DocSpot: Seamless Appointment Booking for Health

Team Members :

Team ID : LTVIP2025TMID60971

Team Leader : Battina Parimala

Team member : Emani Sumasri

Team member : Gandikota Geethanjali

Team member : Kallagunta Sathvika

Team member : Gunji Vani Akhila

PROJECT OVERVIEW:

Introduction

Healthcare is a critical sector where efficiency and accessibility directly impact people’s lives. One of the most common challenges patients face is booking medical appointments. Traditional methods, such as calling clinics or physically visiting hospitals, are often time-consuming, prone to errors, and inconvenient. Similarly, doctors and healthcare providers struggle with manual scheduling, overlapping bookings, and managing patient records effectively.

DocSpot is designed as a **seamless, full-stack web application** that solves these challenges by providing a modern, user-friendly appointment booking system. The platform allows patients to search for doctors by specialization, location, or availability and book appointments instantly. Patients can also view their booking history, receive reminders, and reschedule or cancel appointments easily.

For doctors, DocSpot simplifies appointment and patient management. Doctors can update their availability, approve or reject appointment requests, and access digital patient histories, which improves time management and overall service quality. Administrators benefit from system-wide monitoring, user management, and analytical reports that help optimize healthcare delivery.

Built using **React.js, Node.js, Express, and MySQL**, DocSpot ensures reliability, scalability, and security. By digitizing the appointment process, it reduces inefficiencies, enhances communication, and makes healthcare services more accessible to patients and providers alike.

Purpose:

The purpose of **DocSpot** is to provide a **seamless digital platform** that simplifies the process of booking, managing, and tracking healthcare appointments. In traditional systems, patients often face difficulties such as long waiting times, unclear schedules, and communication gaps with doctors. Similarly, healthcare providers struggle with manual record-keeping, scheduling conflicts, and time management issues.

DocSpot addresses these challenges by enabling patients to search for doctors based on specialization, availability, and location, and book appointments instantly. The platform ensures **transparency and convenience**, allowing patients to reschedule or cancel appointments without hassle. For doctors, it provides efficient schedule management, patient history access, and better control over their availability. Administrators can oversee the system, manage users, and generate insightful reports.

The ultimate goal of DocSpot is to **enhance healthcare accessibility, reduce inefficiencies, and improve patient-doctor communication** through a secure, user-friendly, and scalable web application.

Healthcare appointment booking has long been a challenge due to long waits, manual errors, and poor communication. **DocSpot** is a full-stack web application designed to solve these issues by providing a seamless, digital platform for patients, doctors, and administrators. Patients can easily search for doctors by specialization, location, or availability and book appointments instantly. Doctors benefit from organized schedule management and access to patient history, while administrators ensure smooth operations through monitoring and reporting.

Features:

DocSpot comes with a wide range of features designed to make healthcare appointment booking simple, efficient, and secure for patients, doctors, and administrators.

□ Patient Features

- **User Registration & Login** – Secure account creation and authentication.
- **Doctor Search & Filters** – Browse doctors by specialization, location, and availability.
- **Appointment Booking** – Instantly book, reschedule, or cancel appointments.
- **Booking History** – Track past and upcoming consultations.
- **Reminders & Notifications** – Automated alerts for upcoming appointments.

□ Doctor Features

- **Profile Management** – Create and update professional profiles.
- **Availability Scheduling** – Set working hours and update availability.
- **Appointment Management** – Approve, reject, or modify patient bookings.

- **Patient History Access** – View consultation records for better treatment.

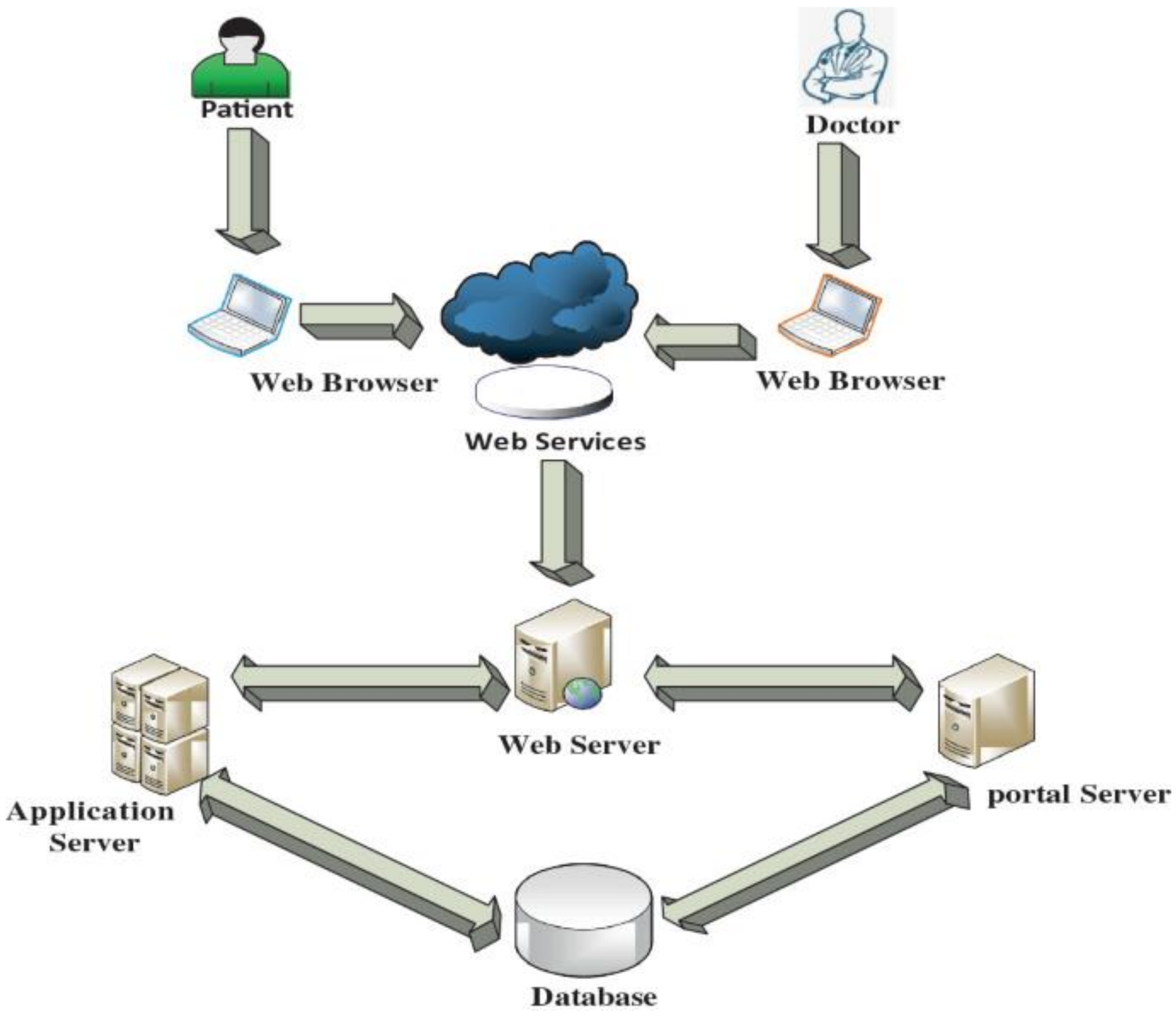
 Admin Features

- **User Management** – Manage patient and doctor accounts.
- **System Monitoring** – Track all appointments and resolve conflicts.
- **Reports & Analytics** – Generate insights on usage and performance.

