**FULL STACK DEVELOPMENT WITH MERN**

**MERN STACK BY MONGODB**

**PROJECT:**

**MEDICAREBOOK : BOOK A DOCTOR USING MERN STACK**

**Team Members**

**ISHA R (au311521104019)**

**MEGHANA G (au311521104027)**

**RANGAPRIYA R (au311521104037)**

**SHWETHA C (au311521104056)**

# 1. INTRODUCTION

* **Project Title:** Book A Doctor using MERN Stack
* **Team Members:**

Isha R - Project Coordinator

Meghana G - Frontend Developer

Rangapriya R - Backend Developer

Shwetha C - Database Manager

# 2. PROJECT OVERVIEW

MediCareBook is a robust web application designed to enhance the healthcare experience by facilitating seamless communication between patients and doctors. The platform allows users to register, manage their profiles, and book appointments with healthcare professionals, while providing doctors with the tools to manage their profiles and appointments effectively. The platform allows qualified doctors to apply for registration, enabling them to create profiles that showcase their specialties, experience, and availability. Users can book, view, and manage their appointments. Doctors can also view their scheduled appointments and update their status (e.g., approved, pending). The application also includes an administrative interface to oversee user and doctor registrations, ensuring efficient management of the platform. Users can securely register, log in, and manage their profiles, with session handling enabled by JWT tokens and bcrypt-secure password hashing.

The MongoDB schema for MediCareBook is designed to support a robust and scalable system , with collections for users, doctors, appointments. The app provides role-specific dashboards and robust functionality that enable users to find doctors, book appointments, and track consultation history, while allowing doctors to manage their availability and administrators to oversee the platform’s operations.

## Purpose:

The primary purpose of the MediCareBook application is to create a secure, efficient, and user-friendly environment for patients and healthcare providers to interact. It aims to streamline the appointment booking process, enhance communication, and ensure that both users and administrators have the necessary tools to manage their interactions effectively.

* **Enhancing Patient Experience:**

MediCareBook offers patients a straightforward platform where they can easily find and schedule appointments with doctors, manage their health information, and receive timely notifications about their appointments and health-related updates.

* **Empowering Doctors:**

The platform provides doctors with a dedicated space to manage their profiles, view appointments, and communicate with patients, ensuring they can focus on providing quality healthcare.

* **Secure Transactions:**

MediCareBook employs advanced security measures, including JWT authentication and password hashing, to ensure that user data is protected and that logins are secure. This includes session handling to maintain user login status across multiple sessions.

* **Comprehensive Doctor Profiles:**

The platform allows users to explore a variety of doctors across different specialties, with detailed profiles that include qualifications, experience, fees, and availability. This helps patients make informed decisions when selecting healthcare providers.

* **Convenient Appointment Management:**

Users can add appointments to their calendars, manage their bookings, and receive notifications, ensuring a personalized and efficient healthcare experience.

* **Diverse Healthcare Options:**

By offering a wide range of healthcare professionals and specialties, MediCareBook caters to diverse patient needs, allowing users to find the right doctor for their specific health concerns.

## Features:

1. **User Authentication and Management:**

MediCareBook provides a secure and streamlined authentication process for both patients and doctors. Users can easily register and log in to their accounts, ensuring that their personal information is protected through robust security measures. The application utilizes JSON Web Tokens (JWT) for session management, which enhances the security of user sessions. Additionally, passwords are securely hashed to safeguard user credentials. Users can manage their profiles, allowing them to update personal information such as contact details and preferences.

1. **Doctor Registration and Profile Management:**

The application allows doctors to apply for registration and create comprehensive profiles that showcase their qualifications and specialties. Each doctor’s profile includes essential information such as their area of expertise, years of experience, consultation fees, and contact details. This feature empowers doctors to manage their availability and appointment slots effectively, ensuring that patients can easily find and book consultations with them.

1. **Appointment Booking:**

MediCareBook simplifies the appointment booking process, enabling users to search for doctors based on various criteria, including specialization, location, and availability. The user-friendly interface allows patients to book appointments with just a few clicks. Additionally, users have the option to upload necessary documents related to their appointments, facilitating better preparation for consultations. Once booked, users can view and manage their upcoming and past appointments seamlessly.

1. **Filtering Functionality:**

To enhance the user experience, MediCareBook incorporates comprehensive filtering options that allow users to narrow down their search for doctors. Users can filter results based on specialization, location, experience, and fees, ensuring they find the most suitable healthcare provider for their needs. Furthermore, users can sort search results by parameters such as rating, availability, and distance, making the search process efficient and tailored to individual preferences. A clear filters option is also available, allowing users to reset their searches easily.

1. **Search Functionality:**

The application features an intuitive search bar that enables users to find doctors by entering keywords related to specialties or names. This quick access to relevant profiles based on search queries enhances the overall usability of the platform, allowing users to locate the healthcare professionals they need without hassle.

1. **Notification System:**

MediCareBook includes a real-time notification system that keeps users informed about their appointments and any relevant updates. Patients receive notifications regarding appointment confirmations, reminders, and changes, while doctors are notified about new appointments and modifications to their schedules. This feature ensures that all parties are kept in the loop, reducing the chances of missed appointments.

1. **User Dashboard:**

The user dashboard serves as a central hub for patients, allowing them to manage their profiles, view appointment history, and access notifications. This user-friendly interface provides an overview of upcoming appointments, past visits, and any pending actions, making it easy for users to stay organized and informed about their healthcare journeys.

1. **Doctor Dashboard:**

Doctors have access to a dedicated dashboard that enables them to manage their profiles, view appointment requests, and communicate with patients effectively. This dashboard provides an overview of scheduled appointments and patient interactions, allowing doctors to maintain a clear and organized workflow.

1. **Admin Dashboard:**

The administrative dashboard offers a centralized control hub for administrators to manage users, doctors, and appointments efficiently. Admins can approve or reject doctor registrations, monitor appointment statuses, and access reporting features that analyze user engagement and appointment trends. This oversight ensures that the platform operates smoothly and maintains a high standard of service.

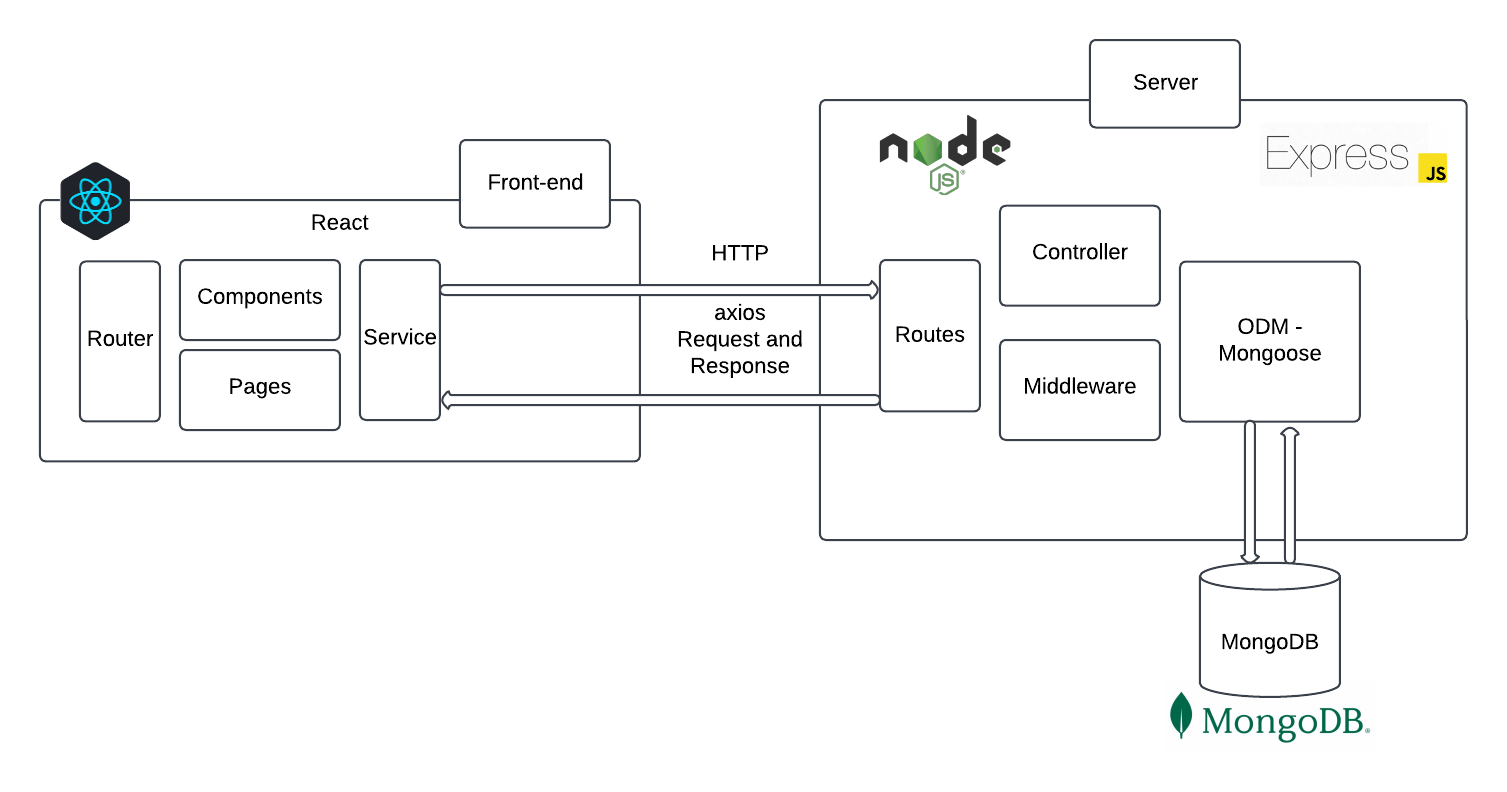
1. **Appointment History:**

Users can view detailed records of their past and upcoming appointments, including status and doctor information. This feature allows patients to easily track their health visits and maintain a comprehensive history of their interactions with healthcare providers.

1. **Secure Communication:**

MediCareBook prioritizes secure communication between patients and doctors. The application includes messaging features that allow users to communicate regarding appointments and health queries while ensuring privacy and protection of sensitive information.

# 3. ARCHITECTURE

****

*Figure 1.Architecture*

## Frontend:

The frontend is structured as a single-page application (SPA) using React. This architecture aims to deliver an interactive user experience, where various components dynamically load and update content without needing full page reloads. The frontend follows a component-based structure, where each component represents a specific part of the application, such as user profiles, appointment booking forms, and doctor listings. The frontend communicates with the backend through RESTful APIs, enabling seamless data exchange. Additionally, libraries such as Ant Design and Bootstrap are utilized for styling and layout, ensuring a modern and user-friendly interface.

1. **Component Structure:**

Functional Components: All React components are structured as functional components for better performance and use of React hooks.

* + Common Components:
    - Home: The landing page that introduces the application and its features.
    - Login: The login form for users to access their accounts.
    - Register: The registration form for new users to create accounts.
    - Notification: Displays notifications for users and doctors.
  + User Components:
    - User Home: The dashboard for users, providing access to appointments and doctor listings.
    - User Appointments: A component that shows all the user's appointments and their statuses.
    - ApplyDoctor: A form for users to apply for registration as doctors.
    - DoctorList: Displays a list of available doctors for users to book appointments.
  + Admin Components:
    - AdminHome: The dashboard for administrators to manage users and appointments.
    - AdminUsers: Lists all users with options to filter and manage their accounts.
    - AdminDoctors: Lists all doctors with options to approve or reject their registrations.
    - AdminAppointments: Displays all appointments with filtering capabilities

1. **State Management:**

State management in MediCareBook is handled using React's built-in useState and useEffect hooks for local component state. For more complex state management that involves multiple components, the application can utilize the Context API or third-party libraries like Redux (if needed). This allows for a global state that can be accessed across various components, facilitating data sharing and synchronization. Context API, combined with custom hooks and reducers, is used for managing global state (e.g., user authentication). Session storage is used to store JWT tokens and user data to manage sessions across different tabs. Tokens are verified before making requests.

1. **Routing:**

React Router: react-router-dom is used for client-side routing, with different routes for each main page and protected routes for user-specific and admin-specific pages.

The application uses React Router for handling navigation and routing between different components. The routing structure allows users to navigate seamlessly through the application. Key routes include:

* /: Home page
* /login: Login page
* /register: Registration page
* /userhome: User dashboard
* /adminhome: Admin dashboard
* /userhome/userappointments/:doctorId: Specific appointment details page

The Routes component from React Router is used to define the different routes and their corresponding components, ensuring that users are directed to the correct view based on their actions.

1. **API Communication:**

Axios is used for HTTP requests to communicate with the backend API. The JWT token is included in the header of each request to ensure secure access to protected routes. Centralized error handling with feedback for users, such as login failure, invalid inputs, or out-of-stock items.The application sends requests to various endpoints to perform actions such as user authentication, fetching doctor lists, and managing appointments. For example:

* Login: Sends a POST request to authenticate the user and retrieve a token.
* Fetch Doctors: Sends a GET request to retrieve a list of available doctors for booking appointments.
* Book Appointment: Sends a POST request with appointment details to schedule a new appointment.

Axios is configured to include the necessary headers, such as authorization tokens, to ensure secure communication with the backend.

1. **Styling and UI:**

Styled components or inline styles are used for modular and scalable CSS. CSS media queries are used to ensure responsiveness across devices, particularly for product lists and the cart.

## Backend:

The backend is built with Node.js and Express.js, providing a RESTful API for the frontend and managing user authentication.

1. **Server Structure:**

The main application is built using Express which initializes all middleware, routes, and database connections. Environment Configuration is done using dotenv and is used to manage environment variables like database URI, JWT secret, and server port. The server structure is organized into several key components:

* **Entry Point:** The main entry point of the application is defined in the index.js file, where the Express application is initialized, and middleware is set up.
* **Middleware:** Middleware functions are used for various purposes, including logging requests, handling errors, and managing authentication. A custom authentication middleware is implemented to verify user tokens and protect certain routes.
* **Controllers:** Controllers are organized into separate files based on functionality. Each controller handles the business logic for specific features, such as user registration, login, appointment management, and doctor registration. This separation of concerns improves maintainability and readability.
* **Routes:** Routes are defined in separate files for users, doctors, and admin functionalities. Each route file maps HTTP requests to the corresponding controller functions, ensuring a clear and organized routing structure.
* **Schemas:** Mongoose schemas are defined for users, doctors, appointments, and notifications. These schemas define the structure of the data stored in the MongoDB database and enforce validation rules.

1. **API Design:**

The API of MediCareBook follows RESTful principles, allowing for clear and consistent interaction between the frontend and backend. Key design principles include:

* **Resource-Based Endpoints:**
  + Each resource (e.g., users, doctors, appointments) has its own set of endpoints, typically following the format:
    - GET /api/user/: Retrieve all users
    - POST /api/user/register: Register a new user
    - POST /api/user/login: Authenticate a user
    - GET /api/doctor/getalldoctors: Retrieve all doctors
    - POST /api/user/getappointment: Book an appointment
* **HTTP Methods:**
  + The application utilizes standard HTTP methods to perform CRUD operations:
    - GET: Retrieve data
    - POST: Create new resources
    - PUT: Update existing resources
    - DELETE: Remove resources
* **Status Codes:**
  + The API responds with appropriate HTTP status codes to indicate the result of each request, such as:
    - 200 OK: Successful request
    - 201 Created: Resource successfully created
    - 400 Bad Request: Invalid request data
    - 401 Unauthorized: Authentication failure
    - 404 Not Found: Resource not found
    - 500 Internal Server Error: Server error

1. **Authentication and Authorization:**
   * JSON Web Tokens are used for secure access, generated on login and verified on protected routes. bcrypt is used to hash user passwords before storing them in the database.
   * Role-Based Access Control functions are used to control access for Customer and Admin roles, allowing only admins to access certain endpoints.
2. **Session and Token Handling:**

Tokens have an expiry time, after which users need to log in again. The token, user ID, and user type are stored in the frontend session storage for identifying the logged-in user across pages.

## Database:

The database uses MongoDB with a schema defined through Mongoose. Collections are created to manage Users, Doctors, Appointments, and Notification ensuring data integrity and efficient querying.

**Database Structure:**

**i) User Schema:**

* **Fields:** fullName, email, password, phone, type, notification, seennotification, isDoctor
* **Indexes:** 
  + Unique index on the email field to ensure no duplicate email addresses.
  + Index on the type field to facilitate efficient querying based on user roles.
* **Relationships:** 
  + Doctor Schema: One-to-one relationship with the Doctor schema (a user can be a doctor).
  + Appointment Schema: One-to-many relationship with the Appointment schema (a user can have multiple appointments).

**ii) Doctor Schema:**

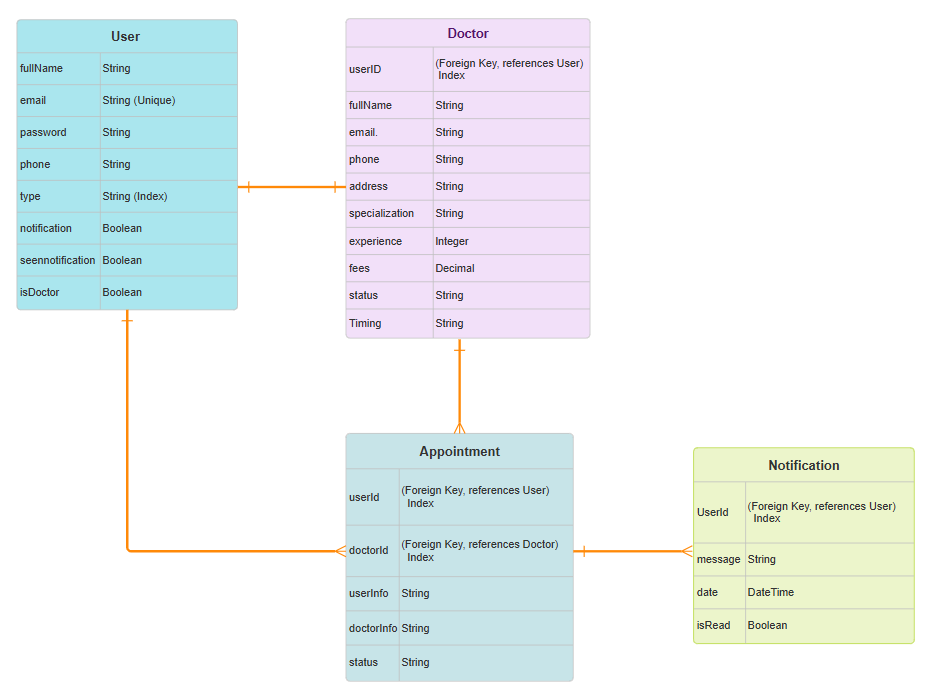
* **Fields:** userId, fullName, email, phone, address, specialization, experience, fees, status, timings
* **Indexes:**
  + Index on the userId field to quickly find the doctor associated with a user.

