Doctor Appointment Booking Application

Submitted by: Omprkash Kumar

Roll Number: 2214502456

Program: BCA

Session: June/2022

Guide: Mrs. Dipali Borade

Organization: NeoSoft

Project Duration: July 2025 – August 2025

Table of Contents

25. Data Dictionary

1. Introduction & Objectives

A web-based system designed to simplify and digitize the doctor appointment booking process.

Objectives:
- Enable patient registration and login
- List/search doctors based on specialization
- Allow real-time booking and availability tracking
- Provide admin controls to manage doctors, patients, and appointments

2. System Analysis

This system addresses inefficiencies in traditional appointment systems through real-time booking, better record management, and accessibility via mobile/desktop.

3. Identification of Need

Manual systems suffer from mismanagement, double bookings, and lack of transparency. Patients and clinics benefit from automation and digitization.

4. Preliminary Investigation

Analysis was conducted via:
- Informal interviews with doctors and patients

- Study of platforms like Practo, Zocdoc: A comparative analysis of existing healthcare platforms such as **Practo** and **Zocdoc** was performed. These platforms provided insights into best practices, common features, and user expectations, which helped in defining the scope and feature set of the proposed system.

5. Feasibility Study

- Technical Feasibility: MERN Stack supports scalable web apps.
- Operational Feasibility: System is simple to operate for users and admins. - Economic Feasibility: Minimal cost using open-source tools and free hosting (Render/Vercel).

Technical Feasibility:
The system is built using the **MERN Stack** (MongoDB, Express.js, React.js, Node.js), which is widely adopted for building modern, scalable, and high-performance web applications. The technology stack is well-supported and suitable for both current and future expansion needs.

Operational Feasibility:
The system is designed with a user-friendly interface for both patients and administrators.

Basic training or prior technical knowledge is not required, making it easy to operate and manage on a day-to-day basis.

Economic Feasibility:
Development and deployment costs were kept minimal by leveraging **open-source technologies** and **free-tier hosting services** such as **Render** and **Vercel**. This makes the system financially feasible, especially for academic and small-scale deployment scenarios.

6. Project Planning

Adopted Agile methodology with 2-week sprints for feature rollout and feedback.

Sprint-Based Development:
Each sprint included planning, development, testing, and review phases. At the end of every sprint, the system was evaluated, and feedback was collected to refine upcoming tasks.

Continuous Feedback and Improvement:
Regular feedback loops from test users (doctors and patients) allowed for timely improvements and feature enhancements, aligning the project closely with real-world requirements.

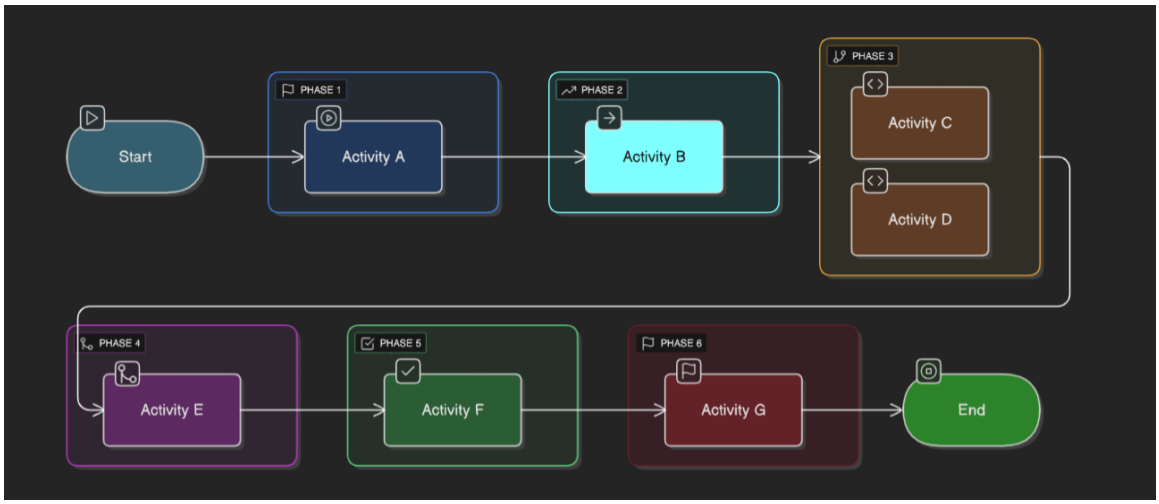
Task Management:
Project tracking tools (such as Trello or GitHub Projects) were used to manage tasks, set priorities, and monitor progress throughout the development lifecycle.

7. Project Scheduling (PERT Chart & Gantt Chart)

PERT and Gantt charts are used to track the flow and timing of development activities.