 General Features

- **Responsive Design** – Works seamlessly on desktop and mobile devices.
- **Secure Authentication** – JWT-based login with encrypted passwords.
- **Scalable Database** – Efficient storage using MySQL.

ARCHITECTURE:



Frontend Architecture

The **frontend of DocSpot** is developed using **React.js**, chosen for its **component-based architecture, reusability, and fast rendering** with the Virtual DOM. The design follows a **modular structure**, ensuring scalability and ease of maintenance. The application is built with **Vite** for faster development and optimized builds, while **Tailwind CSS** is used for styling and responsiveness.

1. Component-Based Structure

- **Reusable Components** such as Navbar, Footer, Buttons, and Forms ensure consistency across the application.
- **Page Components** like Home, Patient Dashboard, Doctor Dashboard, and Admin Panel are separated for better organization.

- **State Management** is handled using React's useState and useContext hooks, while larger states (like authentication or appointments) can use **Redux/Context API** for global management.

2. Routing

- Implemented using **React Router**.
- Key routes include:
  - / → Homepage
  - /login & /register → Authentication pages
  - /dashboard/patient → Patient Dashboard
  - /dashboard/doctor → Doctor Dashboard
  - /dashboard/admin → Admin Panel

3. API Integration

- The frontend communicates with the backend via **RESTful APIs**.
- Fetch or Axios is used for API calls (e.g., booking an appointment, fetching doctor details, updating profiles).
- **JWT tokens** are stored securely in local storage or cookies for authentication.

4. UI/UX Design

- Built with **responsive layouts** (mobile-first design).
- **Tailwind CSS** ensures fast, modern, and customizable styling.
- Accessibility features like **form validation, alerts, and dark/light mode** improve usability.

5. Frontend Flow

1. User logs in or registers → Token generated and stored.
2. Based on role (Patient/Doctor/Admin), different dashboards load.
3. API requests handle booking, history, availability, and reports.
4. UI updates dynamically without full-page reloads thanks to React's SPA (Single Page Application) model.

Backend Architecture

The **backend of DocSpot** is built using **Node.js** with the **Express.js framework**, providing a lightweight, fast, and scalable environment for handling healthcare appointment workflows. The architecture follows the **RESTful API design pattern**, ensuring clear separation between frontend and backend and easy communication through HTTP requests.

1. Server Layer (Express.js)

- **Express.js** acts as the backbone of the backend, handling **routing, middleware, and request/response cycles**.
- Each resource (users, doctors, appointments) is mapped to a dedicated **API endpoint**.
- Middleware is used for:
  - **Authentication & Authorization** (JWT tokens)
  - **Input Validation** (ensuring correct data formats)
  - **Error Handling & Logging**

2. Controller Layer

- Contains the **business logic** of the application.
- Example:
  - Appointment controller → handles booking, rescheduling, or cancellation.
  - Doctor controller → manages availability and profile updates.
  - User controller → handles registration, login, and authentication.

3. Service Layer

- Contains **reusable functions** that interact with the database.
- Keeps controllers clean by abstracting database queries and transformations.
- Example: appointmentService.createAppointment() manages appointment creation logic before saving it to the DB.

4. Database Layer

- **MySQL** database stores structured data: users, doctors, appointments, and medical history.
- **Sequelize/Knex.js ORM** (optional) for simplified query handling and schema management.
- Relationships:
  - One-to-Many → Doctor ↔ Appointments
  - One-to-Many → Patient ↔ Appointments

5. Security & Authentication

- **JWT-based authentication** for session management.
- **Role-based access control (RBAC)** to differentiate patient, doctor, and admin privileges.
- **Password encryption (bcrypt.js)** for secure user credentials.

6. Scalability & Deployment

- Backend designed as **modular APIs** for scalability.
- Deployed on **Heroku/Render/AWS**, ensuring availability and performance.
- Can be extended into **microservices architecture** if scaled for hospitals with large user bases.

Database Architecture

1. The database uses **MongoDB** for flexible, document-oriented storage.
2. Data is structured to manage patients, doctors, and appointments efficiently.
3. Relationships like patient–doctor and doctor–appointment are mapped via references.
4. **Mongoose** handles schema validation and smooth data interaction.
5. Indexing ensures fast queries for search and availability checks.
6. The architecture is scalable, secure, and optimized for healthcare workflows

SETUP INSTRUCTIONS

Prerequisites

To run the **DocSpot** project, a few software dependencies are required. First, **Node.js** and **npm** are essential for managing the backend server and installing required packages. **Express.js** is used as the web framework for handling routes and APIs. For the database, **MongoDB** is needed to store and manage patient, doctor, and appointment data. **Mongoose** helps in modeling and validating data. On the frontend, **React.js** is required to build an interactive user interface, along with supporting tools like **Vite** or **Webpack** for bundling. Additionally, tools like **Postman** (for API testing) and **Git** (for version control) are recommended.

Installation

Follow these steps to set up and run the **DocSpot** project on your local system:

1. **Clone the Repository**

```
git clone https://github.com/your-username/docspot.git

cd docspot
```

2. **Install Dependencies**
   - **For backend**

```
cd backend

npm install
```

   - **For frontend**

```
cd frontend

npm install
```

3. **Set Up Environment Variables**

- In the **backend** folder, create a .env file and add:

```
PORT=5000

MONGO_URI=your_mongodb_connection_string

JWT_SECRET=your_secret_key
```

4. **Run the Project**

- Start backend server:

```
npm run dev
```

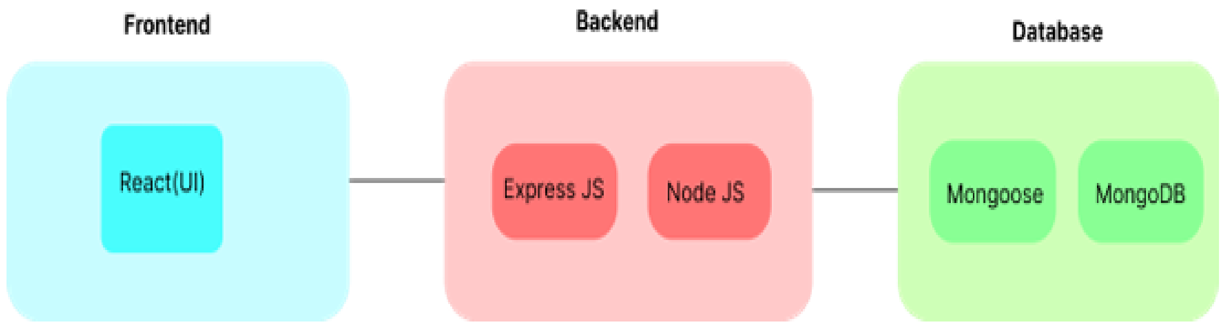
- Start frontend server:

```
npm run dev
```