**iii) Appointment Schema:**

* **Fields:**  userId, doctorId, userInfo, doctorInfo, status.
* **Indexes:**
  + Index on the userId field to quickly find appointments associated with a user.
  + Index on the doctorId field to efficiently query appointments for a specific doctor.

**iv) Notification Schema:**

* **Fields:** userId, message, date, and isRead.



*Figure 2.ERD Diagram*

## Database Operations:

The database operations in MediCareBook are primarily handled using Mongoose, which provides an elegant way to interact with MongoDB through models and schemas. The application includes CRUD operations for each schema, such as creating, reading, updating, and deleting user, doctor, appointment, and notification records. Password hashing with bcrypt is used for secure authentication during user registration and login. Indexes are carefully defined for efficient querying, and relationships between schemas are managed through references and populating methods. The application can efficiently handle user appointment bookings, doctor registrations, and notification management using these database operations. Once logged in, users can access a personal dashboard where they can view, cancel, reschedule appointments, and upload necessary medical documents. The dashboard provides a clear overview of all future and past appointments. Doctors can manage their schedules, accept or decline appointment requests, update their profiles (including fees, experience, and availability), and view patient records.

# 4. SETUP INSTRUCTIONS

## PRE-REQUISITES:

* **Node.js (Version 20.x):** Download and install from Node.js official website, which includes npm (Node Package Manager).
  + Verify the installation by running:
    - node -v
    - npm -v
* **MongoDB:** Set up MongoDB using MongoDB Compass for a local instance or create a MongoDB Atlas account for a cloud-based setup.
* **Git:** Install Git from Git downloads for version control.
* **npm -** Node Package Manager (comes with Node.js).
* **Code Editor -** Visual Studio Code or another preferred IDE.

## INSTALLATION:

Follow these steps to set up the project on your local machine:

1. **Download Project Files:**

Place all project files in a dedicated project directory on the local machine.

1. **Install backend dependencies:**
   * Navigate to the backend directory:
     + - **cd backend**
   * Install the required Node.js packages:
     + - **npm install**
2. **Install frontend dependencies:**
   * Navigate to the frontend directory:
     + - **cd ../frontend**
   * Install the required Node.js packages:
     + - **npm install**
3. **Set Up Environment Variables:**

In the project root, create a .env file to store environment-specific variables such as database connection strings and JWT secrets. Contents in the .env file is as follows:

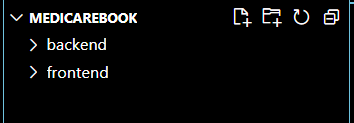
* + **PORT=8001**
  + **MONGO\_URI=<mongodb-connection-string>**
  + **JWT\_SECRET=<jwt-secret>**

1. **Run the Application:**
   * Start the backend server:
     + - **cd ../backend**
       - **npm start**
   * In a new terminal window, start the frontend application:
     + - **cd frontend**
       - **npm start**
2. **Access the application:**

Open your web browser and navigate to http://localhost:3000 to access the MediCareBook application.

# 5. FOLDER STRUCTURE

The MediCareBook : Book A Doctor Application follows a well-organized folder structure for both the **Frontend** (React) and **Backend** (Node.js) to ensure clarity, scalability, and maintainability of the project.

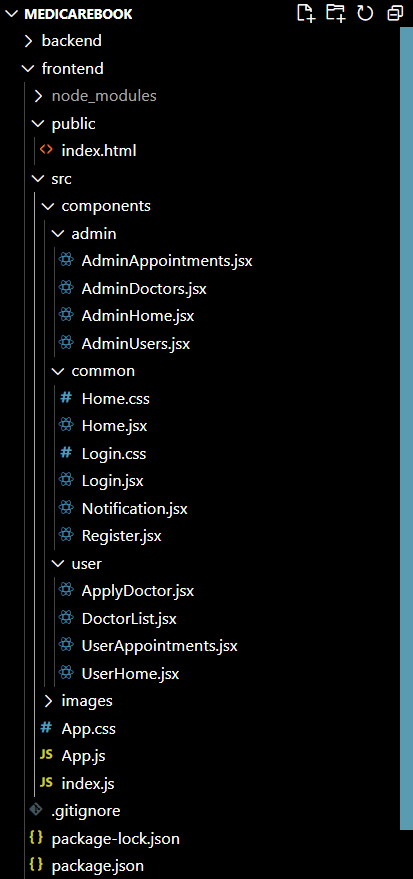
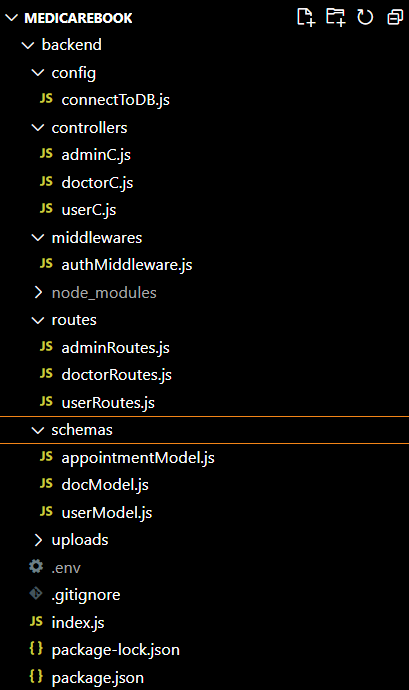


*Figure 3.Overall Folder Structure*

## Frontend : React Frontend Structure

The frontend of MediCareBook is built using **React**, and its folder structure is organized as follows:

* **public/**:
  + Contains static files that are directly served to the client, including the main HTML file and any images or icons.
* **src/**:
  + This is where all the source code of the React application resides.This is where all the JavaScript and React components are located, along with styles and images.
  + **components/**: This folder contains all the React components that make up the user interface of the application. Components are reusable pieces of code that define how a certain part of the UI should look and behave.
  + **Subdirectories:**
    - **admin/:** Contains components that are specific to the admin functionalities of the application. These components are designed for users with administrative roles and include features for managing users, doctors, and appointments.
    - **user/:** Contains components that are specific to regular user functionalities. These components are designed for standard users who can book appointments, view doctors, and manage their profiles.
    - **common/:** Contains components that are shared across the application. These components are not specific to either admin or user roles and can be used by both.
* **App.js :**
  + This is the main application component that serves as the entry point for the React application. It sets up the routing for the application using react-router-dom and defines the overall layout and structure of the application.
* **index.js :** 
  + This is the entry point of the React application. It renders the App component into the DOM. This file typically includes the necessary imports for React and ReactDOM
* **App.css :** 
  + This file contains the CSS styles for the application. It is used to style components and layout globally. This file can include styles for typography, colors, spacing, and responsive design elements.
* **images/ :** 
  + This folder contains image assets that are used throughout the application.
* **package.json**:
  + This file manages project dependencies, scripts, and other configurations for the frontend application.

*Figure 3.1 Client Structure Figure 3.2.Server Structure*

## Backend : Node.js Backend Structure

The backend is built using **Node.js** and follows a simple but effective folder structure for API development:

#### backend/ : This is the root directory for the backend application of the MediCareBook project. It contains all the files and folders necessary for building the server-side functionality of the application.

#### config/ : This folder contains configuration files for the backend application. connectToDB.js file contains the logic for connecting to the MongoDB database using Mongoose. It handles the connection setup and error handling.

#### controllers/ : This folder contains controller files that define the logic for handling requests and responses for different functionalities.

* **SubDirectories Files:**
  + **adminC.js:** Contains functions for handling admin-related operations, such as managing users and appointments.
  + **doctorC.js:** Contains functions for handling doctor-related operations, such as updating profiles and managing appointments.
  + **userC.js:** Contains functions for handling user-related operations, such as registration, login, and managing appointments.
* **middlewares/ :** This folder contains middleware functions that can be used in the request handling process. authMiddleware.js checks for authentication tokens in requests, verifies them, and ensures that only authenticated users can access certain routes.
* **routes/ :** This folder contains route definitions for the application. Each file typically corresponds to a specific resource or functionality. Files:
  + **adminRoutes.js:** Defines the routes for admin-related API endpoints, such as getting all users or approving doctor registrations.
  + **doctorRoutes.js:** Defines the routes for doctor-related API endpoints, such as updating doctor profiles and fetching appointments.
  + **userRoutes.js:** Defines the routes for user-related API endpoints, such as user registration and login.
* **schemas/ :** This folder contains Mongoose schema definitions for different data models used in the application. Files:
  + **appointmentModel.js:** Defines the schema for appointment documents in the database.
  + **docModel.js:** Defines the schema for doctor documents in the database.
  + **userModel.js:** Defines the schema for user documents in the database.
* **index.js :** This is the main entry point for the backend application. It sets up the Express server, connects to the database, configures middleware, and defines the API routes. It listens for incoming requests on a specified port.
* **.env :**  This file contains environment variables used in the application, such as database connection strings, JWT secret keys, and other sensitive information. This file is typically not included in version control for security reasons.

# 6. RUNNING THE APPLICATION

To run the MediCareBook : Book A Doctor Application locally, the following steps are done for starting both the frontend and backend servers:

## Frontend:

**1. Navigate to the Client Directory:**

Open a terminal window on your system.

Use the cd (change directory) command to navigate to the frontend folder, which contains all the frontend code for the application.

*cd C:\MediCareBook\frontend*

**2. Install Frontend Dependencies:**

Before starting the frontend server, ensure that all required dependencies are installed. If this is the first time running the project or if dependencies have been added or updated, run the following command to install them:

npm install

This command will install all the necessary packages listed in the package.json file, including React and other dependencies.

**3. Start the Frontend Server:**

After installing the dependencies, use the following command to start the React development server:

npm start

This will compile the React application and start a local development server.

**4. Access the Application:**

* Once the server is running, a web browser is opened and goes to http://localhost:3000.
* The frontend application will be accessible at this URL, and you should see the MediCareBook landing page in your browser. The frontend is now live and connected to the backend server for full functionality.

## Backend:

**1. Navigate to the Server Directory:**

Open a terminal window.

Use the cd (change directory) command to navigate to the server folder, which contains all the backend code for the application.

*cd C:\MediCareBook\backend*

**2. Install Backend Dependencies:**

If this is the first time running the backend server, ensure all required dependencies are installed by running:

npm install

This will install all the necessary backend packages, including Express, Mongoose, and other dependencies as specified in the package.json file.

**3. Start the Backend Server:**

Once the dependencies are installed, use the following command to start the backend server:

npm start

This will initialize the backend server and connect it to the database.

**4. Access the Backend:**

* The backend server will now be running and can be accessed via http://localhost:3000.
* This URL is used to handle all API requests, and the backend will process interactions such as user authentication, product management, and order processing.

# 7. API DOCUMENTATION

The MediCareBook : Book A Doctor Application backend exposes various endpoints to handle operations related to users, doctors, admin, notification and authentication.

## Authentication

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

{

"username": "JohnDoe",

"email": "johndoe@example.com",

"password": "password123",

"usertype": "user"

}

* + **Response:**
    - * **200 OK:** User registered successfully.

{

"message": "User registration successful.",

"user": {

"id": "12345",

"username": "john\_doe",

"email": "john@example.com"

} }

* + - **400 Bad Request:** Missing or invalid parameters.

{ "error": "Missing or invalid fields" }

* **POST /api/user/login**
  + **Description:** Logs in a user and generates a JWT token.
  + **Request Body:**

{ "email": "john@example.com",

"password": "password123" }

* + **Response:**
    - **200 OK:** User Login successfully.

{ "message": "Login successful.",

"token": "jwt-token" }

* + - **400 Unauthorized:** Missing or invalid parameters.

{ "error": "Invalid credentials." }

## User Management

* **POST /api/user/getuserdata**
  + **Description:** Retrieve user data based on the authorization token.
  + **Request Parameters:** None (requires authorization).

Request Header : { "Authorization": "Bearer <JWT\_TOKEN>"}

* + **Response:**
    - **200 OK:** Returns user data.

[ {

"id": "123",

"name": "John Doe",

"email": "johndoe@example.com"

}]

* + - **401 Unauthorized:** Authorization token missing or invalid.

{ "error": "Unauthorized access" }

* **POST /api/user/registerdoc**
  + **Description:** Register a new doctor.
  + **Request Body:**

{

"name": "Dr. John Doe",

"email": "doctor@example.com",

"specialization": "Cardiologist"

}

* + **Response:**
    - **200 OK:** Doctor registered successfully.
    - **400 Bad Request:** Missing or invalid parameters.
* **GET /api/user/getalldoctorsu**
  + **Description:** Retrieve a list of all doctors for users.
  + **Request Parameters:** None (requires authorization).
  + **Response:**
    - **200 OK:** Returns a list of doctors.
    - **401 Unauthorized:** Authorization token missing or invalid.

## Admin Management

* **GET /api/admin/getallusers**
  + **Description:** Retrieve a list of all registered users.
  + **Request Parameters:** None (requires authorization).

Request Header : { "Authorization": "Bearer <JWT\_TOKEN>"}

* + **Response:**
    - **200 OK:** Returns user data.

[ {

"id": "123",

"name": "John Doe",

"email": "johndoe@example.com"

}]

* + - **401 Unauthorized:** Authorization token missing or invalid.

{ "error": "Unauthorized access" }

* **GET /api/admin/getalldoctors**
  + **Description:** Retrieve a list of all registered doctors.
  + **Request Parameters:** None (requires authorization).
    - Headers: { "Authorization": "Bearer <JWT\_TOKEN>"}
  + **Response:**
    - **200 OK:** Returns a list of doctors.

[ { "id": "101",

"name": "Dr. Alice",

"specialization": "Cardiology"

}, {

"id": "102",

"name": "Dr. Bob",

"specialization": "Dermatology"

}]

* + - **401 Unauthorized:** Authorization token missing or invalid.
* **POST /api/admin/getapprove**
  + **Description:** Approve a doctor's registration or request.
  + **Request Body:**

{ "doctorId": "12345"}

* + **Response:**
    - **200 OK:** Doctor approved successfully.

{ "message": "Doctor approved successfully"}

* + - **400 Bad Request:** Missing or invalid parameters.

{ "error": "Missing or invalid doctorId"}

* + - **401 Unauthorized:** Authorization token missing or invalid.
* **POST /api/admin/getreject**
  + **Description:** Reject a doctor's registration or request.
  + **Request Body:**

{ "doctorId": "12345"}

* + **Response:**
    - **200 OK:** Doctor rejected successfully.

{ "message": "Doctor rejected successfully"}

* + - **400 Bad Request:** Missing or invalid parameters.

{ "error": "Missing or invalid doctorId"}

* + - **401 Unauthorized:** Authorization token missing or invalid.
* **GET /api/admin/getallAppointmentsAdmin**
  + **Description:** Retrieve all appointments for the admin.
  + **Request Parameters:** None (requires authorization).

Headers: { "Authorization": "Bearer <JWT\_TOKEN>"}

* + **Response:**
    - **200 OK:** Returns a list of all appointments.

[ {

"appointmentId": "501",

"user": "John Doe",

"doctor": "Dr. Alice",

"date": "2024-11-20",

"status": "Pending"

} ]

* + - **401 Unauthorized:** Authorization token missing or invalid.

## Doctor Management

* **POST /api/doctor/updateprofile**
  + **Description:** Update the profile details of a doctor.
  + **Request Body:**

{ "name": "Dr. John Doe",

"specialization": "Cardiologist",

"experience": "10 years"}

* + **Response:**
    - **200 OK:** Profile updated successfully.
    - **400 Bad Request:** Missing or invalid parameters.
    - **401 Unauthorized:** Authorization token missing or invalid.
* **GET /api/doctor/getdoctorappointments:**
  + **Description:** Retrieve a list of appointments for a specific doctor.
  + **Request Parameters:** None (requires authorization).
  + **Response:**
    - **200 OK:** Returns a list of appointments.

[ {

"appointmentId": "601",

"user": "Jane Smith",

"date": "2024-11-21",

"status": "Approved"

} ]

* + - **401 Unauthorized:** Authorization token missing or invalid.
* **POST /api/doctor/handlestatus**
  + **Description:** Handles appointment status updates by the doctor.
  + **Request Body:**

{ "appointmentId": "12345",

"status": "Approved"}

* + **Response:**
    - **200 OK:** Appointment status updated successfully.
    - **400 Bad Request:** Missing or invalid parameters.
    - **401 Unauthorized:** Authorization token missing or invalid.
* **GET /api/doctor/getdocumentdownload**
  + **Description:** Download a document associated with a doctor's appointment or profile.
  + **Request Parameters:** None (requires authorization).
  + **Response:**
    - **200 OK:** Returns the requested document.
    - **401 Unauthorized:** Authorization token missing or invalid.

## Notification Management

* **POST /api/user/getallnotification**
  + **Description:** Retrieve all notifications for the user.
  + **Request Parameters:** None (requires authorization).
  + **Response:**
    - **200 OK:** Returns a list of notifications.

[ { "id": "201",

"message": "Your appointment with Dr. Alice has been confirmed.",

"status": "Unread"

}]

* + - **401 Unauthorized:** Authorization token missing or invalid.
* **POST /api/user/deleteallnotification**
  + **Description:** Delete all notifications for the user.
  + **Request Parameters:** None (requires authorization).
  + **Response:**
    - **200 OK:** Notifications deleted successfully.
    - **401 Unauthorized:** Authorization token missing or invalid.

## Appointment Management

* **POST /api/user/getappointment**
  + **Description:** Book an appointment with a doctor.
  + **Request Body:**

{ "doctorId": "12345",

"date": "2024-11-18",

"time": "10:00 AM",

"description": "Regular check-up" }

* + **Response:**
    - **200 OK:** Appointment booked successfully.
    - **400 Bad Request:** Missing or invalid parameters.
* **GET /api/user/getuserappointments**
  + **Description:** Retrieve a list of all appointments booked by the user.
  + **Request Parameters:** None (requires authorization).
  + **Response:** 
    - **200 OK:** Returns a list of appointments.
    - **401 Unauthorized:** Authorization token missing or invalid.

## Document Management

* **GET /api/user/getDocsforuser**
  + **Description:** Retrieves documents available for the user to download.
  + **Request Parameters:** None (requires authorization).
  + **Response:**
    - **200 OK:** Returns a list of documents.

[ {

"docId": "301",

"title": "Blood Test Report",

"link": "/uploads/blood-test-report.pdf"

}]

* + - **401 Unauthorized:** Authorization token missing or invalid.

