**FULL STACK DEVELOPMENT PROJECT WITH MERN**

**PROJECT REPORT**

**BOOK A DOCTOR APP**

Team Details

NAME (TEAM LEADER) : SHANTHA PRIYA M

TEAM ID: 15474

NM ID : C3BC8F0D5251117872C43CE88E074D54

MAIL ID : [mshanthapriya@gmail.com](mailto:mshanthapriya@gmail.com)

NAME : NITHYA SREE K

NM ID : 2CCD0485F0816F7A69747C705AEEC7FC

MAIL ID : [kandapp1973@gmail.com](mailto:kandapp1973@gmail.com)

NAME : SANDHYA SHANKAR

NM ID: 0C8ACD15F35BA7382AF4582782B4F945

MAIL ID : sandhya.shankar.2002@gmail.com

NAME: SABARISRINATH R

NM ID : 6213B8665E534FC81CB499E31B43C5E7

MAIL ID : sabarisri1206@gmail.com

**Project Overview:**

The **Doctor Appointment Booking System** is a MERN stack-based web application developed for booking doctor appointments as part of the MongoDB course under the Naan Mudhalvan program. It enables users to register as patients or doctors, with role-based access for tailored functionality. Patients can browse doctor profiles, filter by specialization or location, and book appointments based on real-time availability. Doctors can manage schedules, view appointments, and update their availability. The system includes secure authentication, notifications for confirmations and reminders, and an intuitive UI designed with React.js. MongoDB is used for efficient data management, while Node.js and Express.js power the backend APIs. Deployed using cloud platforms, this project demonstrates a scalable and practical application of full-stack development principles.

**Features:**

**User Authentication**

* Secure login and registration for both patients and doctors with role-based access control.
* Passwords are hashed for security, and sessions are managed using JSON Web Tokens (JWT).
* Supports email verification and password recovery options for enhanced security.

**Doctor Management**

* Doctors can create and update profiles, including specialization, experience, and consultation fees.
* Manage availability by adding or editing time slots for patient appointments.
* View and manage booked appointments to streamline schedules effectively.

**Appointment Booking**

* Patients can search for doctors and book appointments based on availability.
* Reschedule or cancel appointments with automated updates in the doctor's schedule.
* Real-time notifications and reminders for both doctors and patients about upcoming appointments.

**Admin Panel**

* Admins can monitor the platform, manage users, and oversee doctor registrations.
* View and resolve issues related to appointments or user queries.
* Generate reports for system analytics and maintain data integrity.

**Responsive Design**

* Optimized for both desktop and mobile platforms with a clean and intuitive layout.
* Ensures seamless navigation and accessibility across various screen sizes.
* Utilizes modern UI/UX principles for an engaging user experience.

**Prerequisites**

To develop the **Book a Doctor** app, the following prerequisites are required:

1. **Node.js and npm**: Install Node.js for server-side JavaScript execution and npm for managing dependencies.
2. **Express.js**: Install Express to handle server-side routing and API development.
3. **MongoDB**: Set up MongoDB for storing user and appointment data.
4. **React.js**: Use React.js to build dynamic user interfaces for the app.
5. **Moment.js**: Install Moment.js for date manipulation in the app.
6. **Ant Design and Bootstrap**: Use these UI libraries for responsive components.
7. **Basic HTML, CSS, and JavaScript**: Knowledge required for structuring, styling, and adding interactivity to the app.

Dependencies:

1)Clone the repository and navigate to the project directory:



2) Install dependencies for both front-end and back-end



3)Start the Development Server



**Frontend**

**React.js, Bootstrap**

* The frontend is built with **React.js**, enabling the development of a dynamic and interactive user interface.
* **Bootstrap** and **React-Bootstrap** are used for responsive design, ensuring compatibility across different devices.
* Additional libraries like **Ant Design** and **MDB React UI Kit** enhance the UI components, providing ready-to-use elements for faster development.

**Backend**

**Node.js, Express.js**

* The backend leverages **Node.js** for scalable, event-driven server-side operations.
* **Express.js** is used as the web framework to handle routing, middleware, and API endpoints efficiently.
* APIs are designed to facilitate communication between the frontend and the database, ensuring secure and robust interactions.