5. **Access the Application**

- Open your browser and go to:

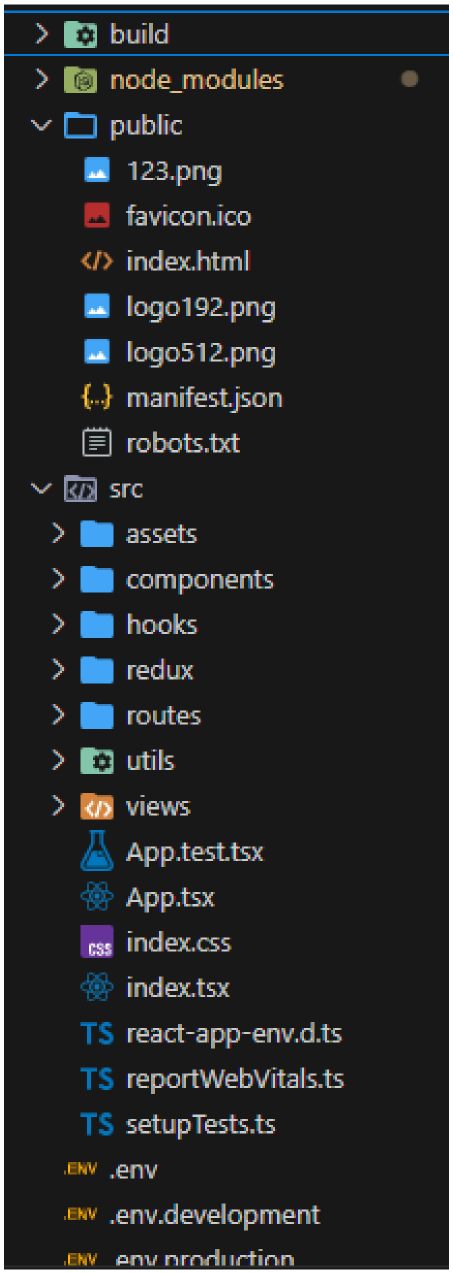
```
http://localhost:5173
```



FOLDER STRACTURE

Client: React Frontend Structure

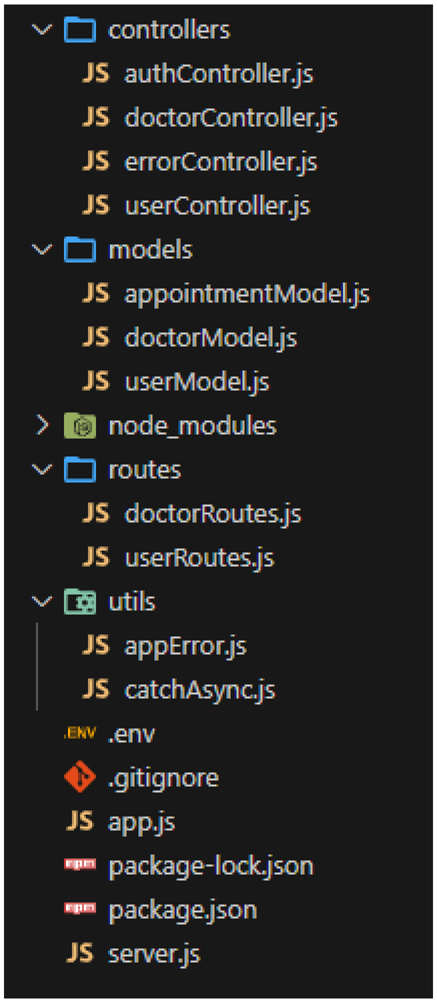
The client-side of **DocSpot** is built using **React.js**, ensuring a modular, scalable, and efficient user interface. The structure follows a **component-based architecture**, where each UI section (Navbar, Forms, Doctor Cards, etc.) is developed as an independent component. This approach makes the application reusable, maintainable, and easy to debug.



```
> build
> node_modules
v public
    123.png
    favicon.ico
    index.html
    logo192.png
    logo512.png
    manifest.json
    robots.txt
v src
  > assets
  > components
  > hooks
  > redux
  > routes
  > utils
  > views
    App.test.tsx
    App.tsx
    index.css
    index.tsx
    react-app-env.d.ts
    reportWebVitals.ts
    setupTests.ts
  .env
  .env.development
  .env.production
```

Server (Backend) Organization – Node.js with Express.js

The Node.js backend is organized in a clean and modular way to ensure scalability and maintainability. The entry point (server.js) initializes the application, connects it to the MongoDB database, and starts the server. Routes are defined to handle incoming API requests, which are then forwarded to controllers. Controllers contain the business logic and communicate with models, which define the database schema and structure. Middleware and utility files are used for tasks like authentication, validation, and error handling, while services help in managing reusable functionalities. This separation of concerns makes the backend efficient, secure



```
controllers
    authController.js
    doctorController.js
    errorController.js
    userController.js
models
    appointmentModel.js
    doctorModel.js
    userModel.js
node_modules
routes
    doctorRoutes.js
    userRoutes.js
utils
    appError.js
    catchAsync.js
.env
.gitignore
app.js
package-lock.json
package.json
server.js
```

RUNNING THE APPLICATION

To run the application, make sure Node.js and MongoDB are installed and running on your system. First, open the backend folder and run npm install to install dependencies, then start the server using npm start. This will launch the Node.js Express backend on the configured port (default: http://localhost:5000). Next, go to the frontend folder, run npm install, and start the React app with npm run dev. The frontend will usually run on http://localhost:5173 and will connect with the backend API for data. This ensures smooth communication between client, server, and database.