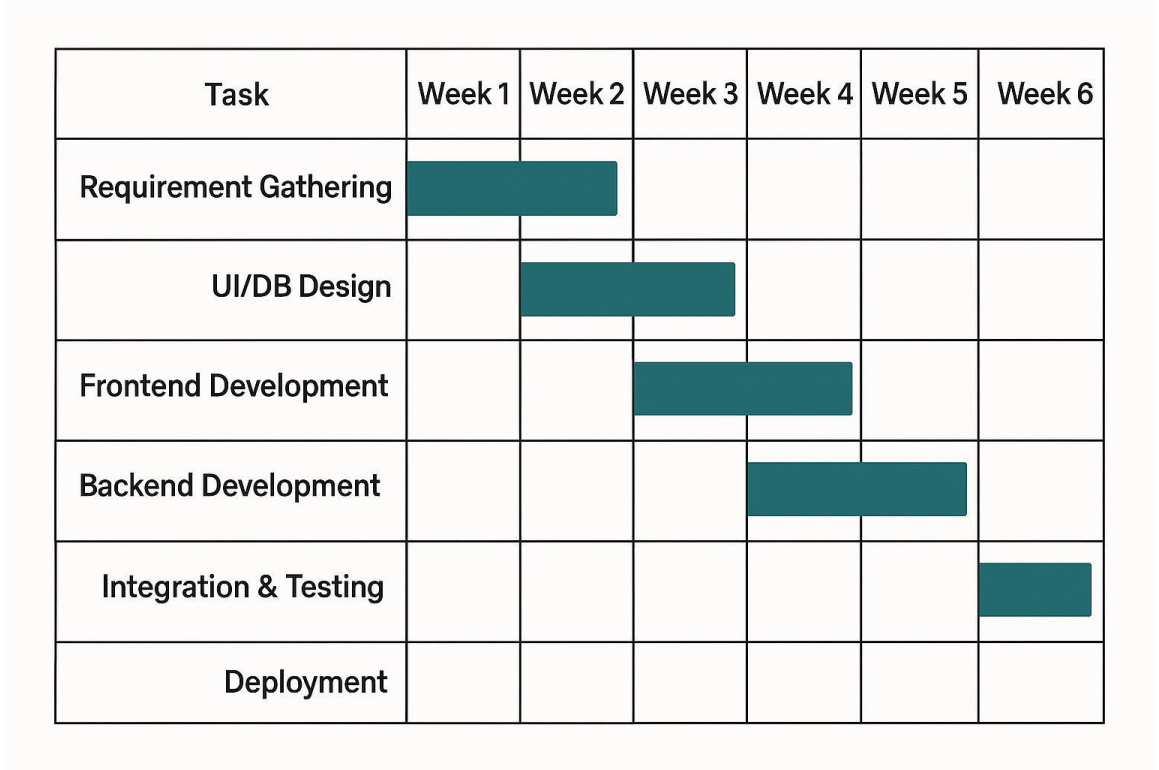
What It Does: It shows the sequence of tasks and dependencies, helping you find the **critical path** (minimum time to complete project)

a) Visual PERT Chart Layout



2. Gantt Chart

What It Shows: Timeline view of tasks — great for showing when and how long each task runs.



8. Software Requirement Specifications (SRS)

Includes:

8.1 Functional Requirements

- **User Registration and Login** (with role-based access for Patients, Doctors, and Admins)
- **Appointment Booking and Cancellation**
- **Doctor Availability Management**
- **Admin Dashboard for User and Appointment Oversight**
- **Email Notifications and Appointment Reminders**

8.2 Non-Functional Requirements

- **Performance:** Fast and responsive UI across devices.
- **Security:** JWT-based authentication, role-based access control, and HTTPS deployment.
- **Scalability:** Built using the MERN stack to support future feature additions.
- **Availability:** Deployed on cloud hosting (Render/Vercel) with high uptime using free-tier services.

8.3 User Roles

- **Patient:**
  Can register, log in, view doctors, book or cancel appointments, and view appointment history.
- **Doctor:**
  Can manage availability slots, view appointments, and update profile information.
- **Admin:**
  Has full control over system data, including managing users, doctors, and appointments.

8.4 System Features

- Responsive UI with real-time validations
- Role-based dashboards
- Secure authentication
- MongoDB cloud integration
- Email alert system

- Use Case Diagrams

9. Software Engineering Paradigm

Adopted Agile SDLC with iterative development, testing, and feedback loop.

Key Characteristics of the Adopted Paradigm:

- **Iterative Development:**
  The system was developed in multiple iterations (sprints), with each cycle focusing on a specific module or functionality (e.g., user authentication, appointment booking, admin dashboard).

- **Early and Continuous Testing:**
  Testing was integrated into each sprint cycle to ensure that defects were identified and resolved early, maintaining a high level of code quality.