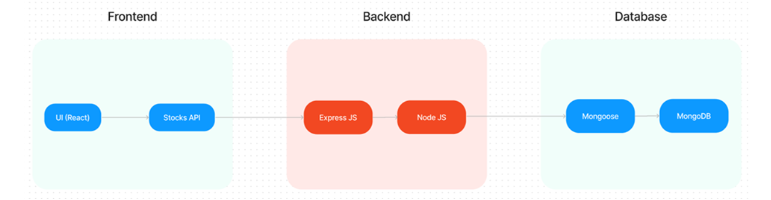
**Database:**

**MongoDB (NoSQL Database)**

* **MongoDB** serves as the database for storing structured and semi-structured data like user profiles, doctor schedules, and appointments.
* It provides a flexible schema design, allowing the application to scale easily as requirements evolve.
* **MongoDB Atlas** (cloud-based) ensures high availability, automatic backups, and enhanced security for the data.

**Technical Architecture**

The technical architecture of the Book a Doctor app follows a client-server model, with the front end acting as the client and the backend as the server. The front end, built with React.js, utilizes Bootstrap and Material UI for a responsive and interactive user interface, ensuring an optimal experience for admins, doctors, and patients. It incorporates the Axios library to enable seamless communication with the backend via RESTful APIs.



The backend, powered by Node.js and Express.js, handles server-side logic and API endpoints. MongoDB serves as the database, providing scalable and efficient storage for user profiles, doctor schedules, and appointments. This architecture ensures real-time communication, smooth data exchange, and reliable performance. Together, these components deliver a seamless experience for booking appointments and managing schedules.

**Working:**

This project was initiated using Create React App, which provides a solid foundation for React applications by setting up the development environment with minimal configuration. Below is a detailed explanation of the available scripts and their functionality:

1. npm start

* Purpose: Runs the application in development mode.
* How to Use: Execute the command in the project directory.
* Outcome:
  + Opens the app at <http://localhost:3000> in your browser.
  + Automatically reloads the page whenever you make changes to the source code.
  + Displays lint errors in the console for debugging purposes.

2. npm test

* Purpose: Launches the test runner in an interactive watch mode.
* How to Use: Execute the command in the project directory.
* Outcome:
  + Allows you to run automated tests for the app.
  + Supports continuous testing as you make changes to the code.

3. npm run build

* Purpose: Builds the application for production use.
* How to Use: Execute the command in the project directory.
* Outcome:
  + Generates a build folder containing the production-ready app.
  + Optimizes the app for best performance by minifying the code and including hashed filenames.
  + The app is now ready for deployment to a hosting service.

4. npm run eject

* Purpose: Provides full control over the app's configuration by ejecting it.
* How to Use: Execute the command in the project directory (only if necessary).
* Outcome:
  + Copies all configuration files (e.g., Webpack, Babel, ESLint) and dependencies into the project folder.
  + Allows manual modification of the configuration.
  + Note: This is a one-way operation and cannot be undone. Use it only if the default configuration is insufficient for your project needs.

Additional Information

Development Mode (npm start):

* Ideal for building and testing features during active development.
* Automatically detects changes in the code and updates the application in real time.

Production Build (npm run build):

* Ensures the application is optimized for deployment.
* Outputs a version of the app that is smaller, faster, and ready to be served from a server or content delivery network (CDN).

Ejecting (npm run eject):

* Recommended only for advanced use cases where you need to customize the app's build process or configurations.
* Once ejected, you are responsible for managing the configurations and dependencies.

**MongoDB Connection with Mongoose**

This code establishes a connection to MongoDB using **Mongoose** and the **dotenv** package to load environment variables. The connection string for MongoDB is retrieved from the MONGO\_DB environment variable. The connectToDB function uses mongoose.connect() with options like useNewUrlParser and useUnifiedTopology for a stable connection. If the connection is successful, it logs "Connected to MongoDB". In case of an error, it throws an exception with the error message, ensuring proper error handling during the connection process.