Frontend (React app)

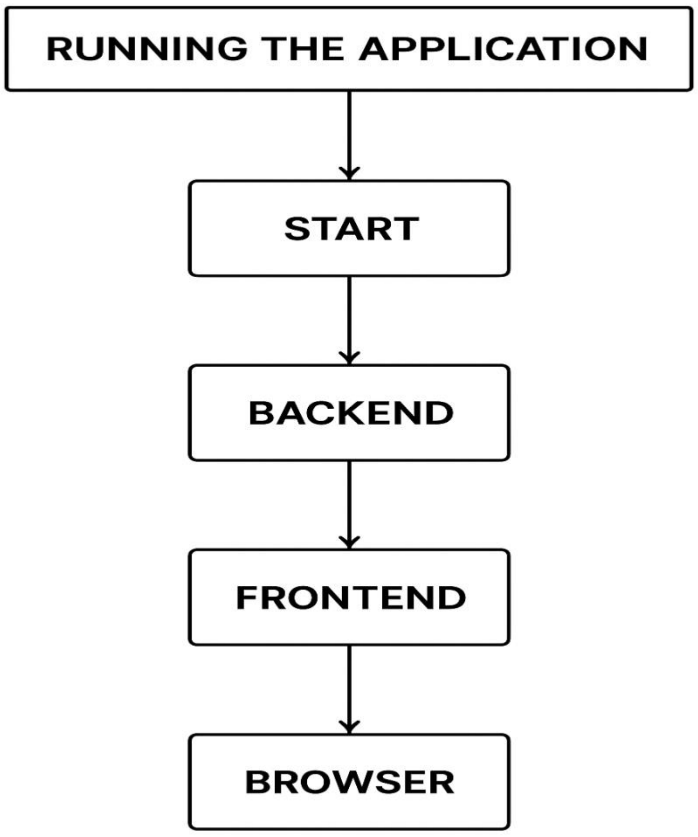
```
cd client

npm start
```

Backend (Node.js/Express app)

```
cd server

npm start
```

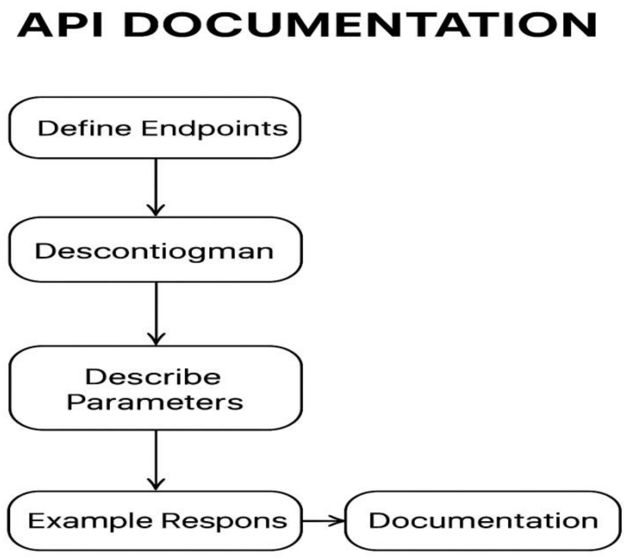


API DOCUMENTATION

The backend exposes a set of RESTful APIs that allow smooth communication between the frontend and the database. Each endpoint is designed with a clear purpose, supporting operations such as creating, reading, updating, and deleting data. APIs use **HTTP methods** like GET for fetching data, POST for inserting new records, PUT for updating existing ones, and DELETE for removing entries. Parameters are passed either through the request body (for sensitive data like login credentials), query strings, or URL paths (for specific resource identification).

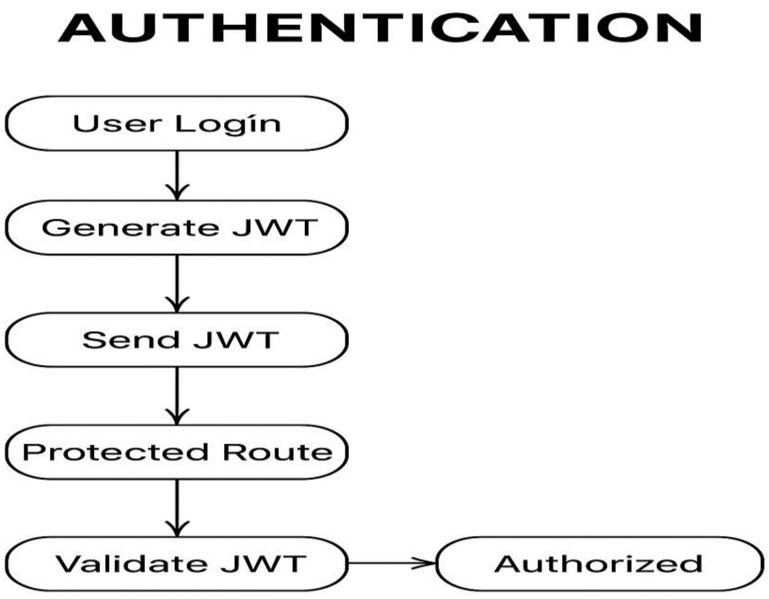
Responses are returned in **JSON format**, containing either the requested data or status messages indicating success or failure. For example, a GET /patients/:id endpoint would return patient details, while a POST /appointments would create a new booking and respond with a confirmation message. Proper error handling ensures that the APIs provide meaningful feedback, such as "User not found" or "Invalid input."

These documented APIs enable the frontend React client to interact seamlessly with the backend, ensuring reliable data flow across users, doctors, and administrators.

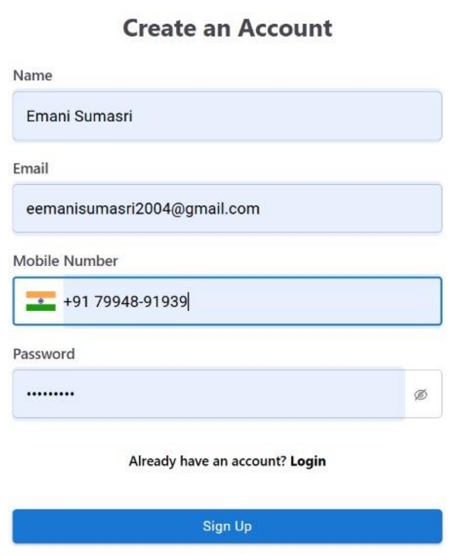


AUTHENTICATION

In this project, authentication is implemented to verify the identity of users before granting access, while authorization ensures that only users with the correct permissions can perform specific actions. When a user logs in with valid credentials, the backend generates a **JSON Web Token (JWT)**, which is sent back to the client and securely stored. For every request to protected routes, the client must include this token in the request header. The server uses middleware to validate the token and confirm the user's identity. If valid, access is granted; if not, the request is rejected. Authorization is enforced by assigning roles (such as user, doctor, or admin), where each role has different levels of access to resources. This approach provides a secure, stateless mechanism using tokens instead of traditional session-based logins, ensuring data protection and controlled access throughout the system.