# 8. AUTHENTICATION

## Authentication:

Authentication is essential for securing the Book a Doctor application and controlling access to protected resources. This project uses JSON Web Tokens (JWT) for authentication, enabling secure, scalable access across the frontend and backend.The authentication and authorization process is designed to securely verify user identity and manage access control using JSON Web Tokens (JWT), session storage, and password hashing.

1. **User Authentication with JWT Tokens**:

* Upon successful login or registration, a JSON Web Token (JWT) is generated and returned to the client. The token is encoded with user-specific details and a secret key to ensure authenticity.
* The registration endpoint is defined in the userC.js controller, specifically in the registerController function.
* This token is stored in the client’s session storage and is included in headers for all protected requests, allowing the server to verify user identity and permissions.

1. **Token-Based Authorization**:

* JWT tokens are validated on each request to ensure that only authenticated users can access restricted features. For instance, customers have access to profile and shopping pages, while admin users have access to the Admin Dashboard.
* The token includes encoded user information (such as user ID and role), enabling role-based authorization and ensuring access control across different parts of the application.

1. **Password Hashing**:

* Passwords are securely hashed with bcrypt before being stored in the database, adding an essential layer of security to protect user credentials.
* During login, passwords are rehashed and verified against the stored hash to authenticate users.

## Authentication Process

Authentication is the process of verifying the identity of a user. In this project:

**Steps:**

1. **User Login:** A user submits their credentials (email and password) to the /api/user/login endpoint.
2. **Verification:**
   1. The backend checks the credentials against the database.
   2. If valid, the backend generates a JWT, signed using a secret key stored in the JWT\_KEY environment variable.
3. **Token Generation:**
   1. The token contains a payload with user-specific data (e.g., id), which is encoded and signed.
   2. The generated JWT is sent to the client in the response.
4. **Client Storage:** The client (frontend) stores the token, usually in localStorage or sessionStorage, or sets it as a cookie for subsequent use.

## 

## Authorization :

* **Middleware for Protected Routes:**
  + An authentication middleware (authMiddleware.js) is implemented to protect certain routes. This middleware checks for the presence of the JWT in the authorization header of incoming requests.
  + If the token is missing or invalid, the middleware responds with an error, denying access to the requested resource. If the token is valid, the middleware decodes the token, retrieves the user ID from it, and attaches it to the request object (req.body.userId), allowing subsequent handlers to access the user's information.
* **Role-Based Access Control:**
  + The project supports different user roles (e.g., admin, user, doctor). Based on the user's role, different routes and functionalities are accessible. After logging in, the user's role is checked, and they are redirected to the appropriate home page (admin or user).

## Authorization Process

Authorization determines what actions or resources a user can access. The JWT is used to authorize user requests to protected endpoints.

**Steps:**

1. Token Inclusion: For every request to a protected route, the client includes the JWT in the Authorization header in the format:

Authorization: Bearer <token>

1. Token Validation:
   1. The backend middleware (authMiddleware) extracts the token and validates it using the jwt.verify function.
   2. If the token is valid:
      * The middleware attaches the decoded payload (e.g., user ID) to the req.body for downstream use.
      * The request is allowed to proceed.
   3. If invalid:
      * The middleware returns an error response (e.g., 401 Unauthorized).

## Details of the Token

**JWT Structure:** A JWT consists of three parts:

* Header:
  + Specifies the algorithm and token type.
  + Example:

{ "alg": "HS256",

"typ": "JWT"}

* Payload:
  + Contains claims (information about the user, such as id or role).
  + Example:

{ "id": "user123",

"role": "admin",

"exp": 1700000000}

* Signature: Ensures the token's integrity and authenticity and created using the header, payload, and secret key.

## Security Measures

1. **Environment Variables:** The secret key used for signing (JWT\_KEY) is stored securely in environment variables to avoid hardcoding sensitive data.
2. Token Expiry:
   1. JWTs are configured with an expiration time (exp) to limit their validity.
   2. After expiration, the user must log in again to obtain a new token.
3. HTTPS: All requests must be made over HTTPS to ensure token security during transmission.
4. Middleware Validation: The middleware ensures that only authenticated and authorized users can access protected resources.

## Example Workflow

1. **Login Request**:

**Endpoint:** POST /api/user/login

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

"password": "password123"}

**Response:**{ "message": "Login successful",

"token": "<jwt-token>"}

1. **Accessing a Protected Route:**

**Endpoint:** GET /api/user/getuserappointments

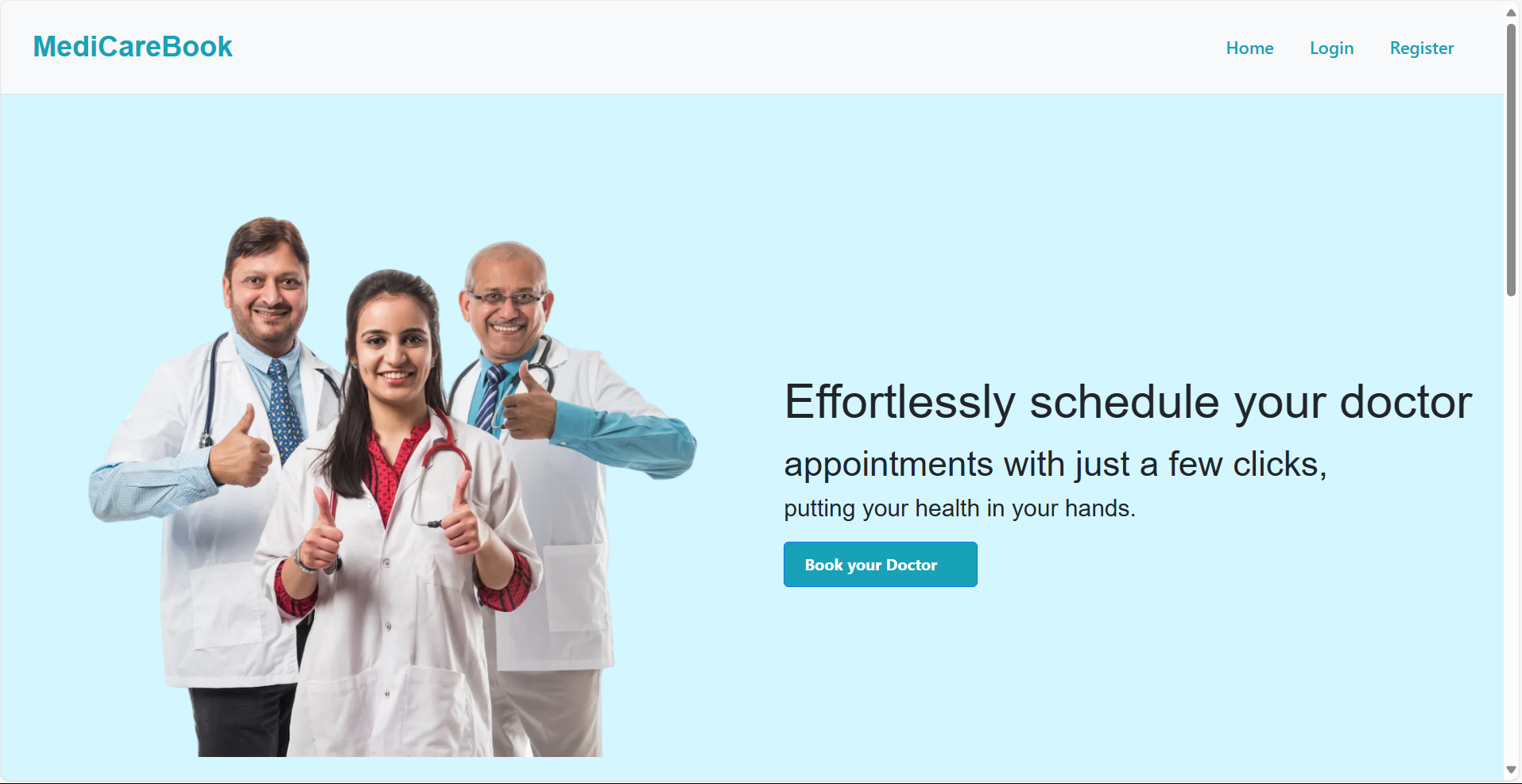
**Request Headers:** { "Authorization": "Bearer <jwt-token>"}

**Response (Valid Token):** {"appointments": [...] }

# 9. USER INTERFACE

The **User Interface (UI)** is designed to provide an intuitive and engaging experience for all users . Built using **React**, the UI ensures responsiveness, smooth interactions

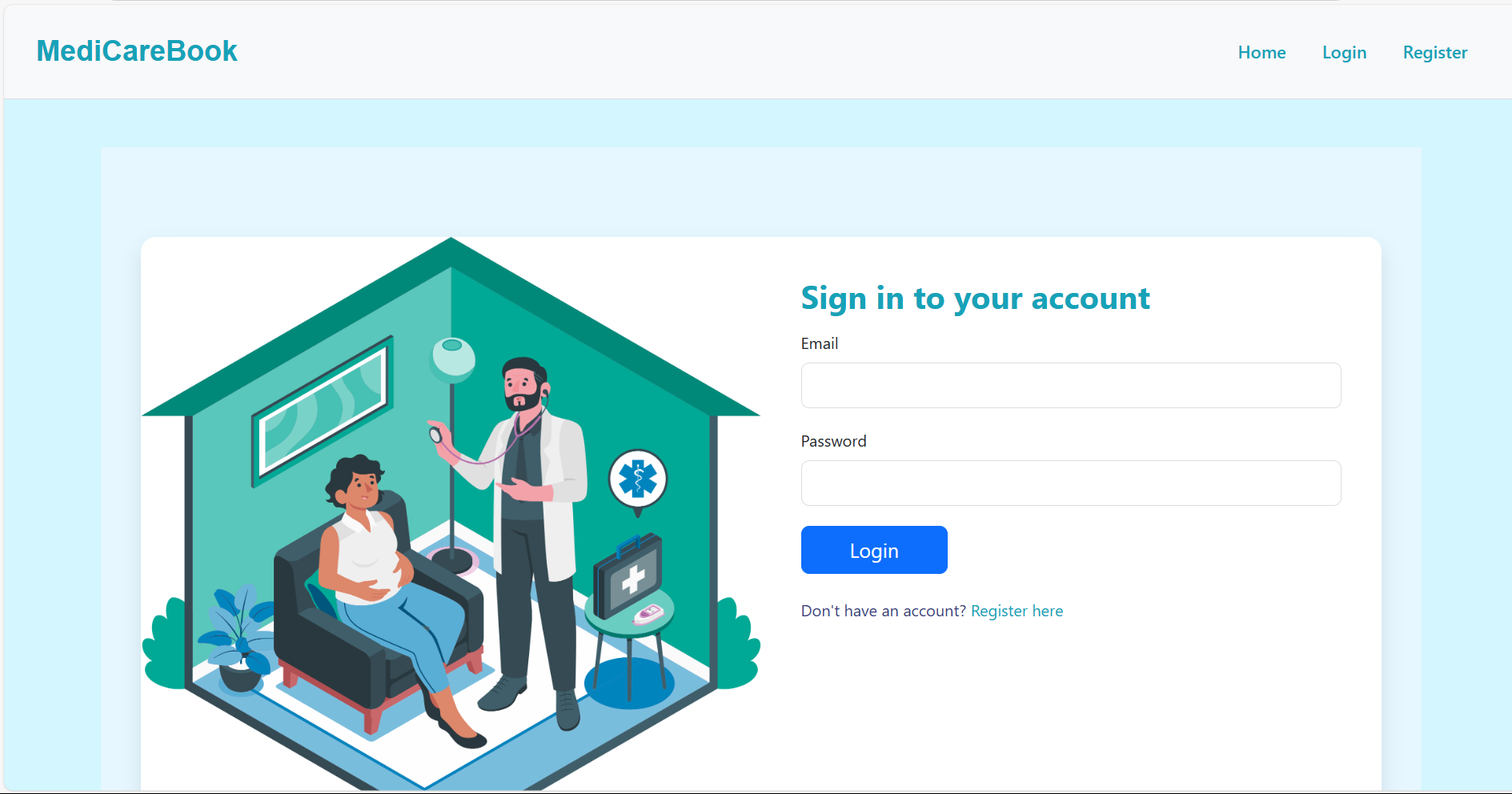
## 1. Home Page :

****

The MediCareBook UI is designed for simplicity and easy navigation. The Homepage serves as the initial entry point for users, providing a welcoming and informative introduction to the Book a Doctor platform. Key features include:

* **User-Friendly Navigation:**
  + Clear options for Login and Registration for customers and doctors.
  + Quick access links to browse available doctors and specialties.
* **Call-to-Action (CTA) Buttons:**
  + Prominent CTAs encouraging users to register or book an appointment.

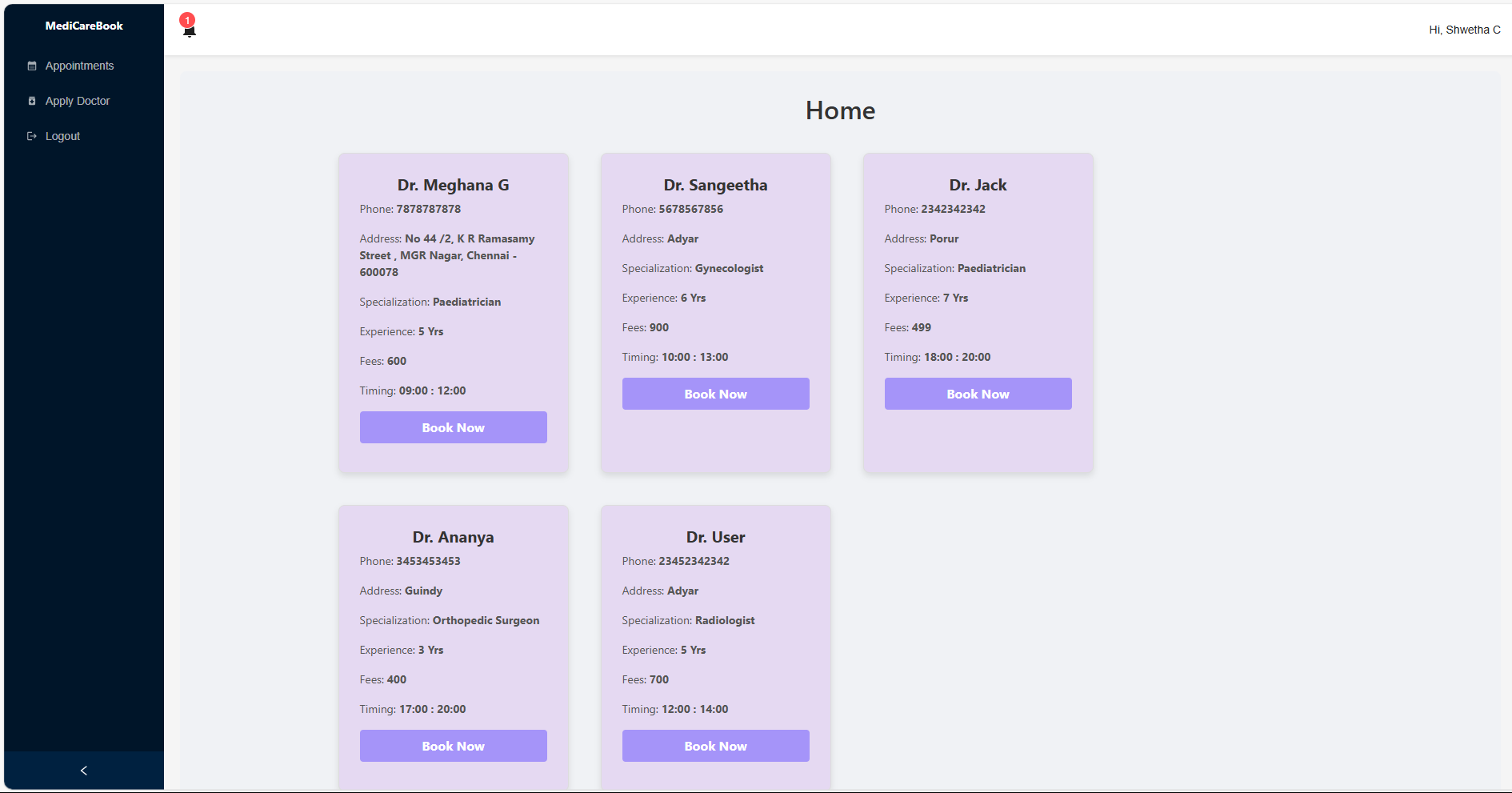
## 2. Login and Register Page :

****

The Register Page allows new users to create an account. Key features include:

* **Role Selection:** Users can select their role (Customer or Doctor), which determines their dashboard and available features.
* **Input Fields:** Fields for essential information such as name, email, password, and (for doctors) additional details like specialization.
* **Form Validation:** Ensures that users enter valid information before submitting the form (e.g., password requirements, unique email).
* **Account Creation:** After successful registration, the user’s information is saved in the database, and they can proceed to log in.