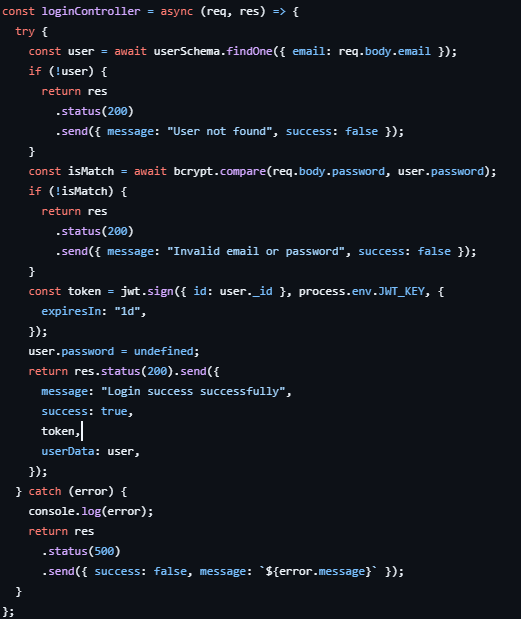


**User Registration Controller**

This code handles user registration by first checking if the email provided already exists in the database using userSchema.findOne(). If a user with the given email is found, it returns a message indicating that the user already exists. If the email is not found, it hashes the password using **bcryptjs** with a salt to ensure secure storage. The hashed password is then stored in the req.body.password. A new user is created with userSchema, saved to the database, and a success response is returned. If an error occurs during the process, a 500 status code is sent with the error message.



**User Login Controller**

****

This code handles user login by first checking if the user exists using the provided email. If the user is found, it compares the provided password with the stored hashed password using **bcryptjs**. If the password matches, a **JWT** token is generated with a 1-day expiration. The response includes the login success message, the token, and user data (excluding the password). If any error occurs during the process, it returns a 500 status code with the error message.

**JWT Authentication Middleware**

****

This code checks if a valid JWT token is present in the request's authorization header. If the header is missing, it responds with a 401 status and an error message. If the token is present, it extracts it and uses jwt.verify() to validate the token with the secret key (process.env.JWT\_KEY). If the token is invalid, it returns a 200 status with an error message. If the token is valid, the decoded user ID is added to the request body, and the next() function is called to proceed to the next middleware or route handler. In case of an error, a 500 status with an internal server error message is returned**.**