USER INTERFACE


Create an Account

Name

Emani Sumasri

Email

eemanisumasri2004@gmail.com

Mobile Number

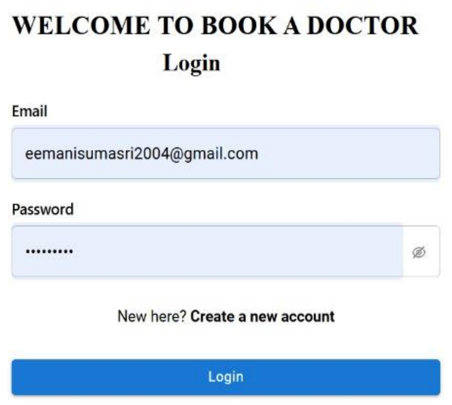
+91 79948-91939

Password

•••••••••

Already have an account? **Login**

Sign Up


WELCOME TO BOOK A DOCTOR

Login

Email

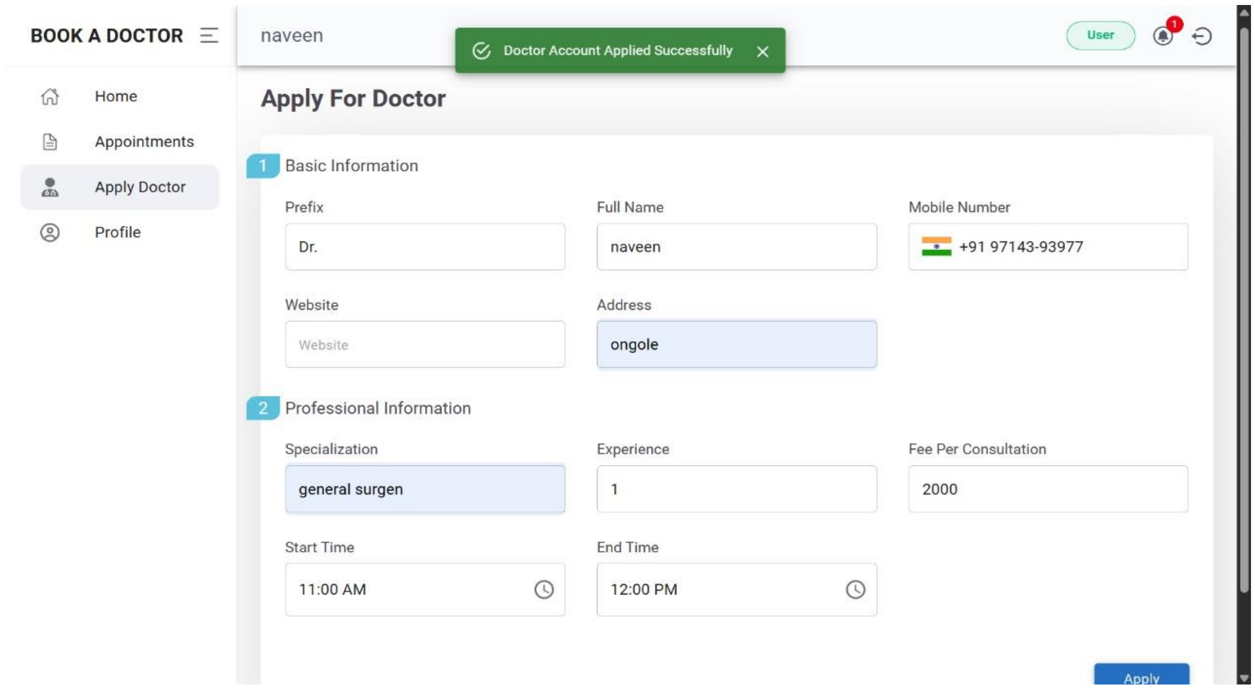
eemanisumasri2004@gmail.com

Password

•••••••••

New here? **Create a new account**

Login


BOOK A DOCTOR

- Home
- Appointments
- Apply Doctor
- Profile

naveen

Doctor Account Applied Successfully

User

Apply For Doctor

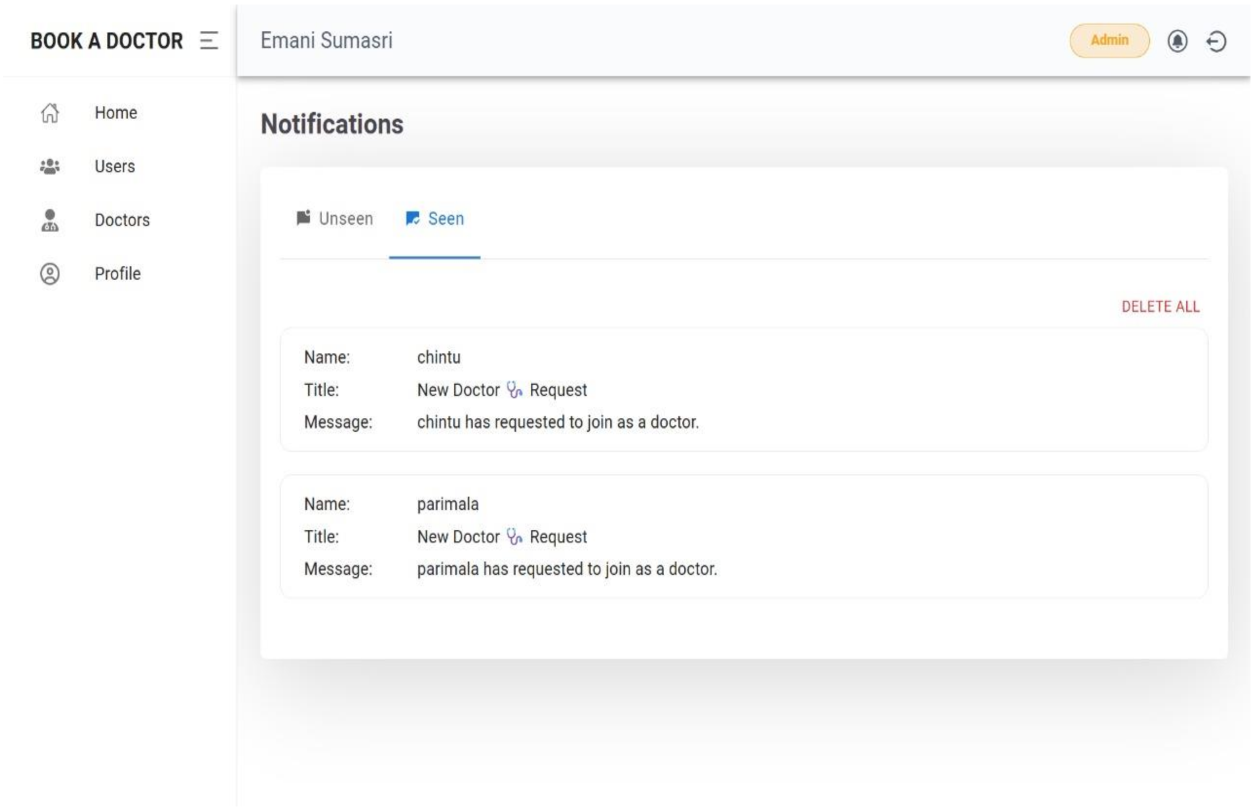
1 Basic Information

Prefix	Full Name	Mobile Number
Dr.	naveen	+91 97143-93977
Website	Address	
Website	ongole	