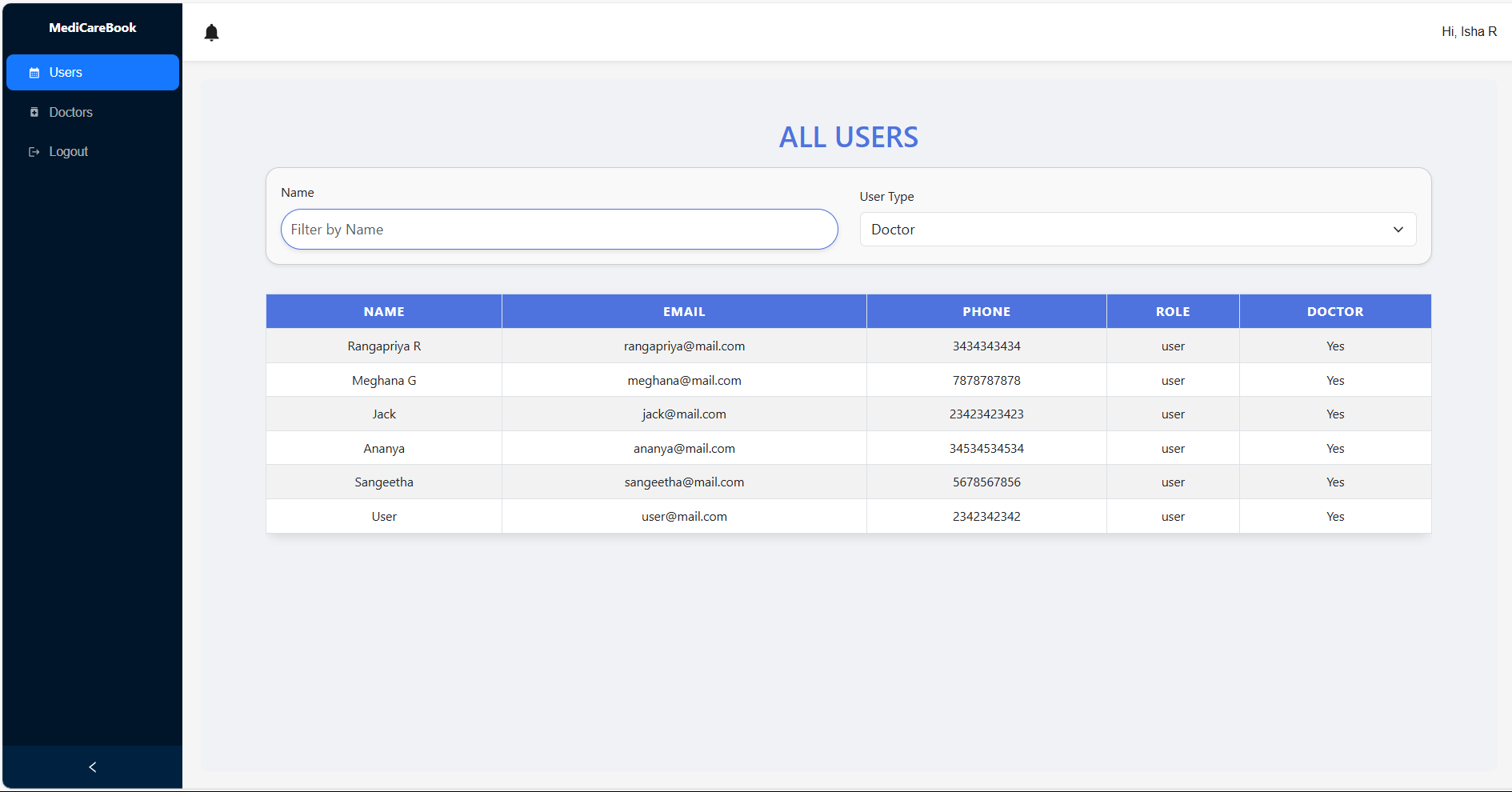
## 3. User Dashboard : The User Dashboard provides a clean and user-friendly interface where users can manage their appointments, apply to become a doctor, and view notifications.

****

Key Features:

* **Appointments Section:** Users can view their upcoming and past appointments in a table format, including details such as the doctor's name, appointment date, and status.
* **Apply Doctor Button:** If the user is not a doctor, they have the option to apply to become one. This feature is accessible through a dedicated button in the sidebar.
* **Notifications:** A notification icon displays the count of unread notifications. Users can click on it to view their notifications.
* The dashboard features a sidebar with icons for navigation (Appointments, Apply Doctor, Logout).
* The header displays the user's name and a notification badge.

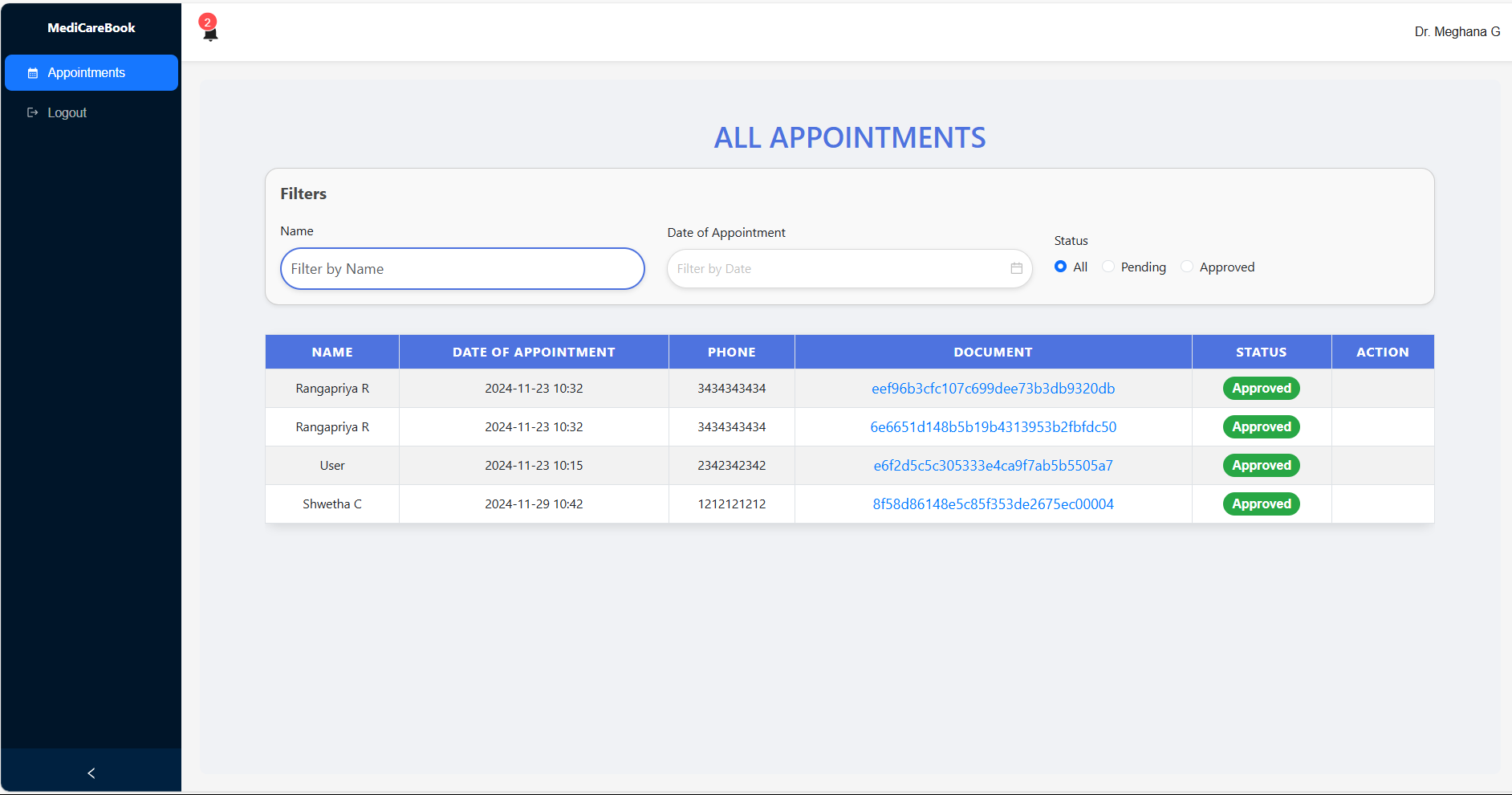
## 4. Admin Dashboard : The Admin Dashboard is designed for administrators to manage users, doctors, and appointments efficiently.

****

Key Features:

* **User Management:** Admins can view all registered users, including their roles (Doctor, User, Admin), and filter users by name and type.
* **Doctor Management:** Admins can approve or reject doctor applications. This section also includes filtering options to search for doctors based on specialization or status.
* **Appointments Management:** Admins can view all appointments made by users and filter them based on the doctor's name and status (Pending, Approved).
* The main content area displays tables with data, allowing for easy management and filtering.

## 5. Doctor Dashboard : The Doctor Dashboard allows doctors to manage their appointments and view their patient details.

****

Key Features:

* **Appointments Overview:** Doctors can see a list of their appointments, including patient names, appointment dates, and statuses.
* **Status Management:** Doctors can approve or reject appointment requests from users, with functionality to update the status directly from the dashboard.
* **Profile Management:** Doctors can update their profiles, including their specialization, experience, and fees.

## 6. Filtering Functionality: The filtering functionality is implemented across various sections of the application (User , Admin, and Doctor dashboards) to help users easily find relevant information.

* **Search Bars:** Users can type in names or other identifiers to filter through appointments, users, or doctors.
* **Dropdown Filters:** Users can select options from dropdown menus to filter results based on specific criteria, such as user type (Admin, User, Doctor) or appointment status (Pending, Approved).
* **Range Filters:** For doctors, there is a fees range filter that allows admins to filter doctors based on their fees.

# 10. TESTING

The testing strategy for the MediCareBook : Book a Doctor application focuses on ensuring the functionality, security, and performance of the application through API testing using Postman and automated UI testing using Selenium .

## Testing Tools

1. **Selenium:** Selenium is used for automated UI testing to ensure that the frontend user interface behaves as expected. It can be integrated with Mocha or Jest for functional testing in a browser environment.

* **Use Case:** Checking user interactions like clicking buttons, selecting dates, and filling out forms.

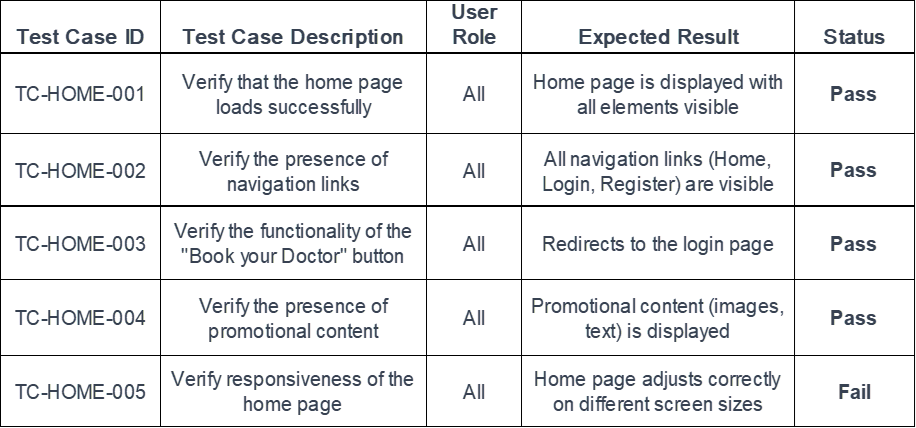
1. **Postman:** Postman is used to test all API endpoints, including user authentication, product management, order processing, and cart functionalities. Each endpoint is tested by sending various requests (GET, POST, PUT, DELETE) with different parameters to verify the accuracy and correctness of the responses.

* **API Testing with Postman**: Postman is used to test all API endpoints. Each endpoint is tested by sending various requests (GET, POST, PUT, DELETE) with different parameters to verify the accuracy and correctness of the responses.
* **Error Handling and Validation**: Responses are tested for proper status codes (200, 400, 401, 404, etc.), ensuring that the system handles errors correctly and returns appropriate messages when invalid data is sent or when users try to access restricted resources without proper authorization.

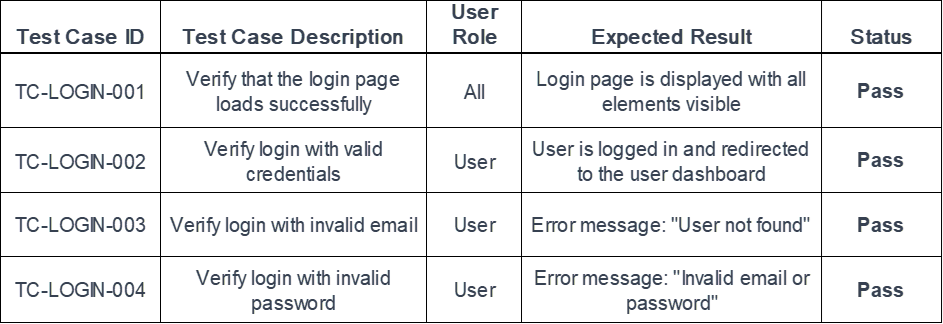
## UI Testing using Selenium :

Comprehensive test case table for the MediCareBook application, focusing on different user roles: Admin, Doctor, and User. Each test case includes the test case ID, description, expected result, and the user role applicable for the test.