**Admin Controls:**

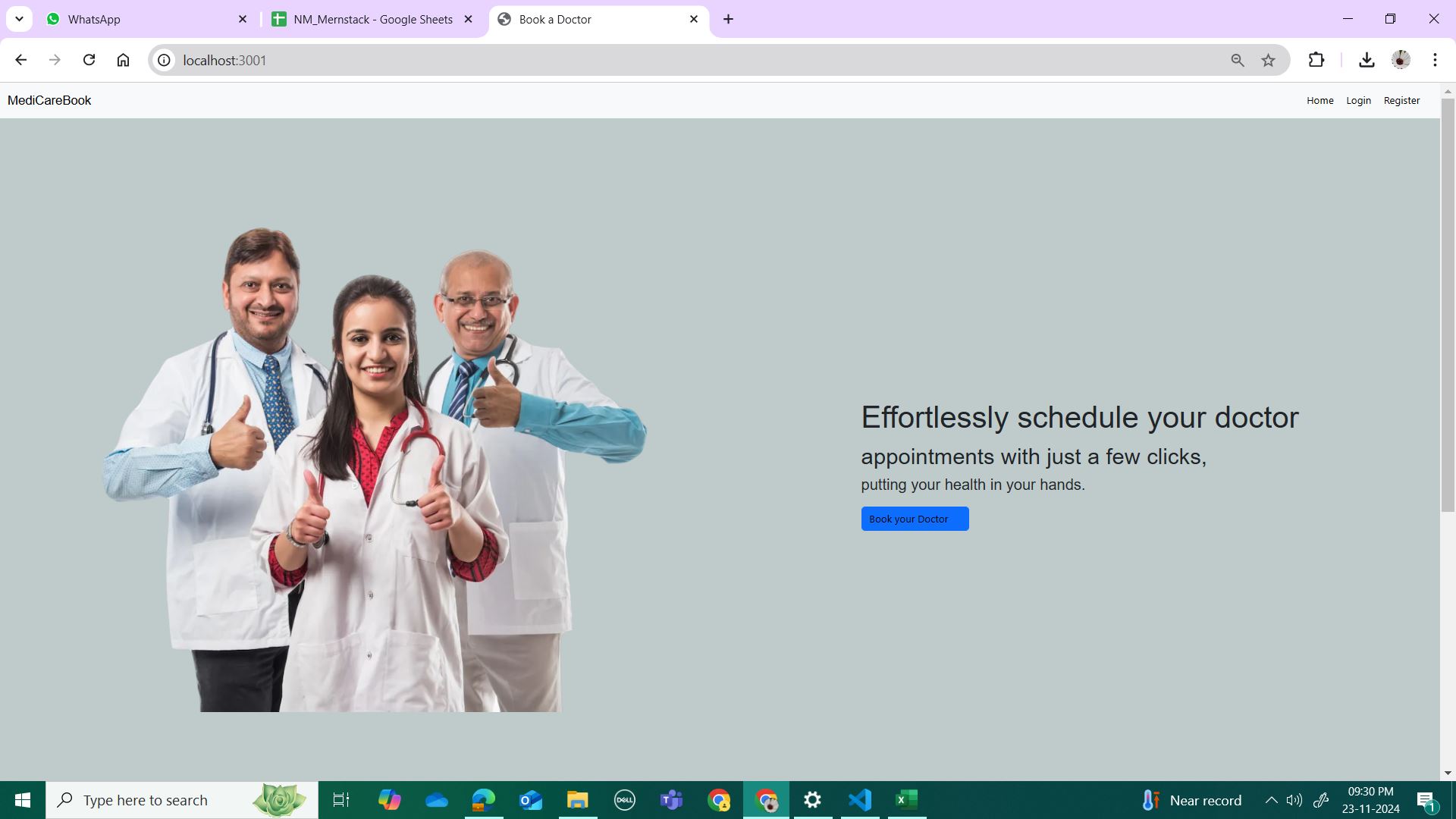
This code defines three controller functions for managing users and doctors. The getAllUsersControllers fetches all users from the database and returns the list with a success message. The getAllDoctorsControllers does the same for doctors. The getStatusApproveController updates the approval status of a doctor and sends a notification to the corresponding user, marking the user as a doctor if approved.

Similarly, the getStatusRejectController updates the doctor's rejection status and notifies the user accordingly. Both getStatusApproveController and getStatusRejectController ensure the user and doctor data are updated and saved, then return a success message with the updated data. In case of errors, they return a 500 status with an error message.