2 Professional Information

Specialization	Experience	Fee Per Consultation
general surgen	1	2000
Start Time	End Time	
11:00 AM	12:00 PM	

Apply


BOOK A DOCTOR

- Home
- Users
- Doctors
- Profile

Emani Sumasri

Admin

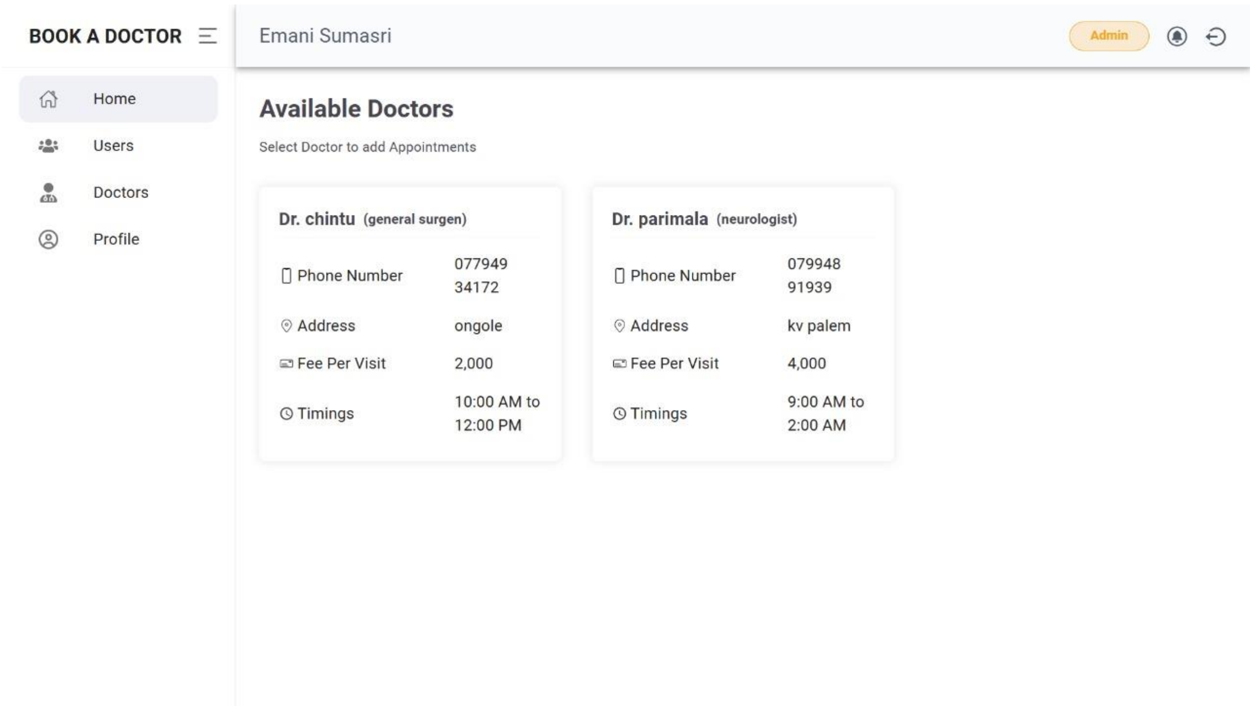
Notifications

Unseen Seen

DELETE ALL

Name:	chintu
Title:	New Doctor Request
Message:	chintu has requested to join as a doctor.

Name:	parimala
Title:	New Doctor Request
Message:	parimala has requested to join as a doctor.


BOOK A DOCTOR

Emani Sumasri

Admin

Home

Users

Doctors

Profile

Available Doctors

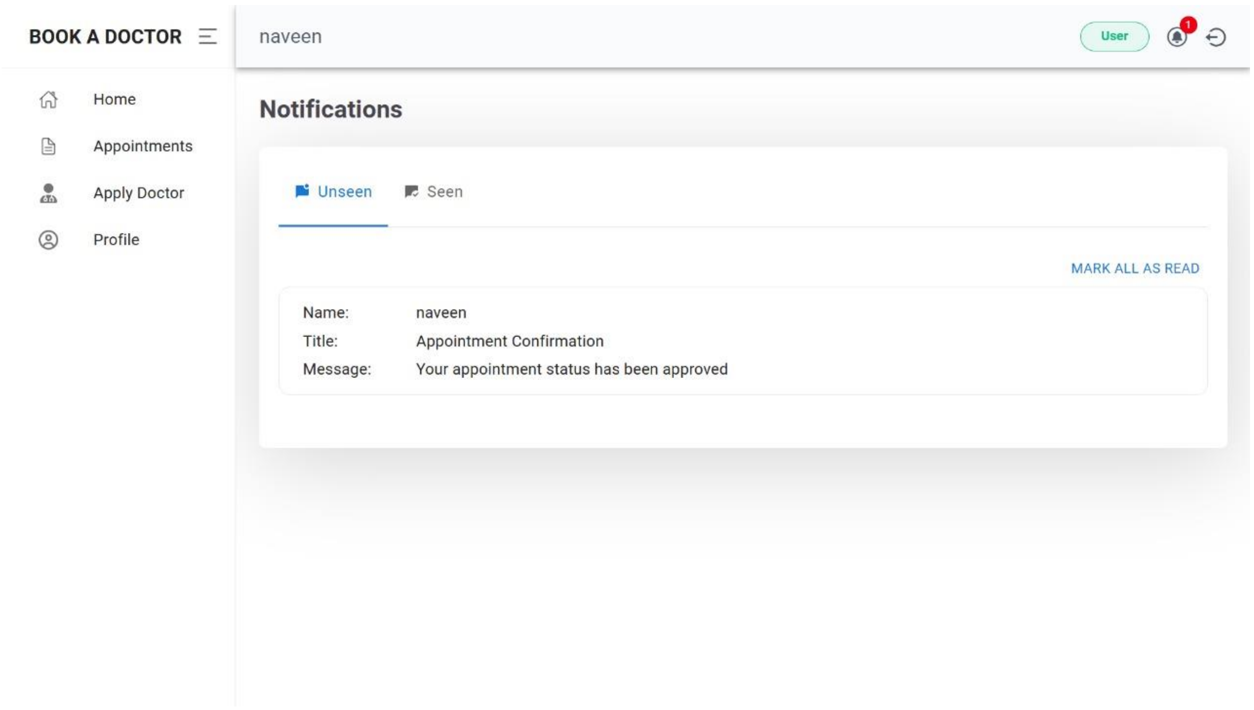
Select Doctor to add Appointments

Dr. chintu (general surgen)

Phone Number	077949 34172
Address	ongole
Fee Per Visit	2,000
Timings	10:00 AM to 12:00 PM