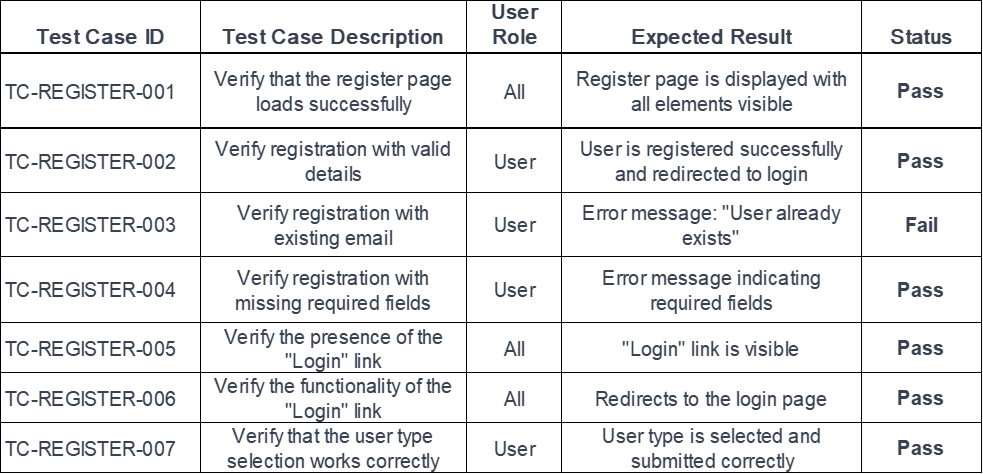
1. **Test Cases for Home Page**

****

1. **Test Cases for Login Page**

****

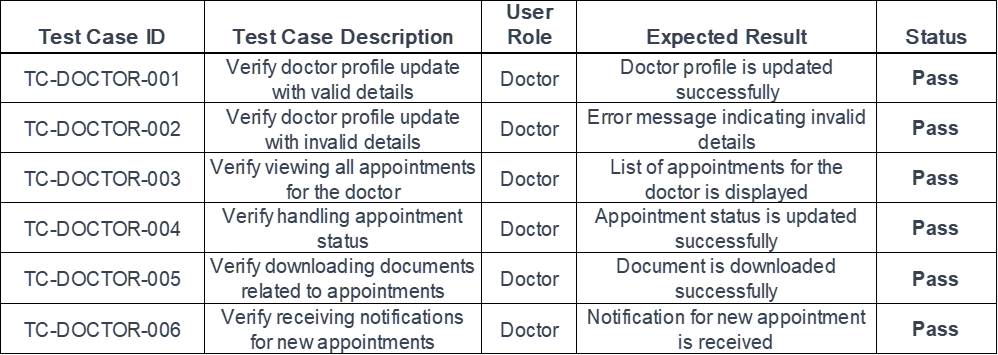
1. **Test Cases for Register Page**

****

1. **Test Cases for User Role: User**

****

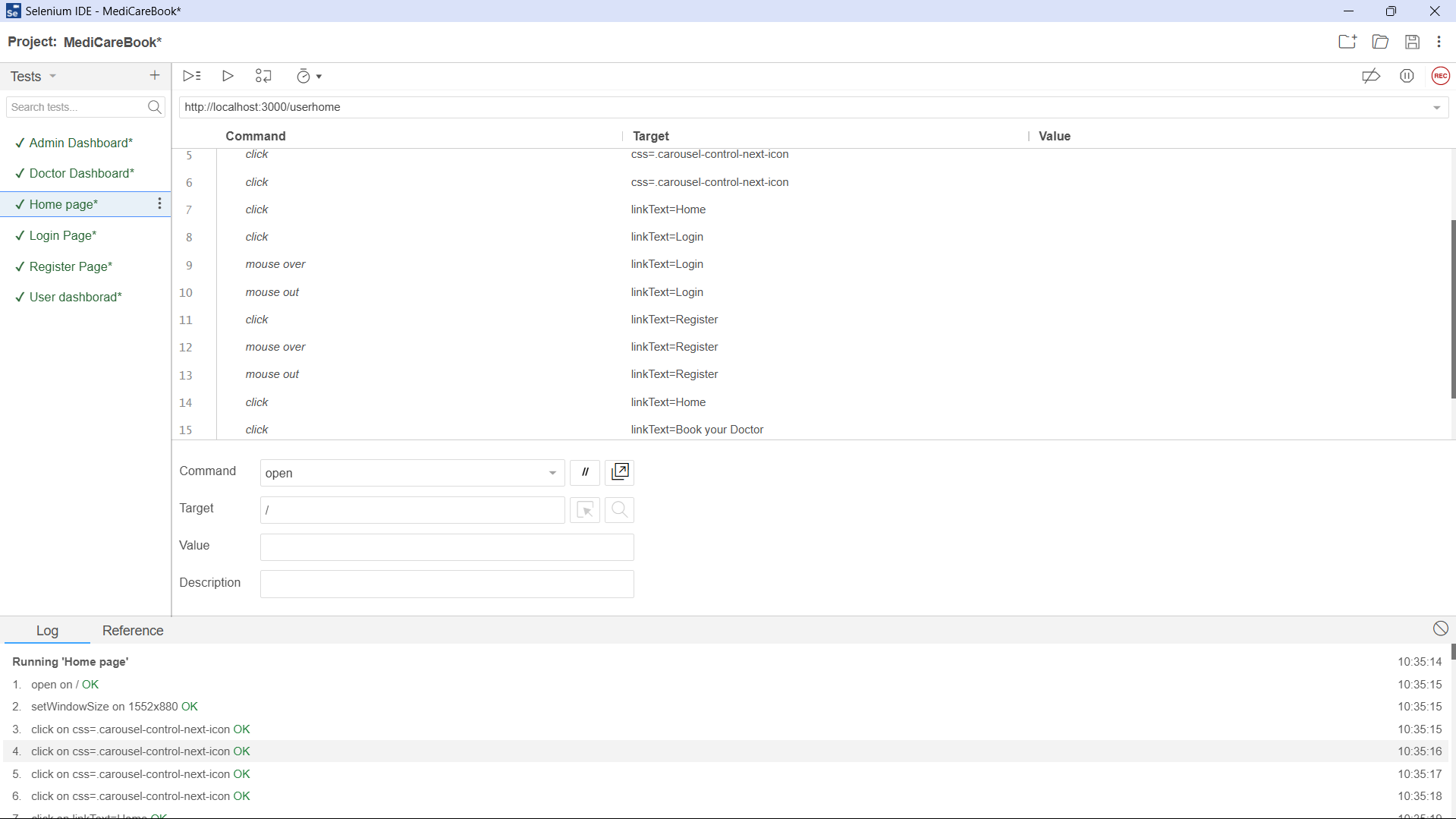
1. **Test Cases for User Role: Doctor**

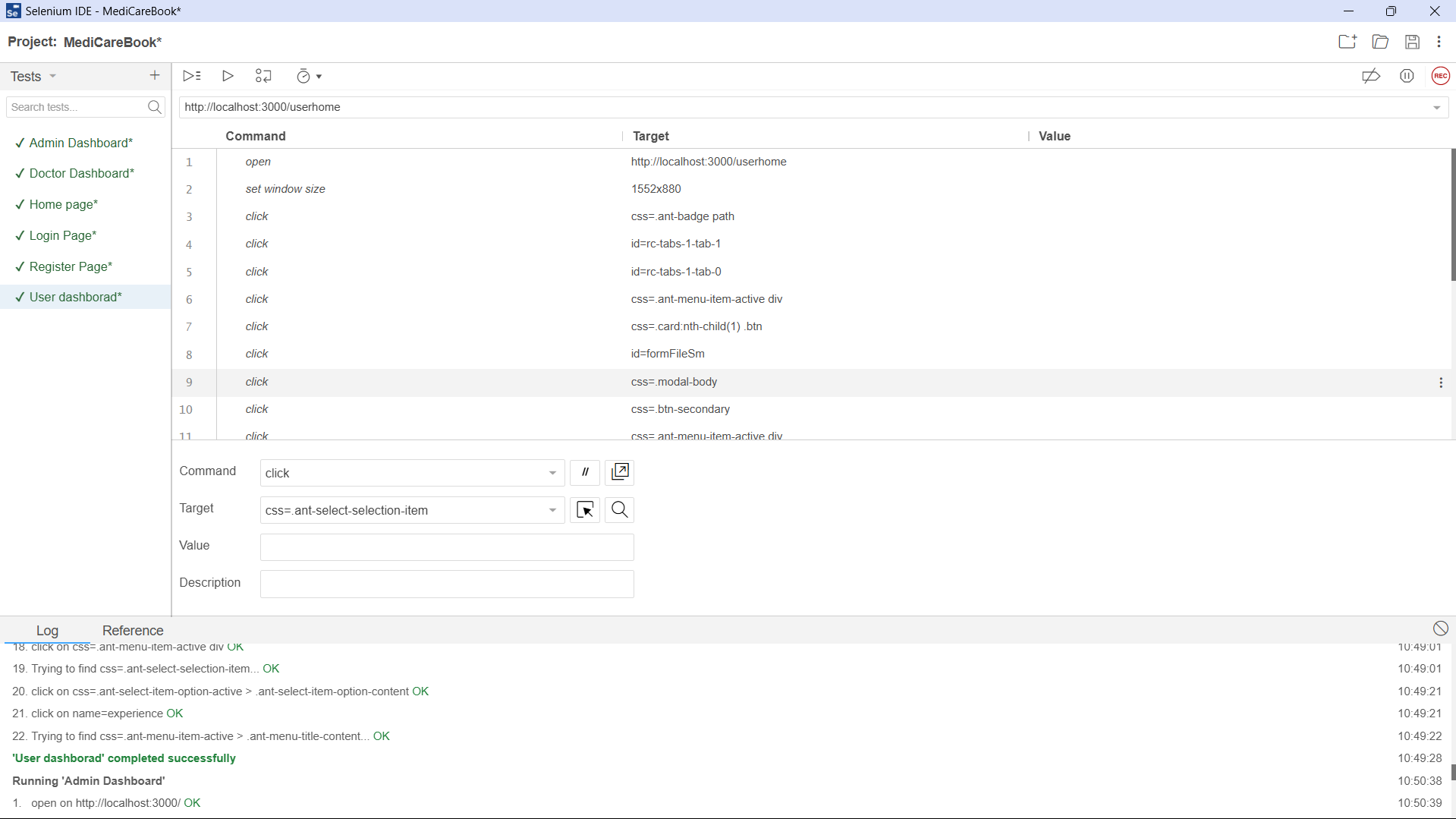
****

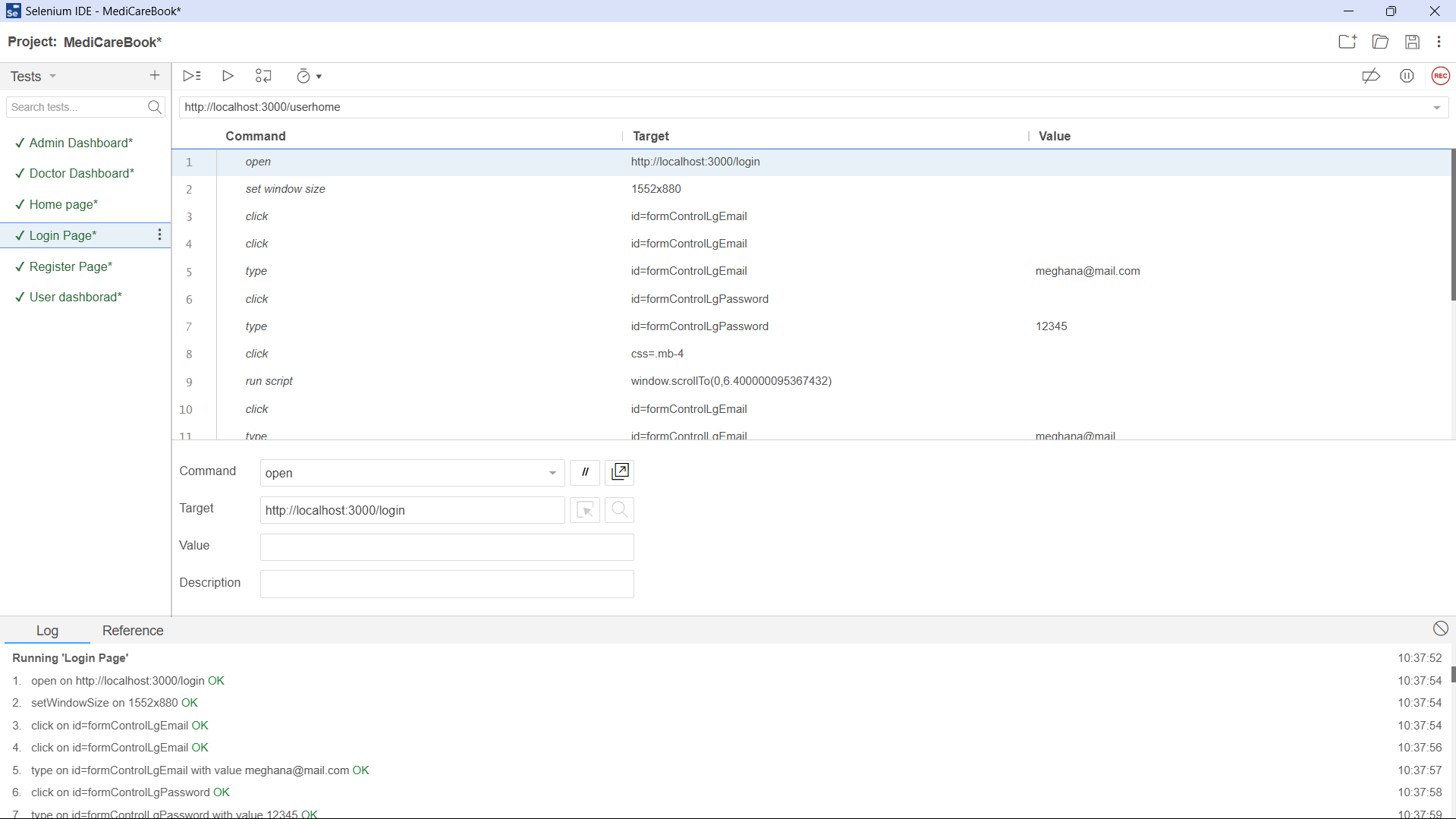
1. **Test Cases for User Role: Admin**

****

Automated UI testing using Selenium was implemented to execute these test cases efficiently, ensuring a robust and reliable user experience.

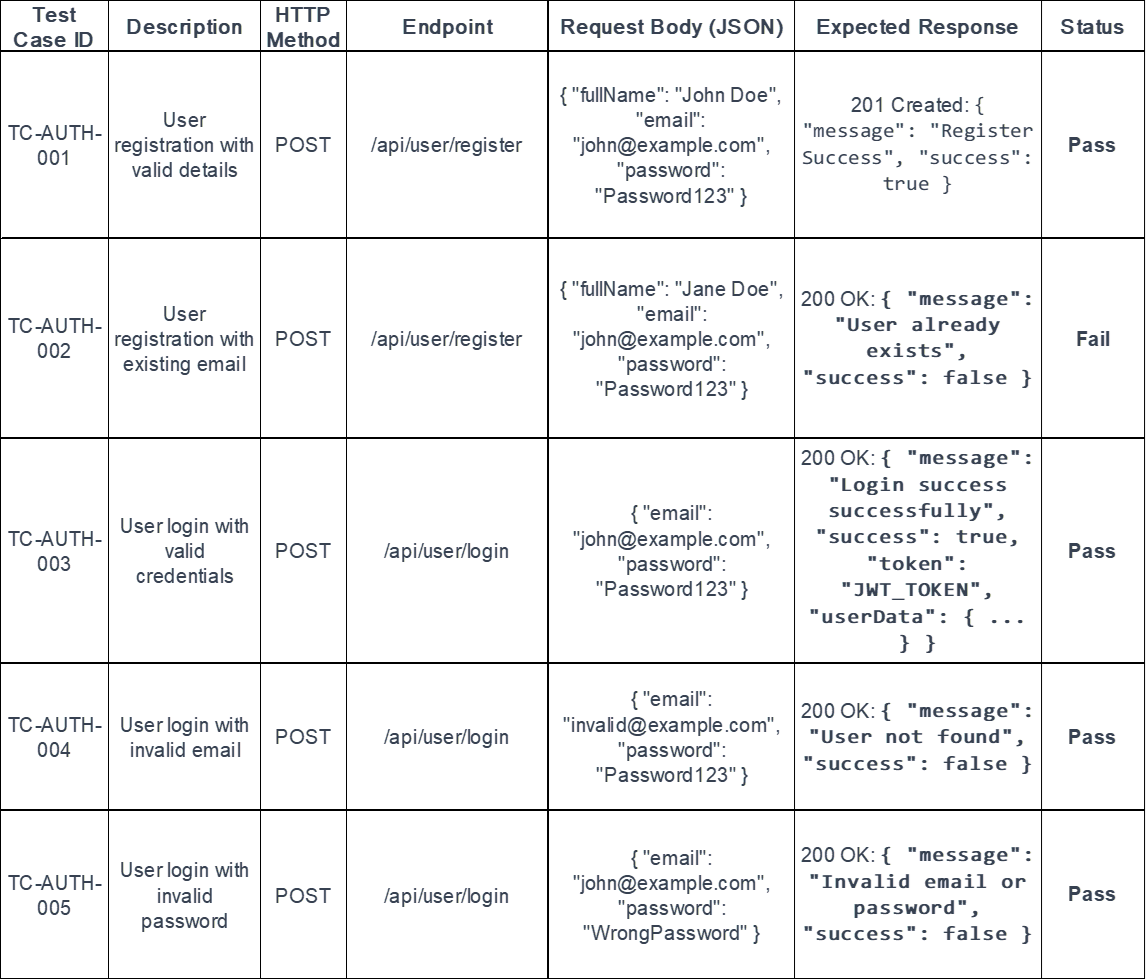






## API Testing using Postman :

A comprehensive test case table for the APIs of the MediCareBook application that can be tested using Postman. Each test case includes the test case ID, description, HTTP method, endpoint, request body (if applicable), expected response, and the applicable user role.