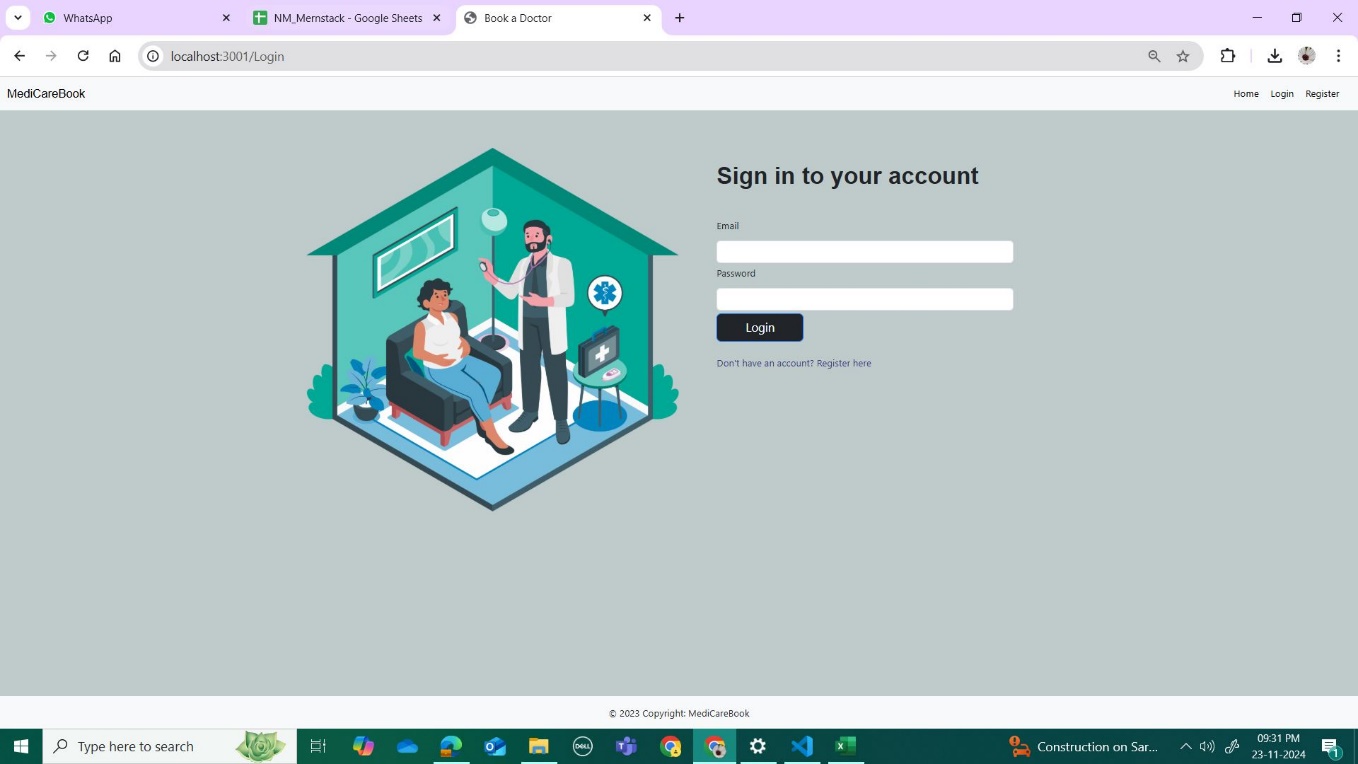
****

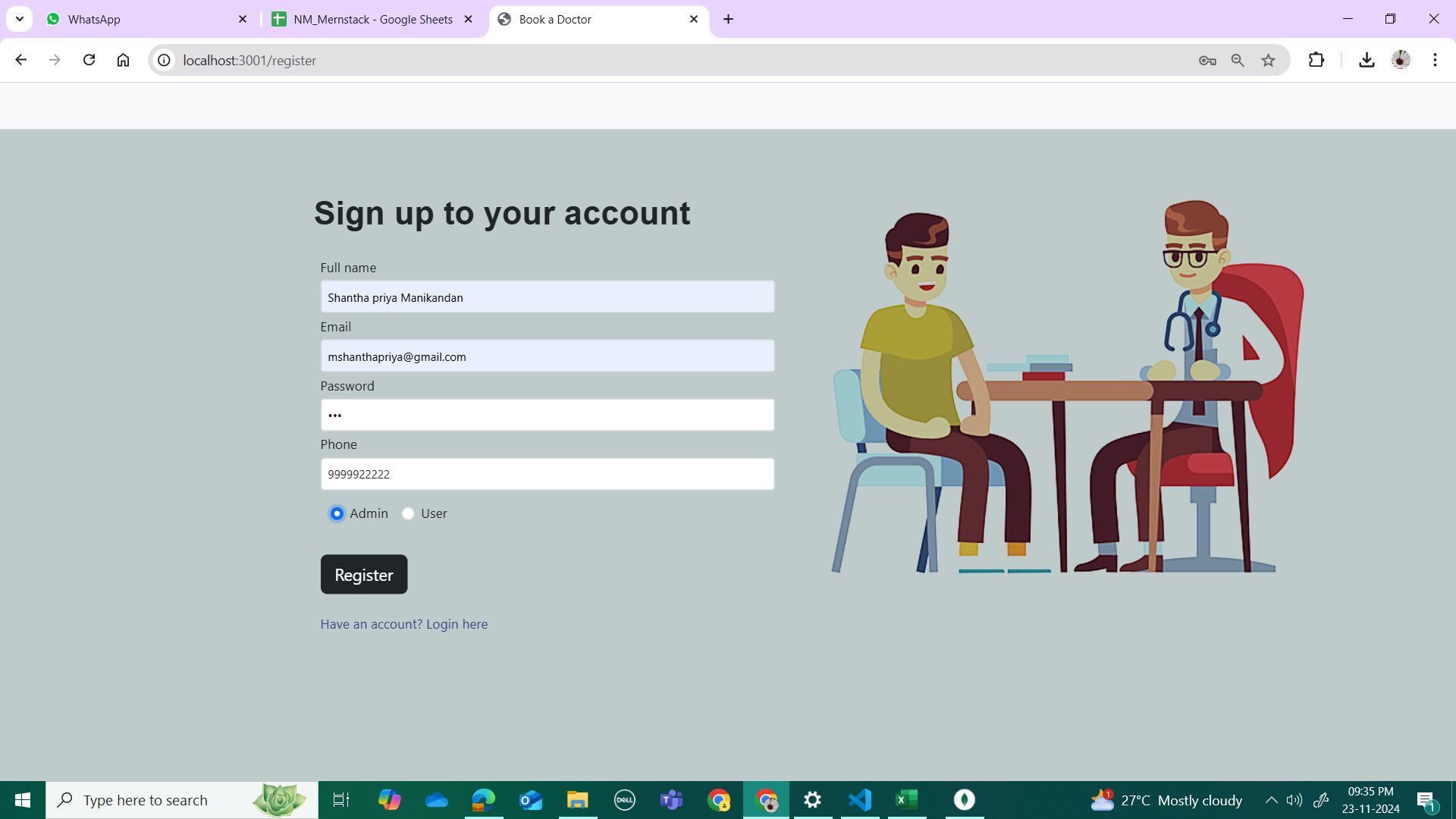
Book doctor app Screenshots:

Interface:

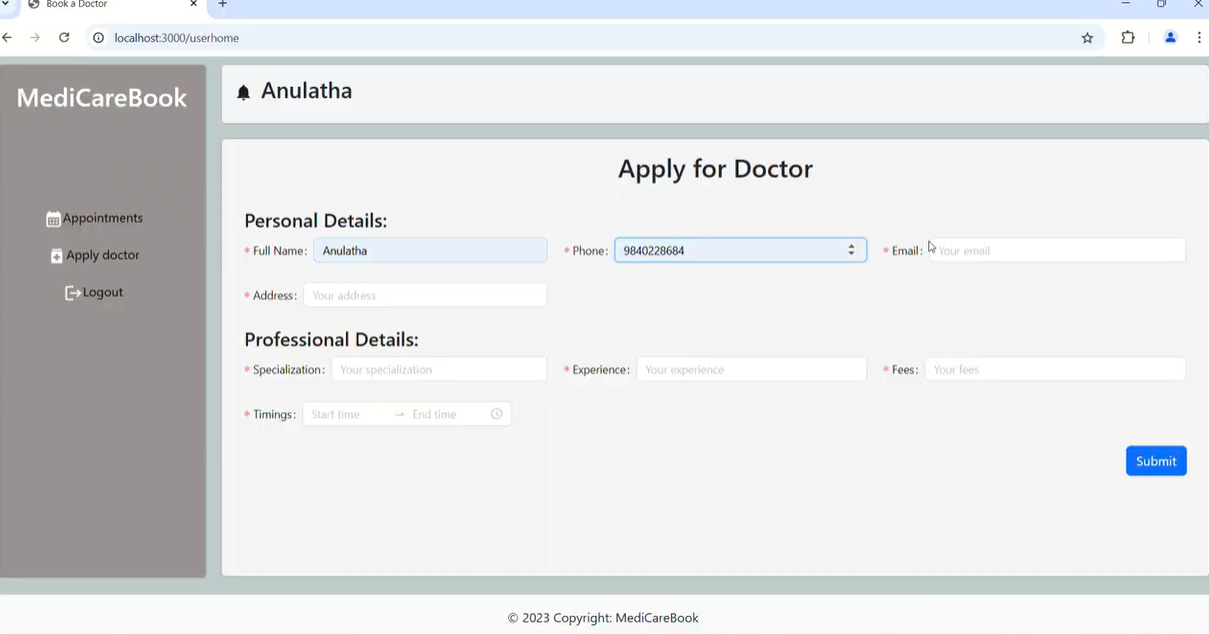


Login Page:

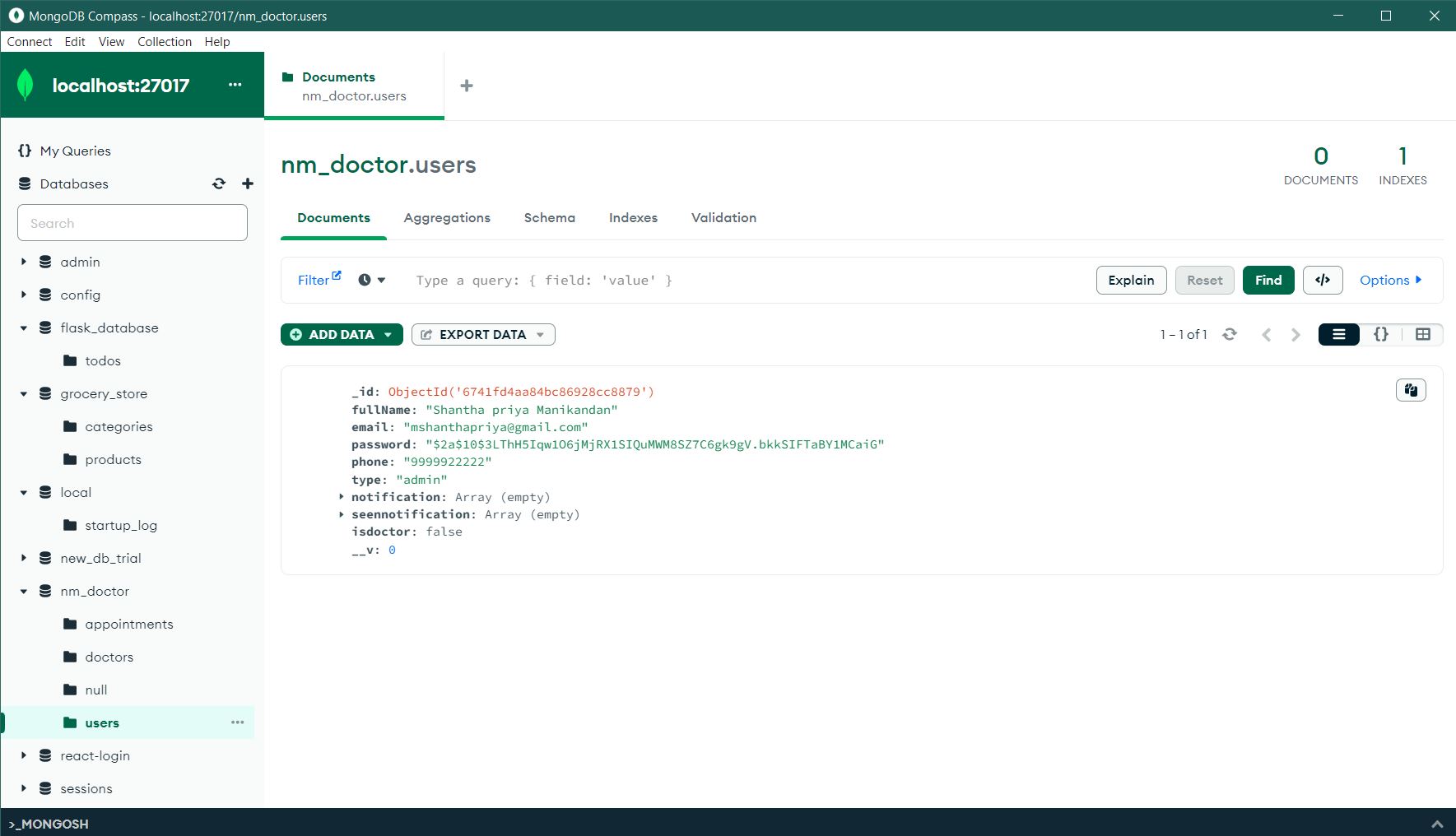




User Dashboard:



Mongodb Database:



**Demo Link:** <https://drive.google.com/file/d/1FwL52sovgUl8Wzp3qAnDXBRkoE37NoHg/view?usp=sharing>

**Github Link:**

<https://github.com/Sandhya-Shankar/NM_book_a_doctor_App>