Dr. parimala (neurologist)

Phone Number	079948 91939
Address	kv palem
Fee Per Visit	4,000
Timings	9:00 AM to 2:00 AM


BOOK A DOCTOR

naveen

User

1

Home

Appointments

Apply Doctor

Profile

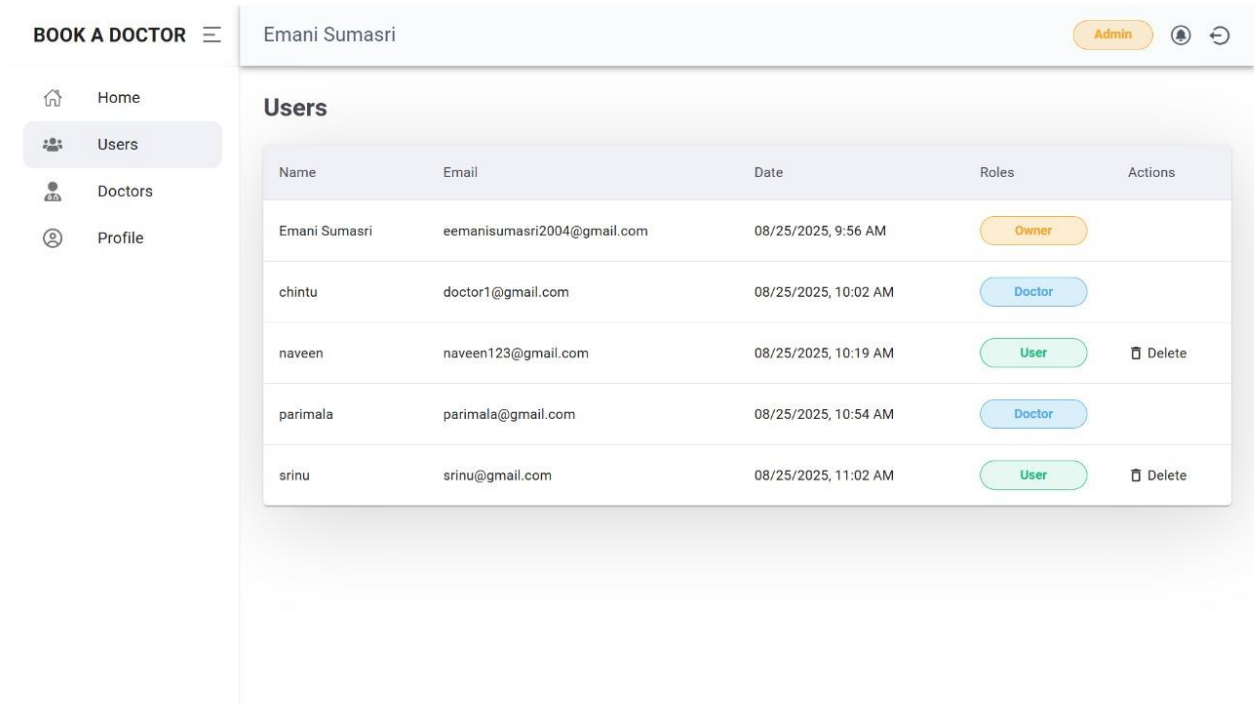
Notifications

Unseen

Seen

MARK ALL AS READ

Name:	naveen
Title:	Appointment Confirmation
Message:	Your appointment status has been approved


BOOK A DOCTOR

Emani Sumasri

Admin

Users

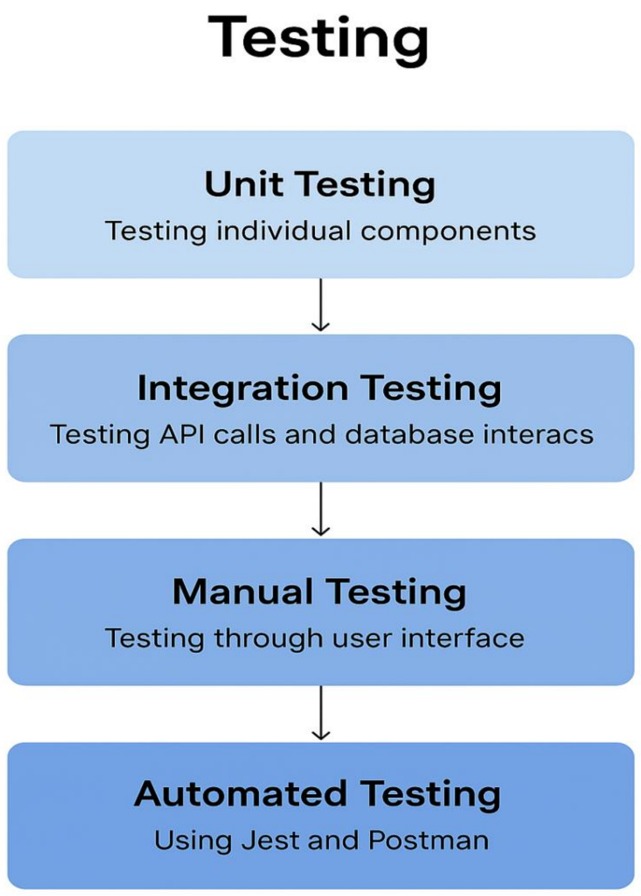
Name	Email	Date	Roles	Actions
Emani Sumasri	eemanisumasri2004@gmail.com	08/25/2025, 9:56 AM	Owner	
chintu	doctor1@gmail.com	08/25/2025, 10:02 AM	Doctor	
naveen	naveen123@gmail.com	08/25/2025, 10:19 AM	User	Delete
parimala	parimala@gmail.com	08/25/2025, 10:54 AM	Doctor	
srinu	srinu@gmail.com	08/25/2025, 11:02 AM	User	Delete

TESTING

The testing strategy for this project follows a layered approach to ensure both functionality and reliability:

1. **Unit Testing** – Individual functions, components, and modules are tested in isolation. For the backend, tools like **Jest** or **Mocha + Chai** are used. On the frontend, **React Testing Library** verifies component behavior.
2. **Integration Testing** – Focuses on testing the interaction between different modules (e.g., API routes with database queries, or frontend forms communicating with backend endpoints).
3. **End-to-End (E2E) Testing** – Ensures the entire flow works as expected from the user's perspective. **Cypress** or **Playwright** is used to simulate real user actions (like booking an appointment).
4. **Manual Testing** – QA or developers test the app manually for edge cases, UI/UX validation, and exploratory testing.
5. **Automation & CI/CD** – Automated test suites are integrated into pipelines (e.g., GitHub Actions) so tests run whenever new code is pushed, reducing bugs before deployment.

So, the project uses **Jest, React Testing Library, Mocha/Chai, and Cypress** as the primary testing tools.