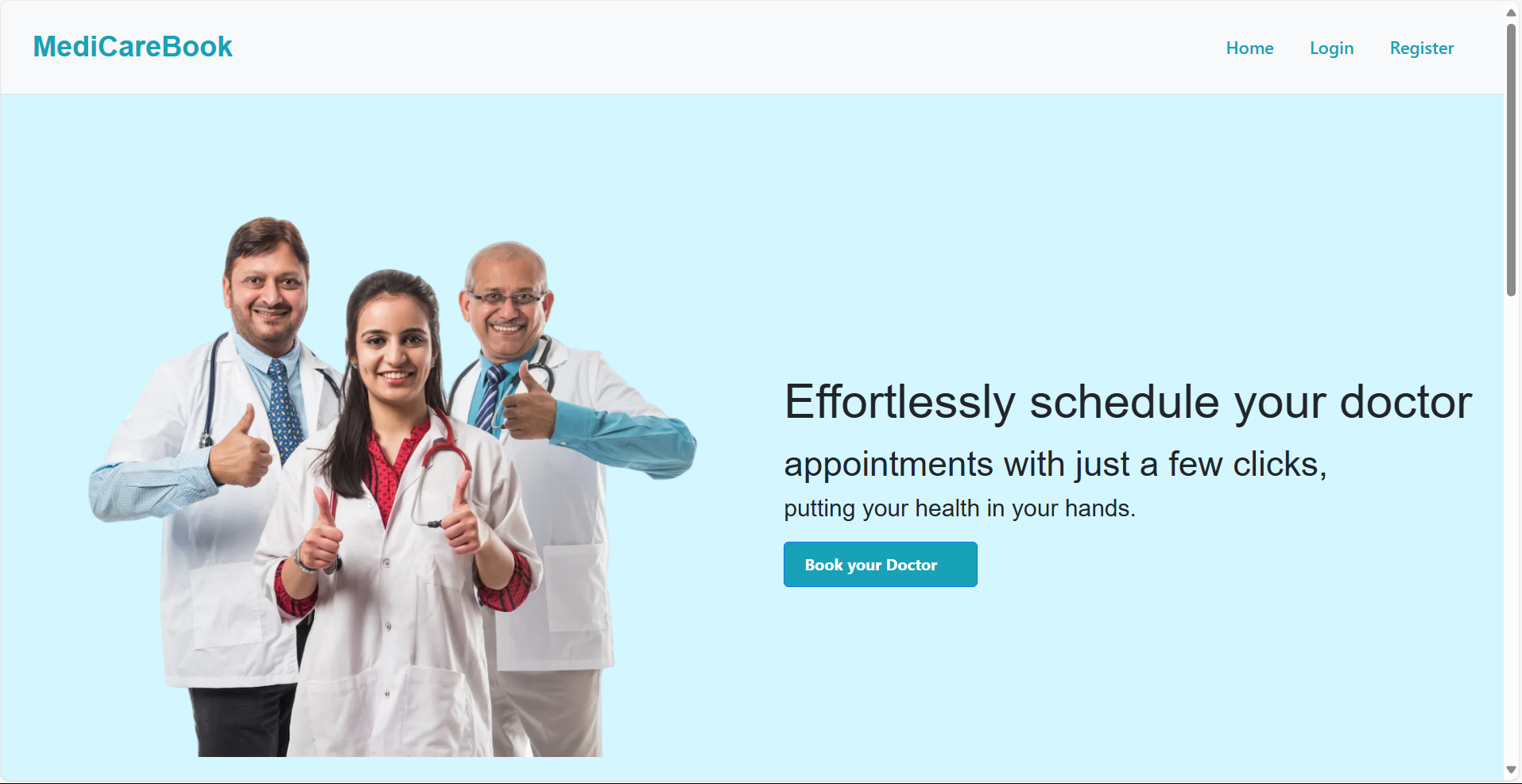




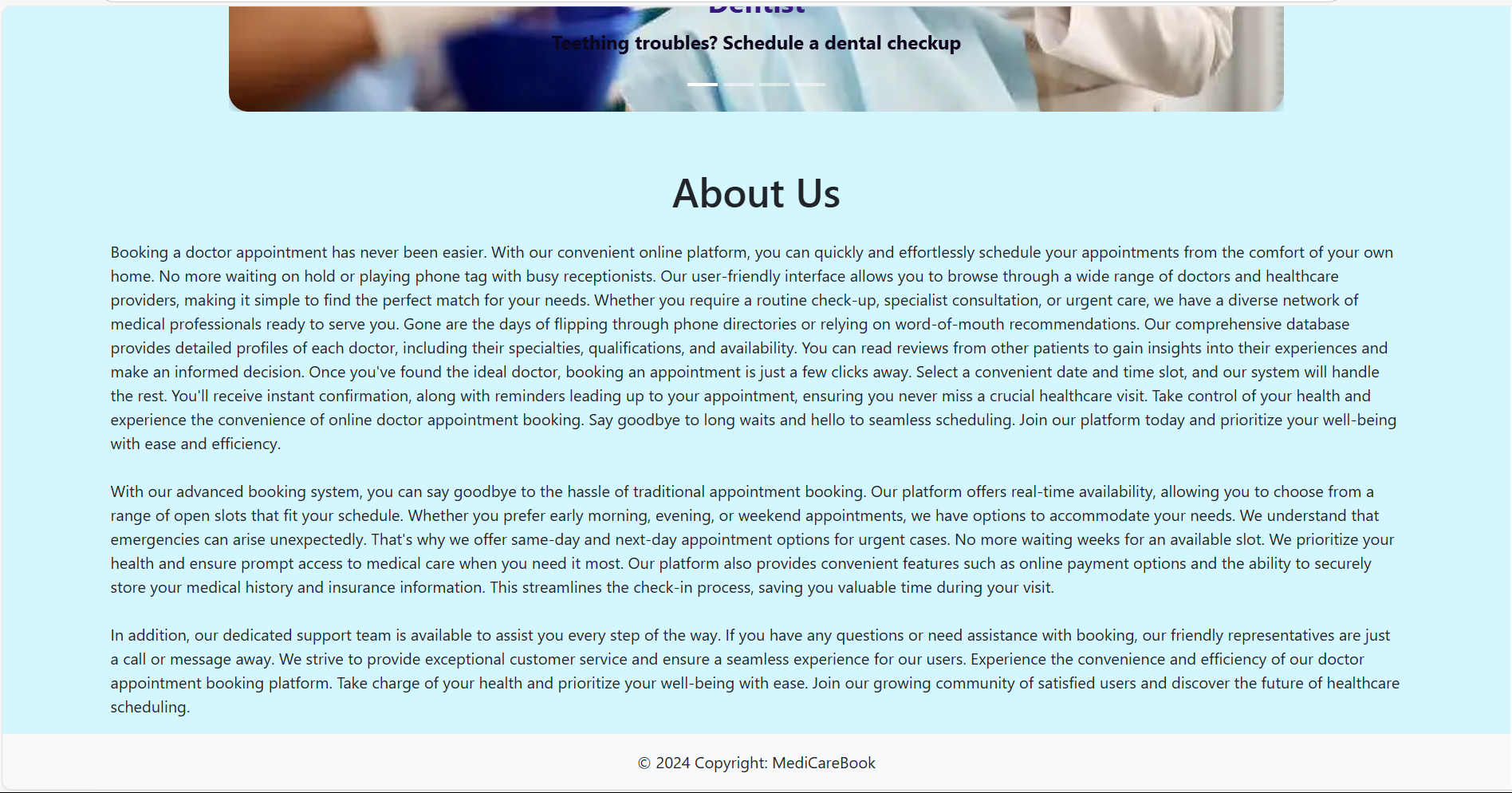


# 11. SCREENSHOTS OR DEMO

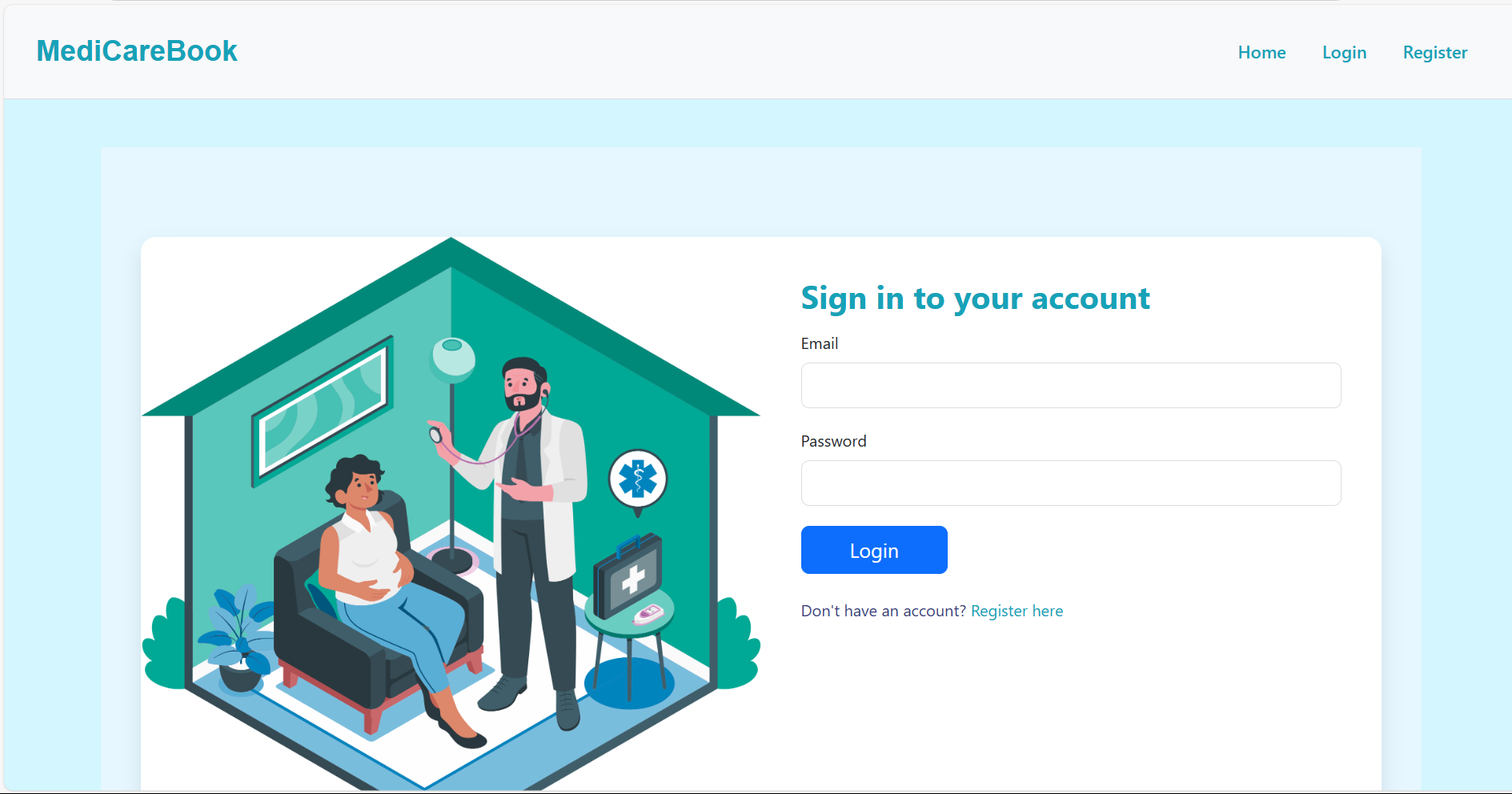
## Home Page :

****

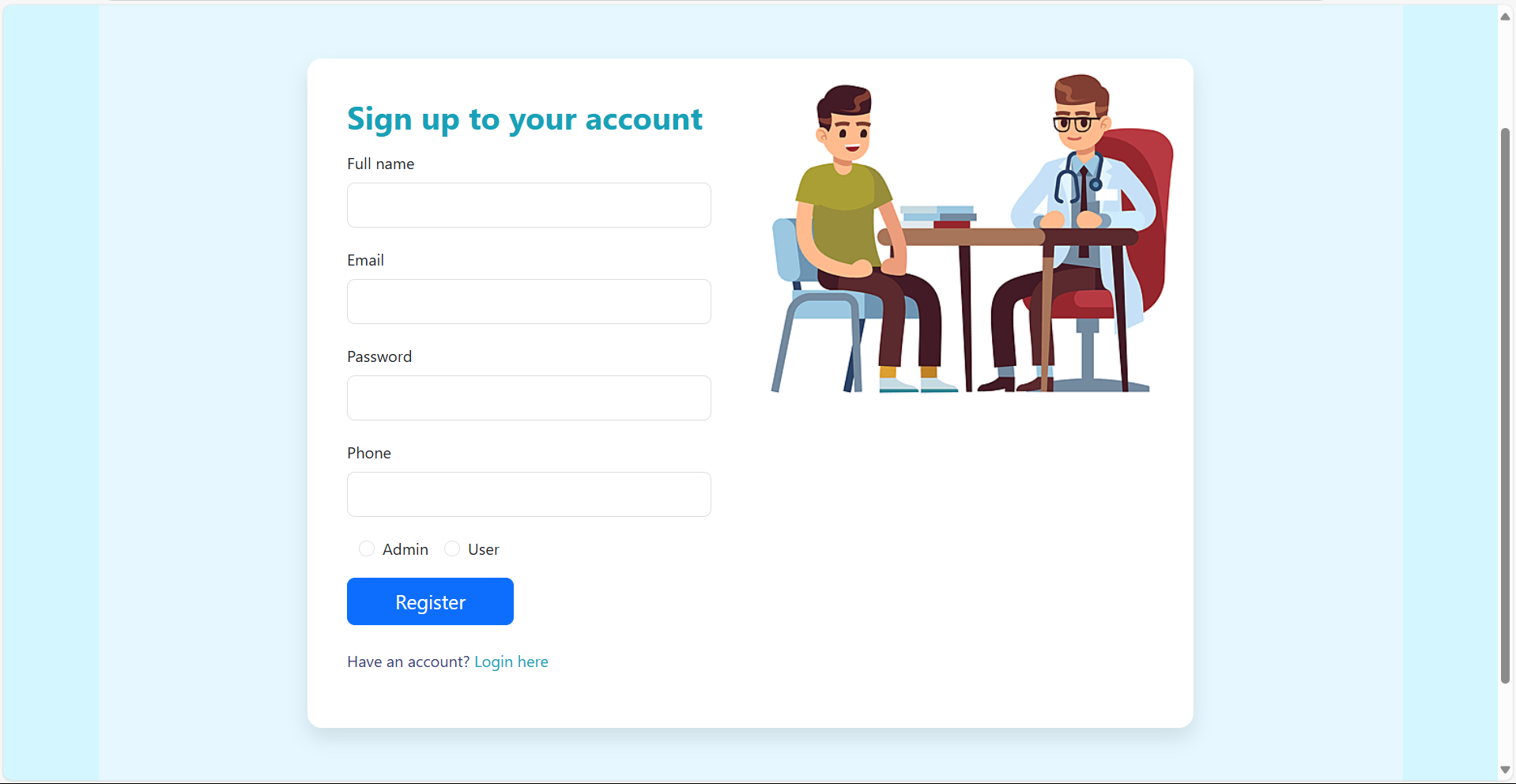
****

****

## Login Page :

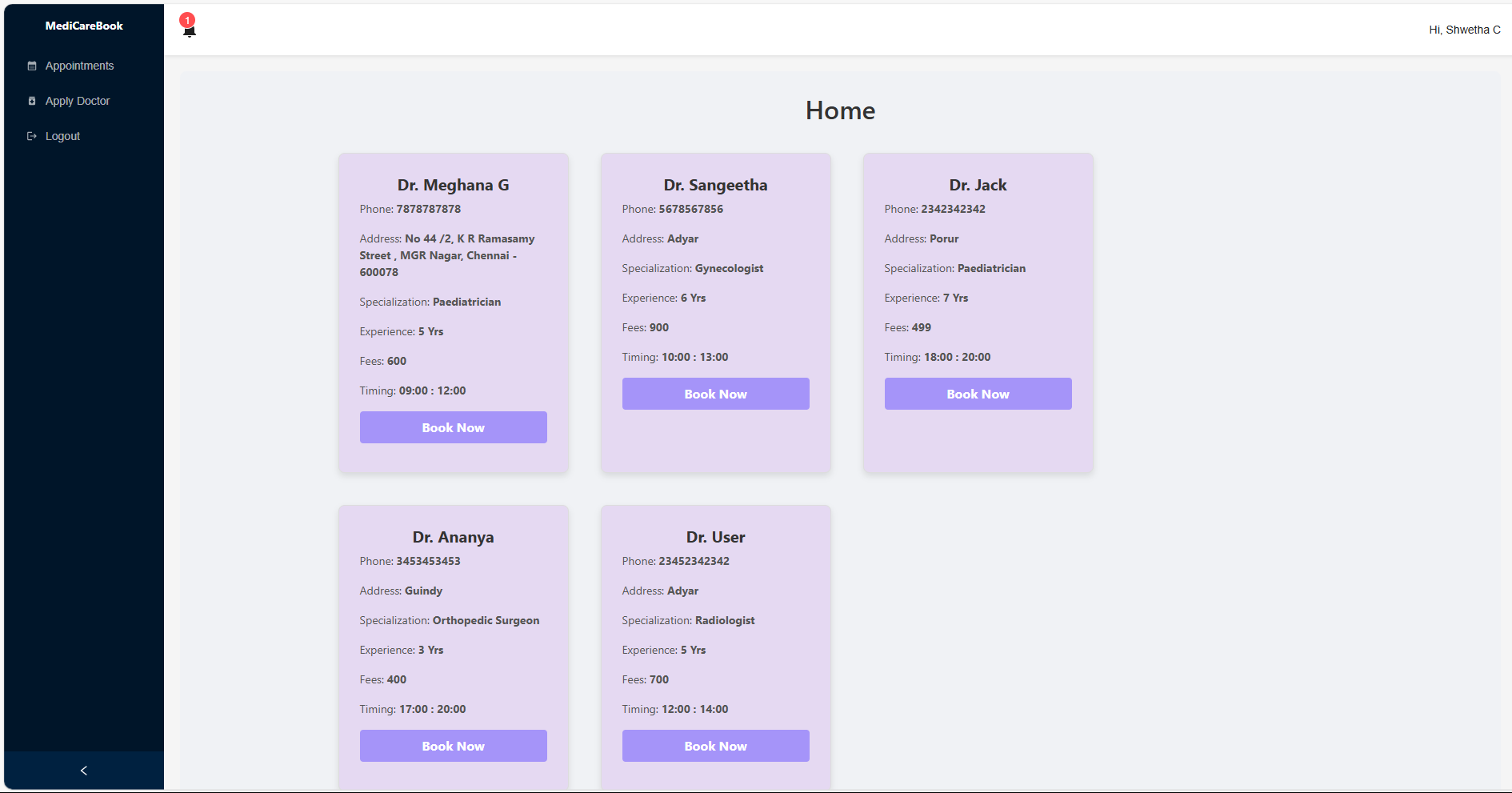
****

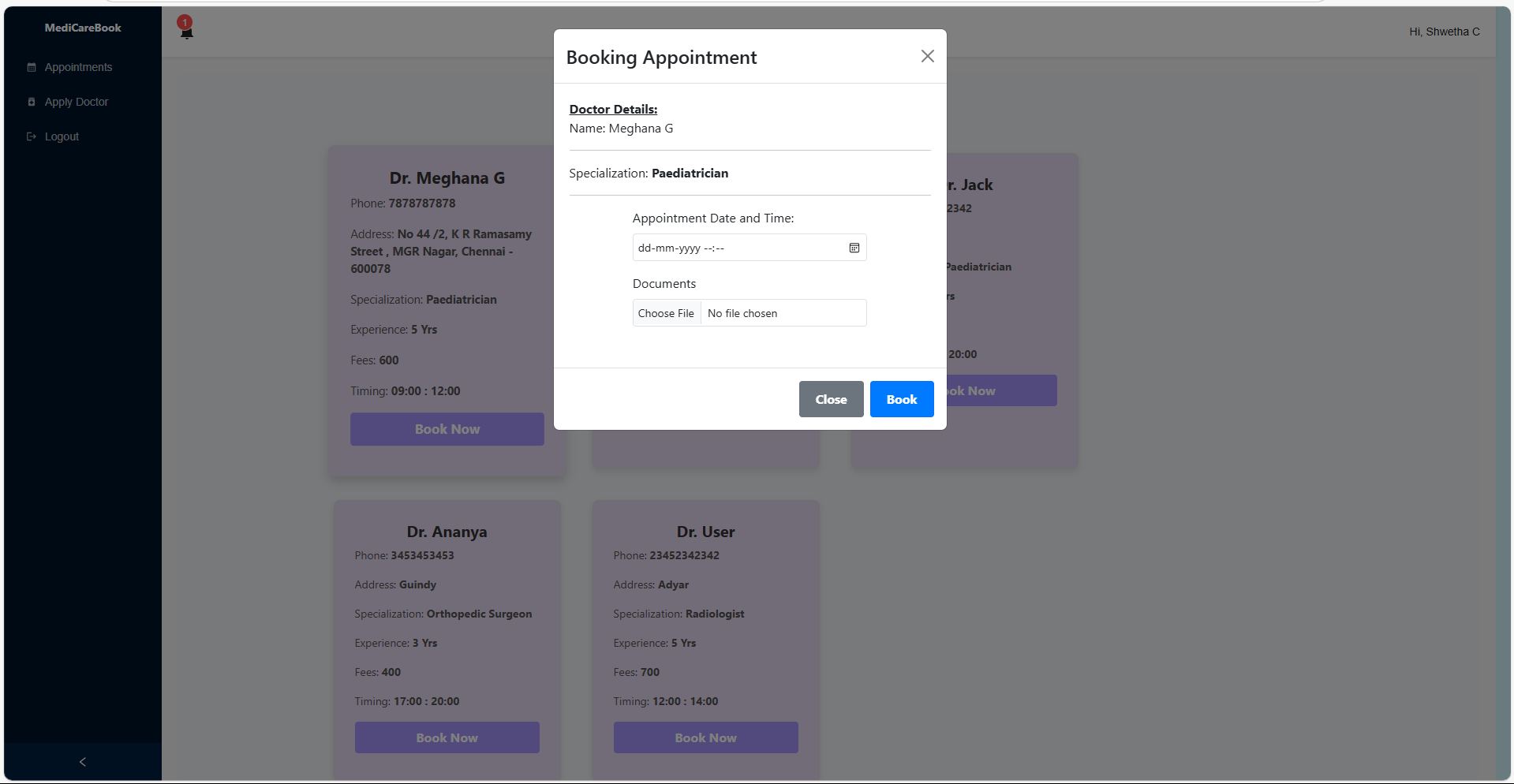
## Register Page :

****

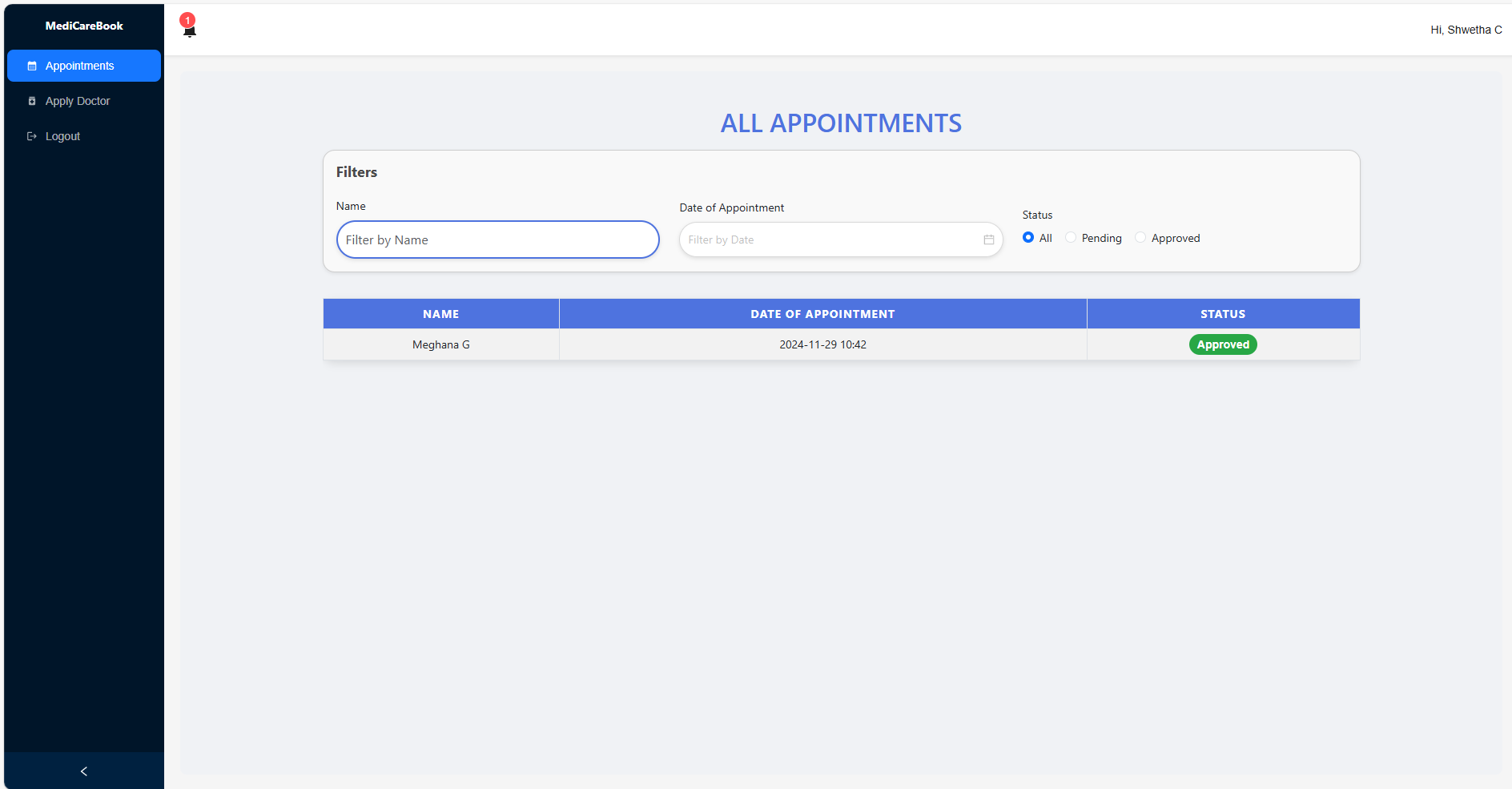
## User Dashboard :

