automated notifications (e.g., booking confirmation, approval status).
* **Enhanced UI/UX:** Improve the overall look and feel, add loading states, better form validation feedback.
* **Dockerization:** Containerize the application using Docker for easier deployment.
* **Testing:** Add unit, integration, and end-to-end tests.
* **Security Audits:** Conduct thorough security testing.

12. Conclusion

The MERN stack Doctor Appointment Booking System successfully provides a foundation for managing appointments with role-based access control. By implementing secure authentication, handling doctor applications, and separating user, doctor, and admin functionalities, it addresses the core requirements of an online booking platform. While there is scope for further enhancements to add more advanced features and polish the user experience, the current implementation demonstrates proficiency in full-stack web development using the MERN stack.