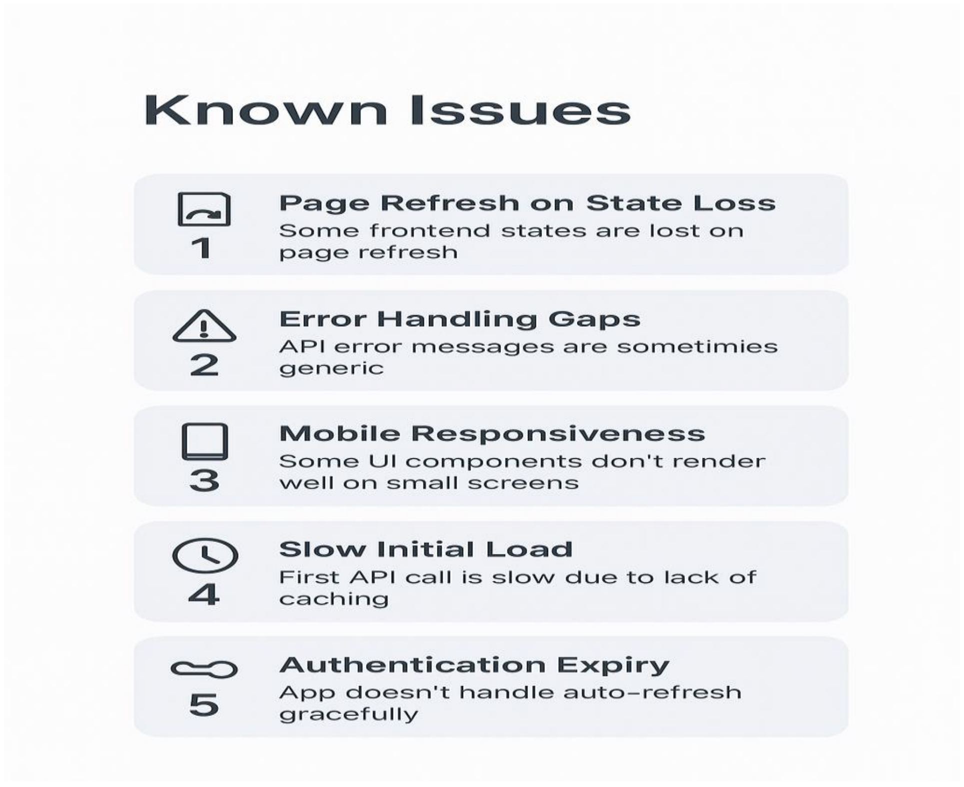

Testing
Unit Testing
Testing individual components
Integration Testing
Testing API calls and database interacs
Manual Testing
Testing through user interface
Automated Testing
Using Jest and Postman

KNOWN ISSUES

Despite testing and quality checks, the project currently has a few limitations and bugs that developers and users should be aware of:

1. **Page Refresh on State Loss** – Some frontend states (like selected filters or unsaved form data) are lost if the page is refreshed, as persistence is not fully implemented.
2. **Error Handling Gaps** – API error messages are sometimes generic (e.g., "Something went wrong") instead of providing user-friendly feedback.
3. **Mobile Responsiveness** – A few UI components may not render perfectly on smaller screens, requiring CSS adjustments.

4. **Slow Initial Load** – The first API call (especially when fetching large datasets) may take longer due to the absence of caching.
5. **Authentication Expiry** – JWT tokens expire after a set duration, and the app doesn’t always handle auto-refresh gracefully, forcing users to re-login.



Known Issues

1 **Page Refresh on State Loss** Some frontend states are lost on page refresh

2 **Error Handling Gaps** API error messages are sometimies generic

3 **Mobile Responsiveness** Some UI components don't render well on small screens

4 **Slow Initial Load** First API call is slow due to lack of caching

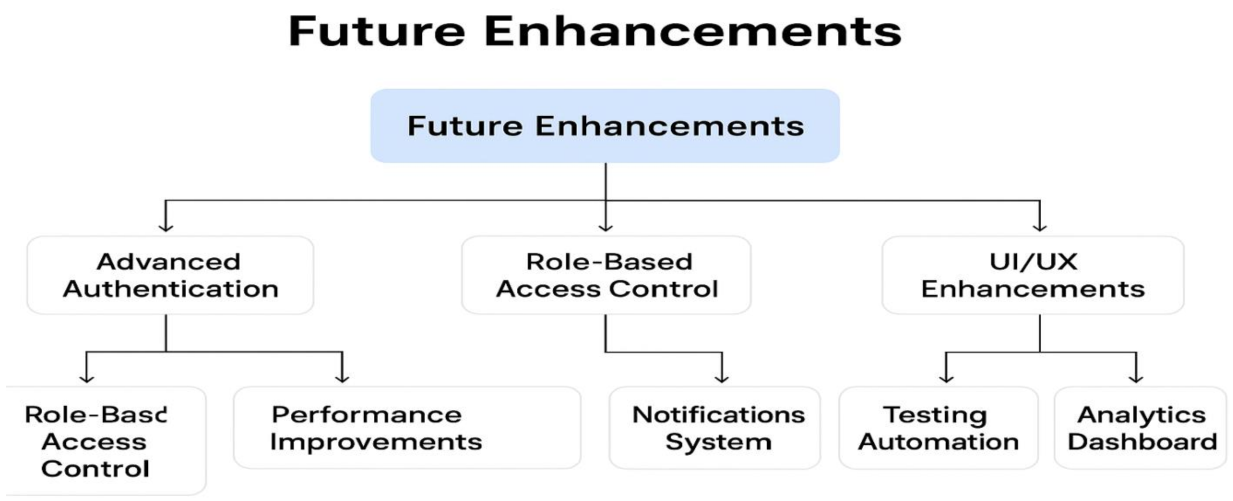
5 **Authentication Expiry** App doesn't handle auto-refresh gracefully

FEATURES – ENHANCEMENTS

Future Enhancements in this project can include several improvements to increase functionality, scalability, and user experience. Some potential features are:

1. **Advanced Authentication** – Integration of OAuth (Google, GitHub) for social login.
2. **Role-Based Access Control** – Fine-grained permissions for admin, user, and guest roles.
3. **Scalability** – Migration to cloud deployment with Docker and Kubernetes support.
4. **Performance Improvements** – Use of caching mechanisms like Redis for faster data retrieval.

5. **UI/UX Enhancements** – Adding dark mode, responsive design upgrades, and accessibility improvements.
6. **Notifications System** – Real-time email/SMS/push notifications for key events.
7. **Testing Automation** – Full integration with CI/CD pipelines and automated test coverage.
8. **Analytics Dashboard** – Adding charts and reports for data visualization.


Future Enhancements
Future Enhancements
Advanced Authentication
Role-Based Access Control
UI/UX Enhancements
Role-Basd Access Control
Performance Improvements
Notifications System
Testing Automation
Analytics Dashboard