### Home :

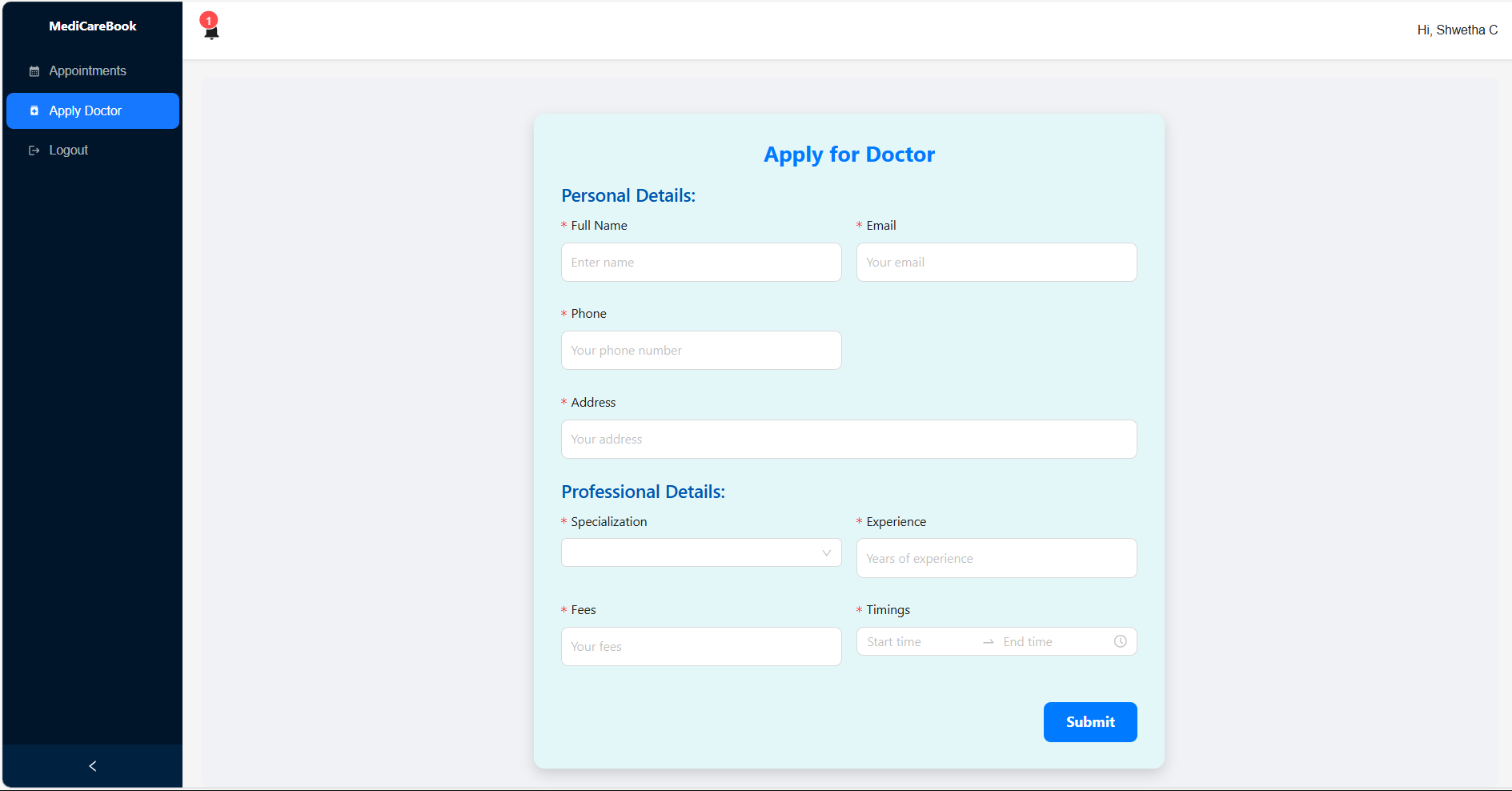
****

****

### Appointments :

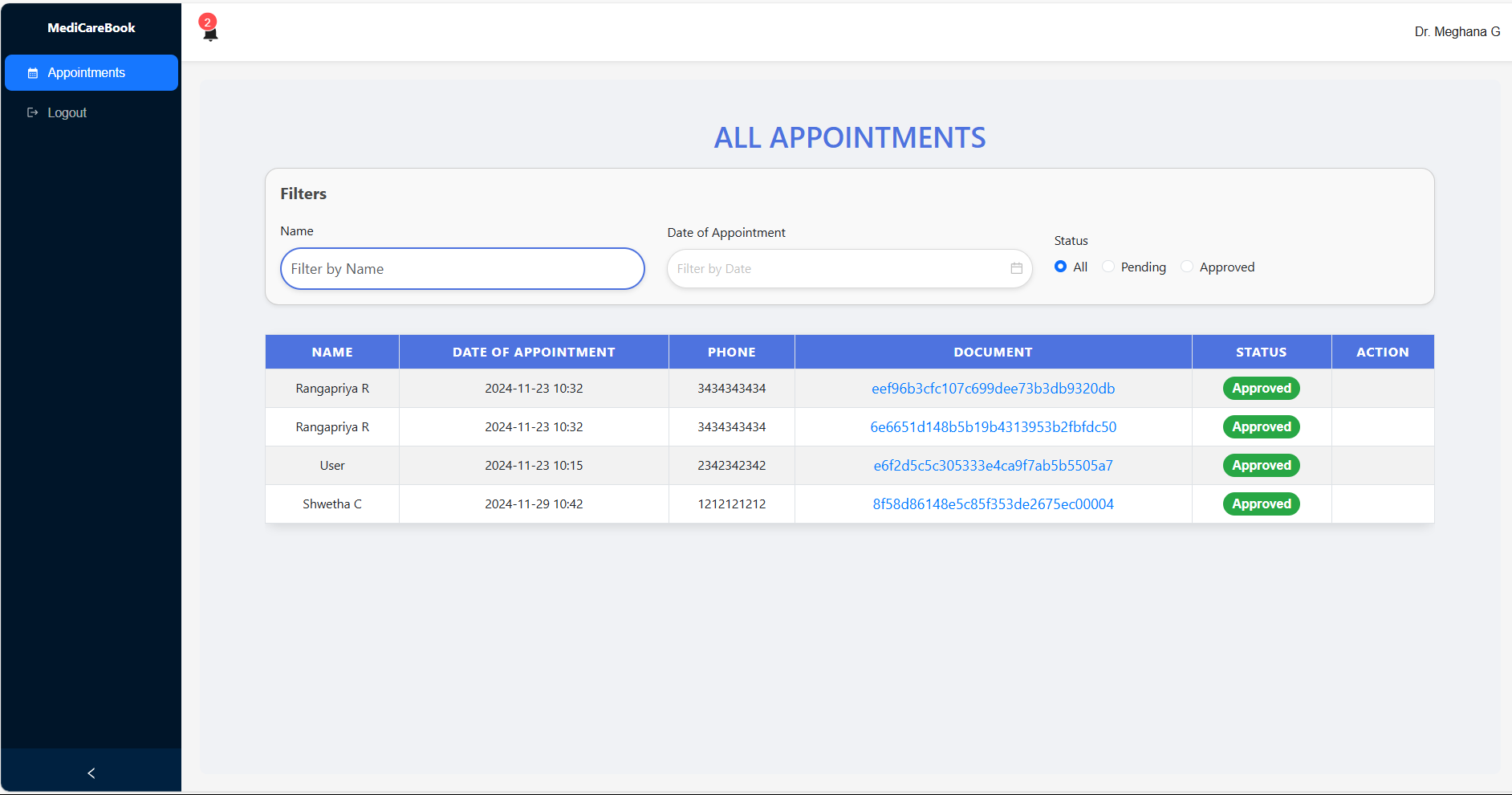
****

### Apply for Doctor :

****

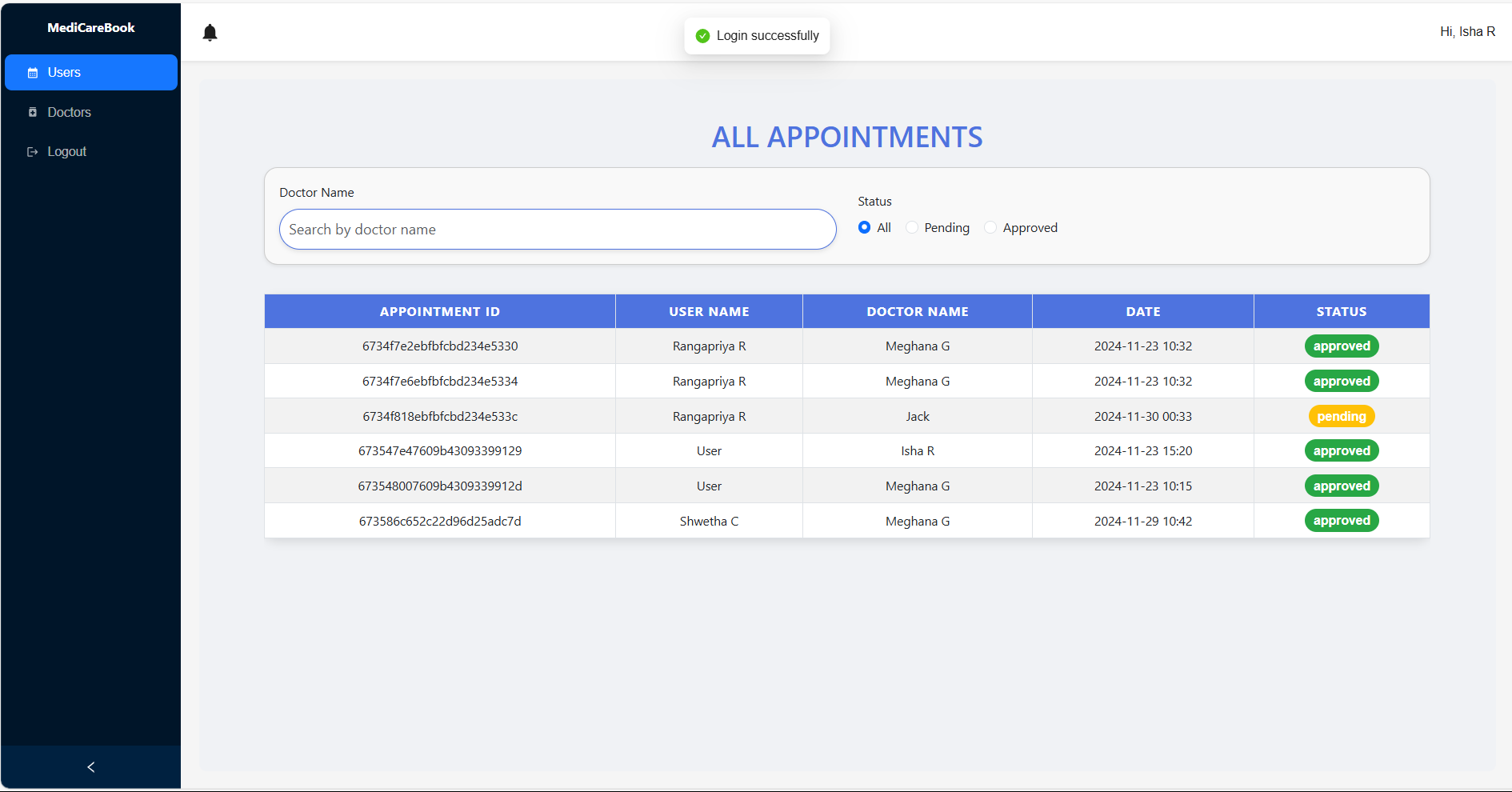
## Doctor Dashboard :

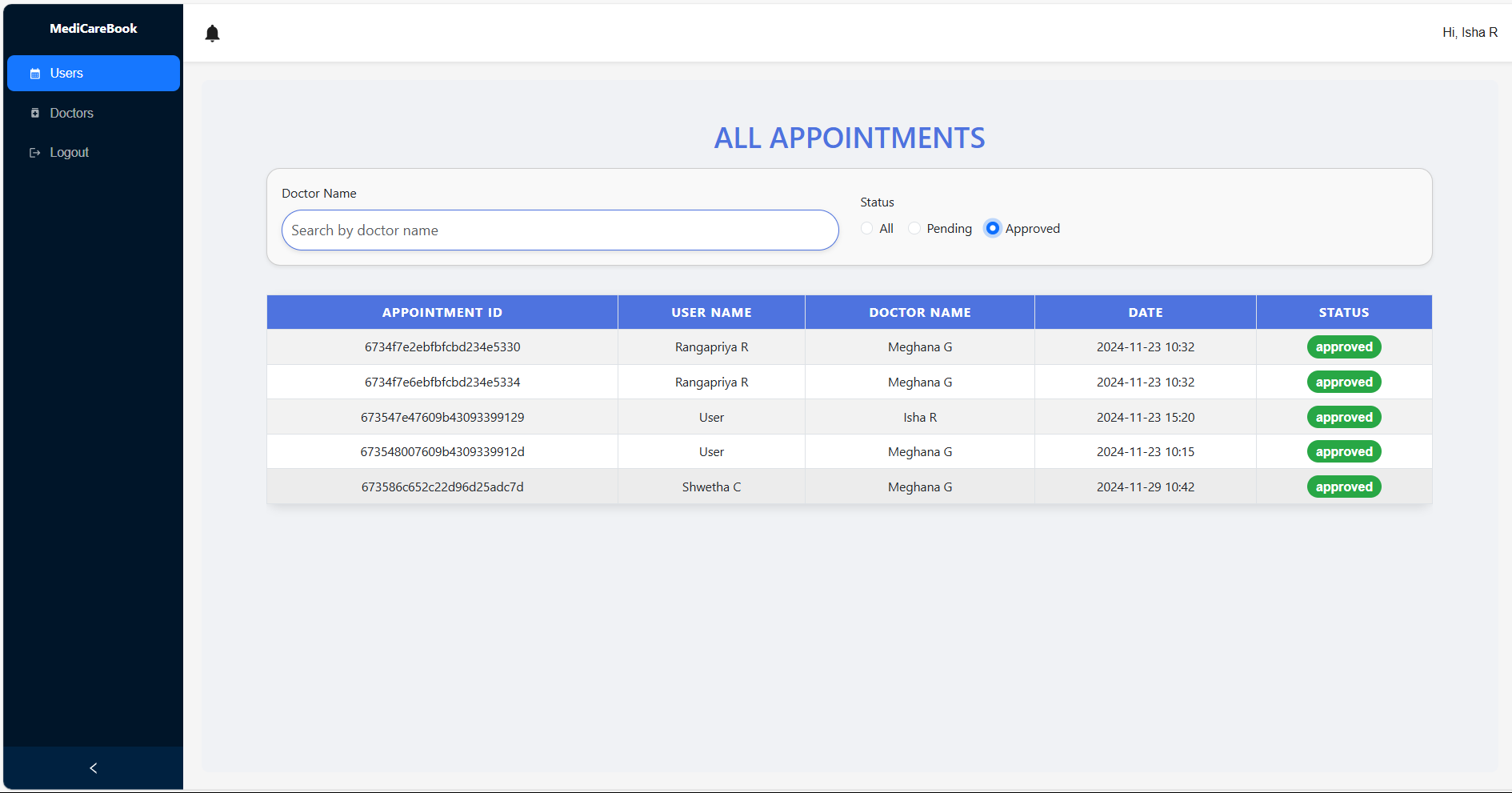
### Appointments:

****

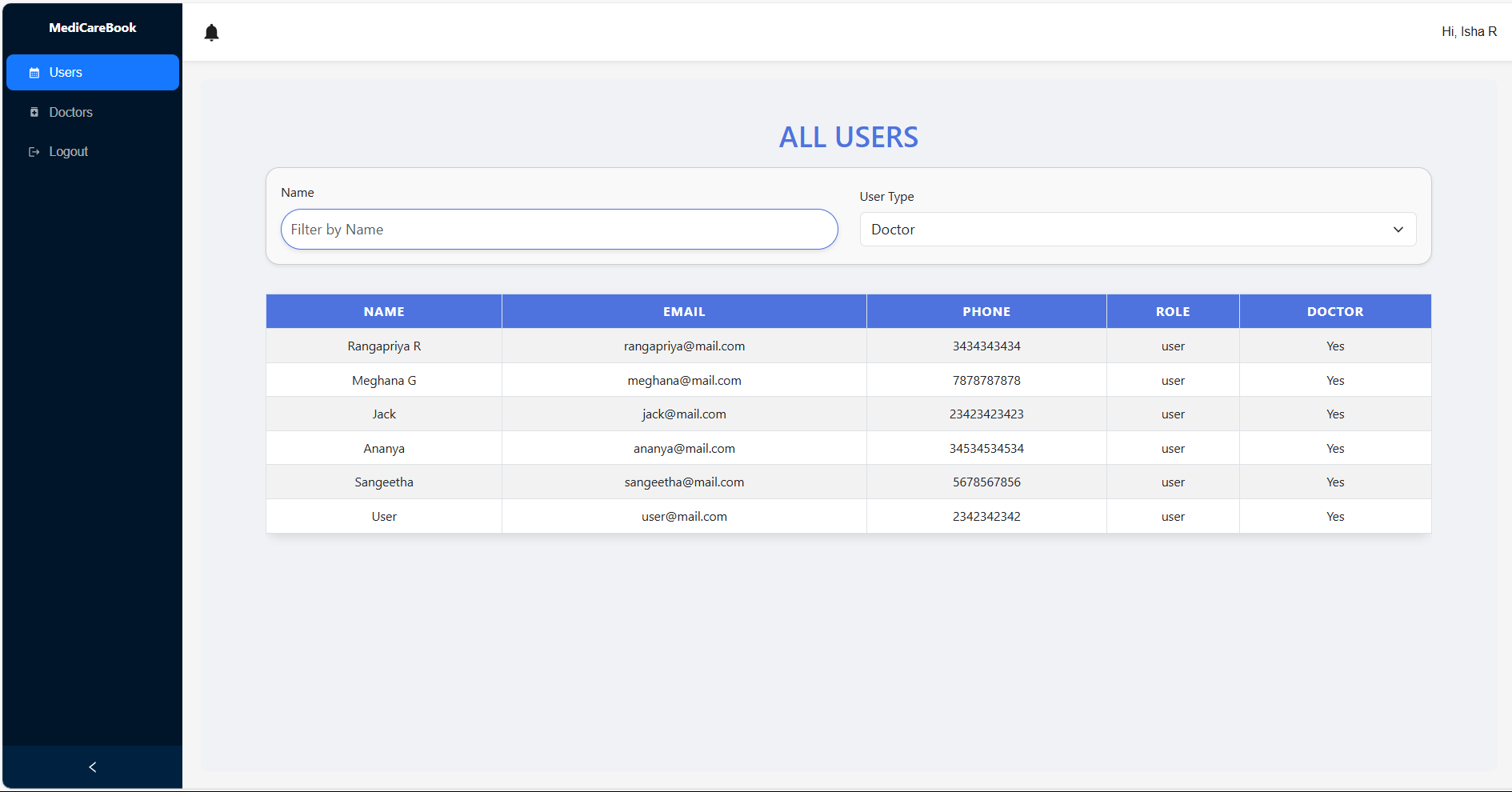
## Admin Dashboard :

### Appointments :

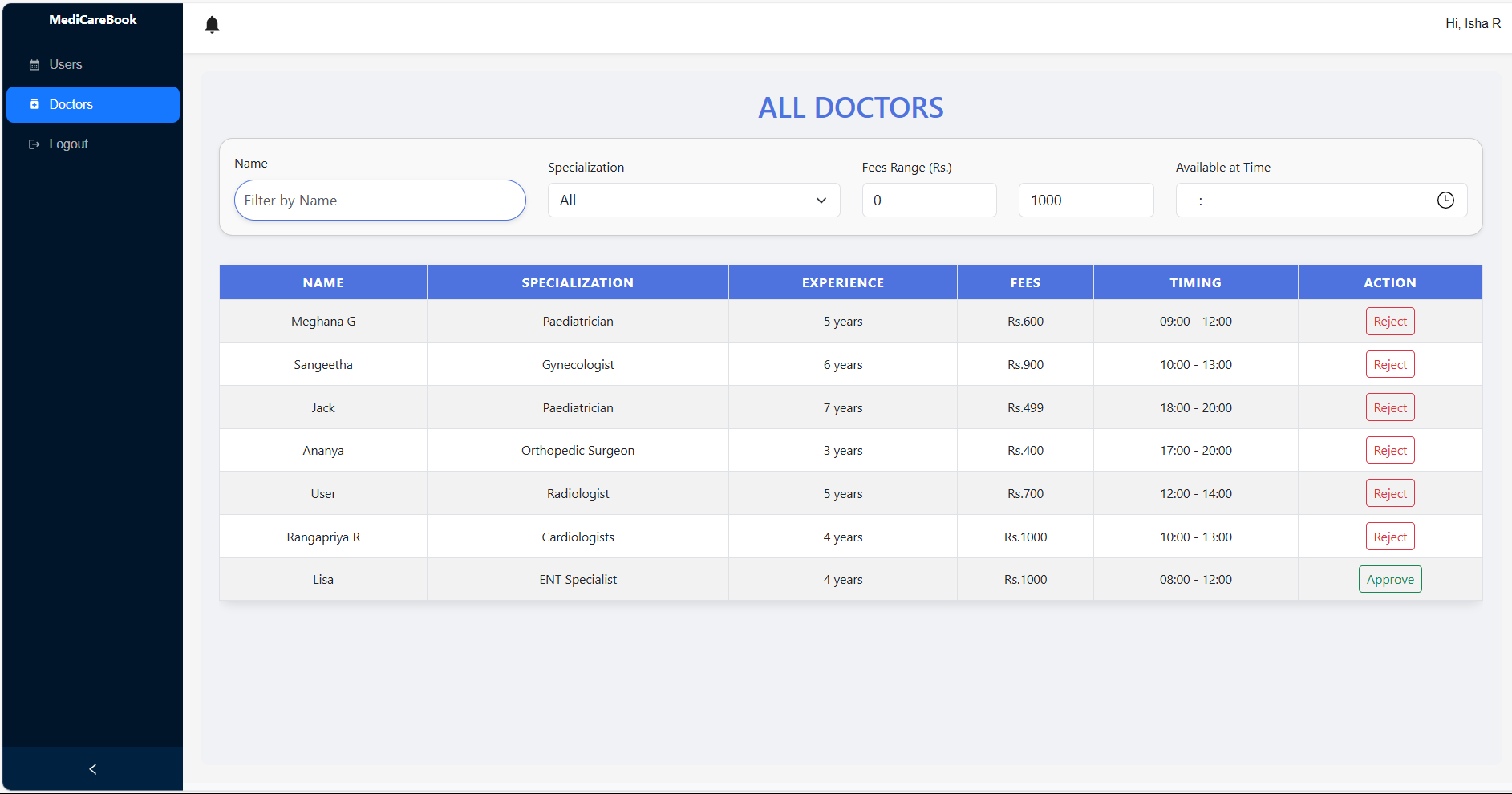
****

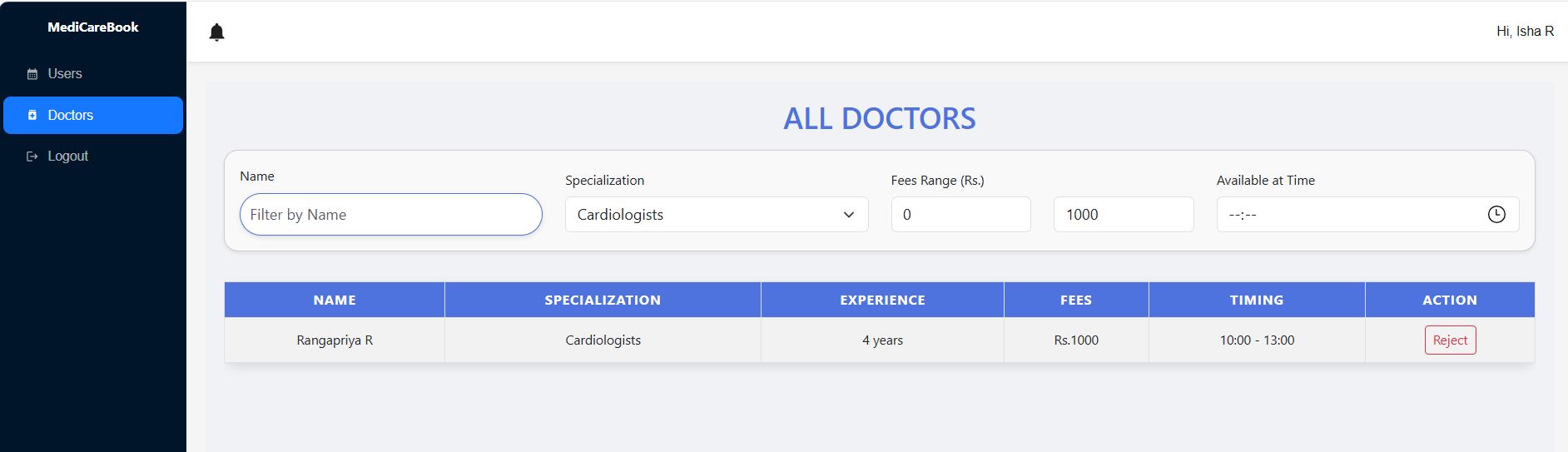
****

### Users :

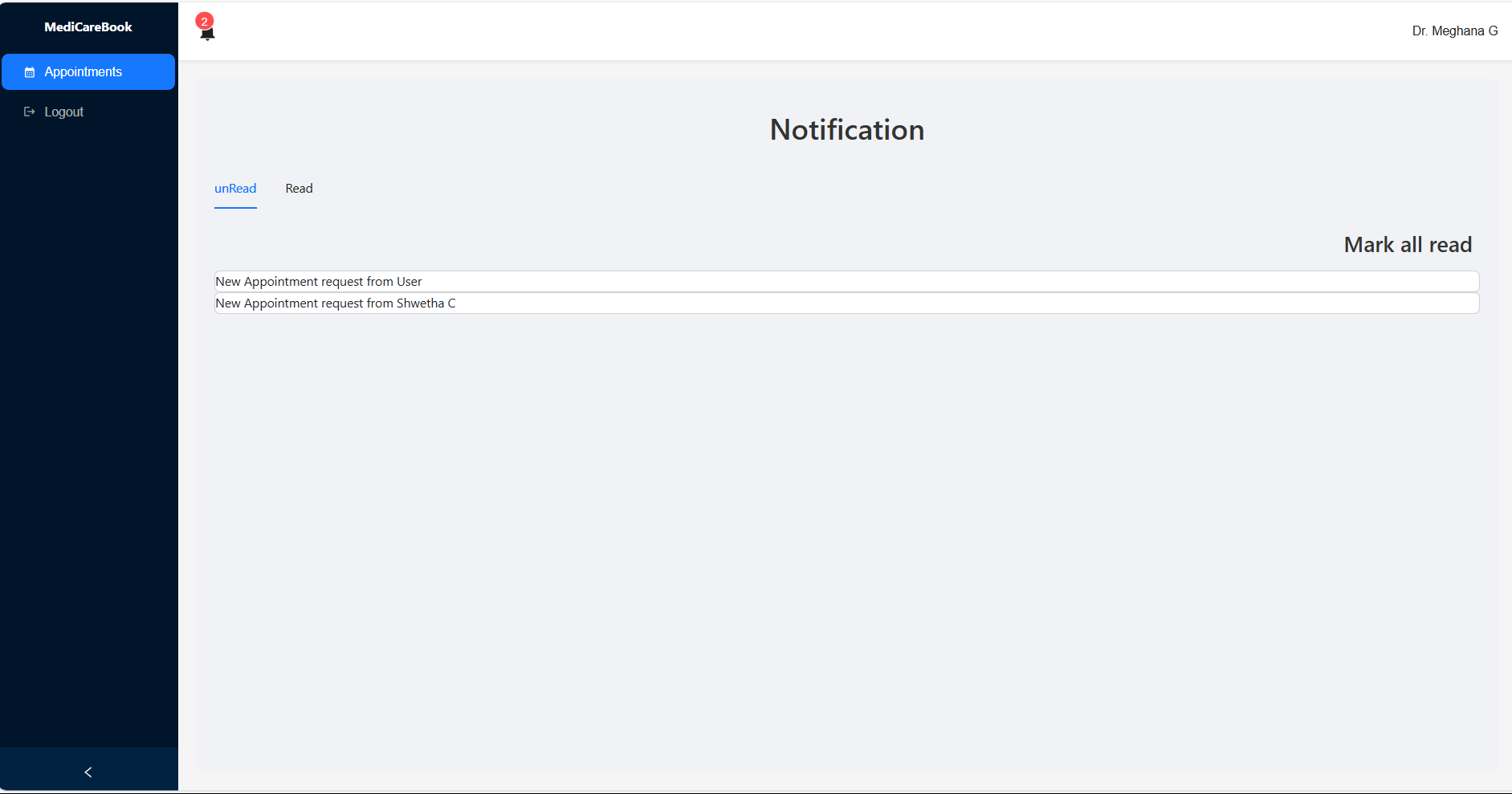
****

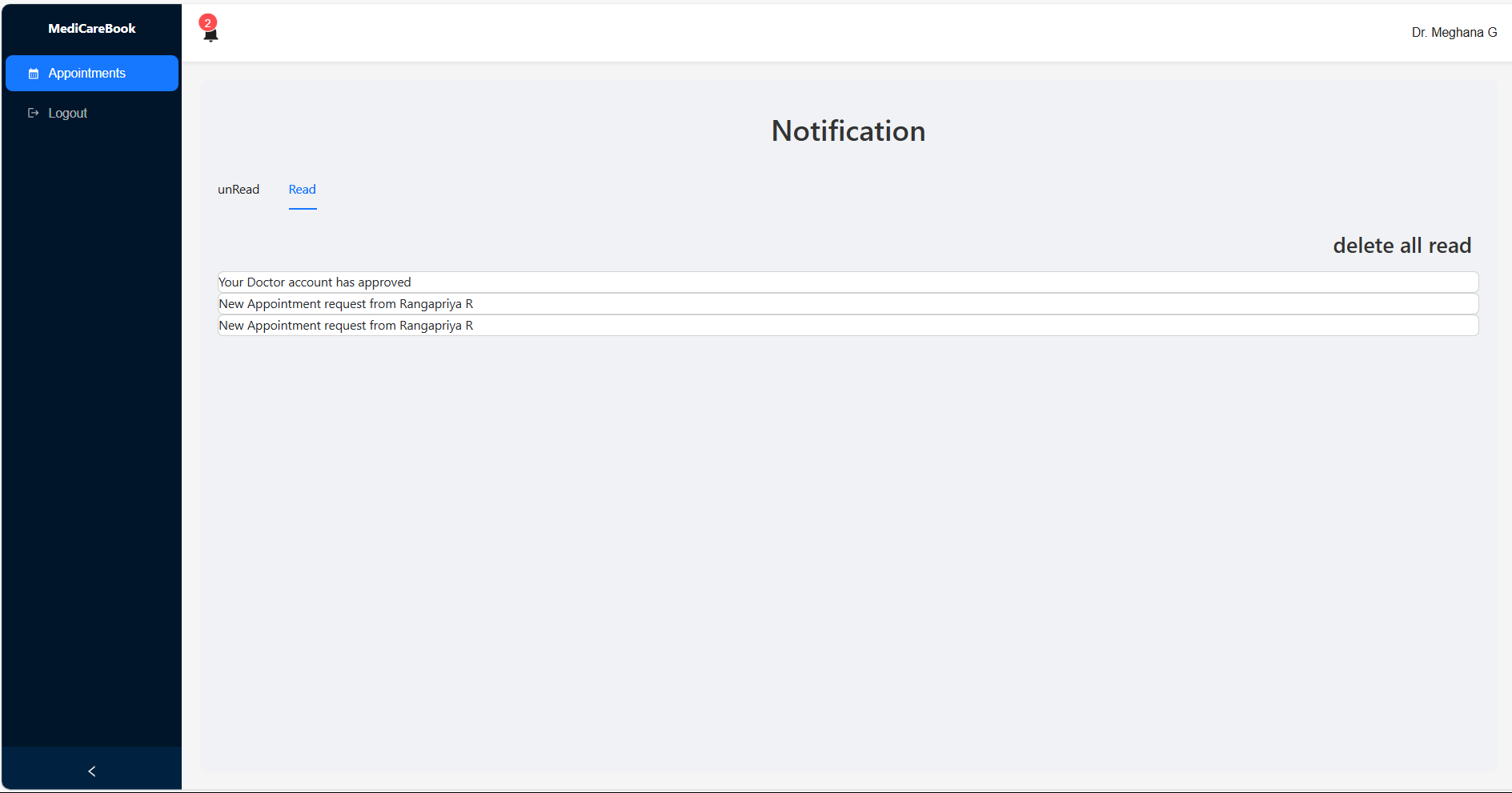
### Doctors :

****

****

## Notification :

****

****

**Demo Video Link :** [MediCareBook demo.mp4](https://drive.google.com/file/d/1-RSa2yp2xc60g5JZwF6f-A63LTF0OhFP/view?usp=drive_link)

# 12. KNOWN ISSUES

1. **Forgot Password Functionality**:

* **Description**: The forgot password functionality does not currently include a verification step to confirm the validity of the email address before proceeding with the password reset process.
* **Impact**: If a user mistypes their email address, they will not receive any feedback, potentially leading to confusion as the reset link may not reach them.

1. **Cross-browser Compatibility**:

* **Description**: Certain features, like the image carousel and zoom functionality, may not display or function properly in all browsers, particularly older versions of Internet Explorer or Safari.
* **Impact**: Users on specific browsers may encounter layout or functionality issues, resulting in a suboptimal experience.

1. **Mobile Responsiveness**:

* **Description**: Some pages, including the Admin Dashboard ,may not be fully responsive on smaller devices or screens.
* **Impact**: Mobile users may experience issues with page alignment, layout, or broken UI components when accessing the app on smaller screens.

1. **Authentication Token Expiry Handling**:

* **Description**: The system does not consistently handle token expiry, which may lead to users staying logged in after their session expires or requiring a manual refresh to reauthenticate.
* **Impact**: Users may experience inconsistencies in session management, such as being logged out unexpectedly or having difficulty accessing features after token expiry.

1. **Rejection of a Doctor in Admin Dashboard:**

* **Description**: The admin dashboard may lack the necessary functionality to reject a doctor's application, which is crucial for managing the approval process for new doctors.
* **Impact**: Admins may be unable to effectively manage doctor applications, leading to a backlog of pending requests and potential delays in the onboarding process.

These issues are being actively worked on and will be addressed in future updates to improve the overall user experience and functionality.

# 13. FUTURE ENHANCEMENTS

* **Advanced Appointment Management :** Introduce features such as appointment reminders via SMS or push notifications, and allow users to reschedule appointments easily.
* **Integrated Telemedicine Features :** Add a video conferencing feature to facilitate virtual consultations between patients and doctors.
* **User Feedback and Rating System :** Implement a feedback system that allows users to rate their experiences with doctors and provide comments.
* **AI-Powered Symptom Checker :** Implement an AI-driven symptom checker that helps users assess their health conditions based on their symptoms before.
* **Improved Analytics for Admin Dashboard :** Enhance the admin dashboard with analytics and reporting tools to track user engagement, appointment statistics, and feedback trends.

# 14 . CONCLUSION

In conclusion, the MediCareBook application successfully addresses the critical need for a user-friendly and secure platform that facilitates seamless communication between patients and healthcare providers, streamlining the appointment management process. Through careful planning, development, and testing, the project has implemented essential features such as user registration, doctor search, and appointment scheduling while ensuring compatibility and responsiveness. The team has successfully implemented core features such as user registration, login, doctor search, appointment scheduling, and user profiles. Despite identifying some areas for improvement and potential enhancements, the application demonstrates significant promise in enhancing healthcare accessibility and efficiency, ultimately contributing to better health outcomes for users. Moving forward, ongoing updates and feature expansions will ensure that MediCareBook continues to meet the evolving needs of its users in the dynamic healthcare landscape.

**GITHUB LINK:** [MediCareBook : Github](https://github.com/meghanagopinath-60/MediCareBook-Naan-Mudhalvan-Project)