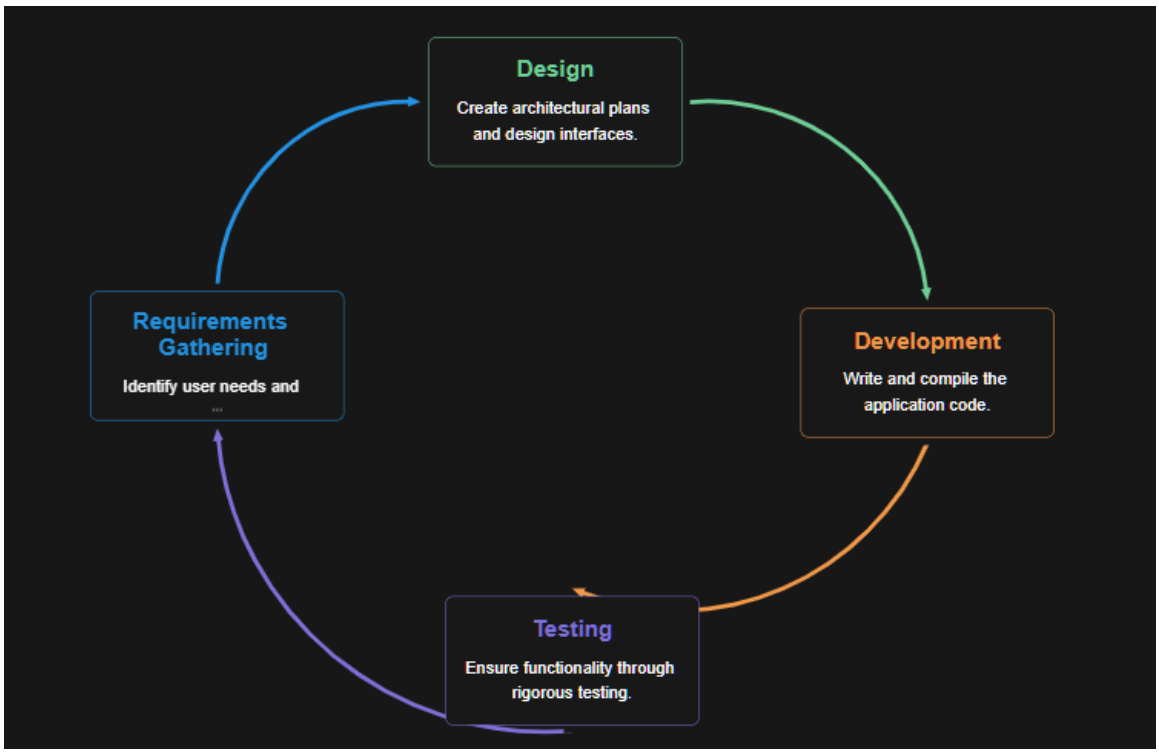
- **Customer-Centric Feedback Loop:**
  Regular feedback was gathered from potential users (patients and doctors) to validate usability, identify missing features, and align the product with real-world needs.

- **Adaptive Planning:**
  Agile allowed changes in requirements to be incorporated easily without disrupting the development flow.

Benefits Observed:

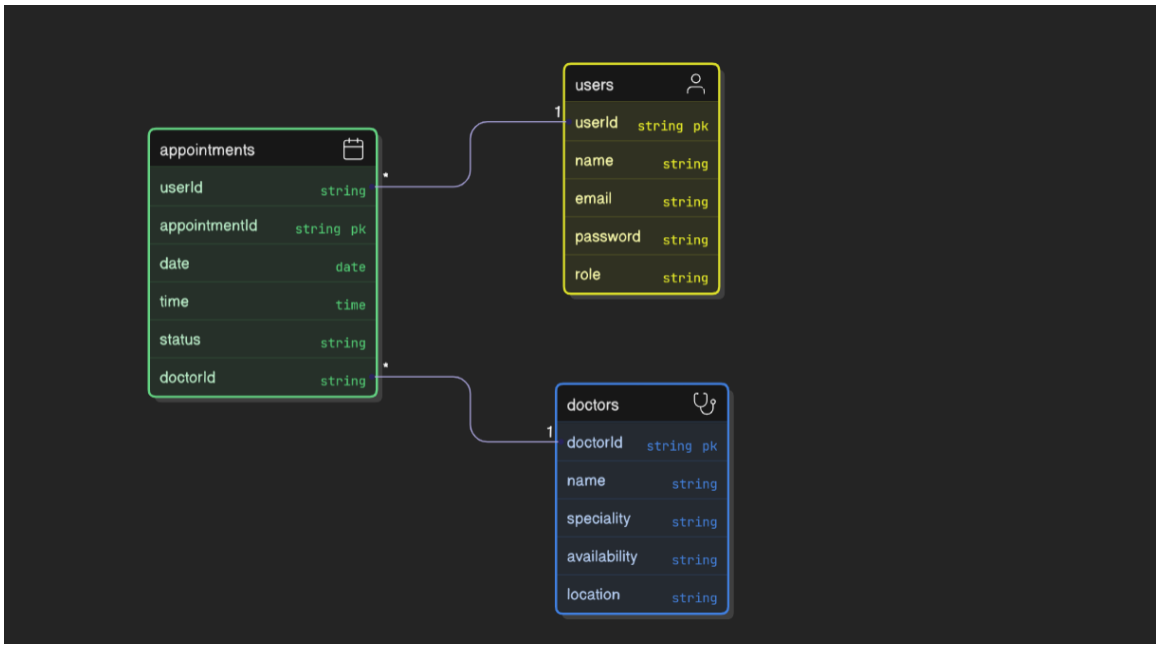
- Faster development and deployment of core features

- Easier debugging due to modular releases
- Improved user satisfaction through continuous feedback and iteration

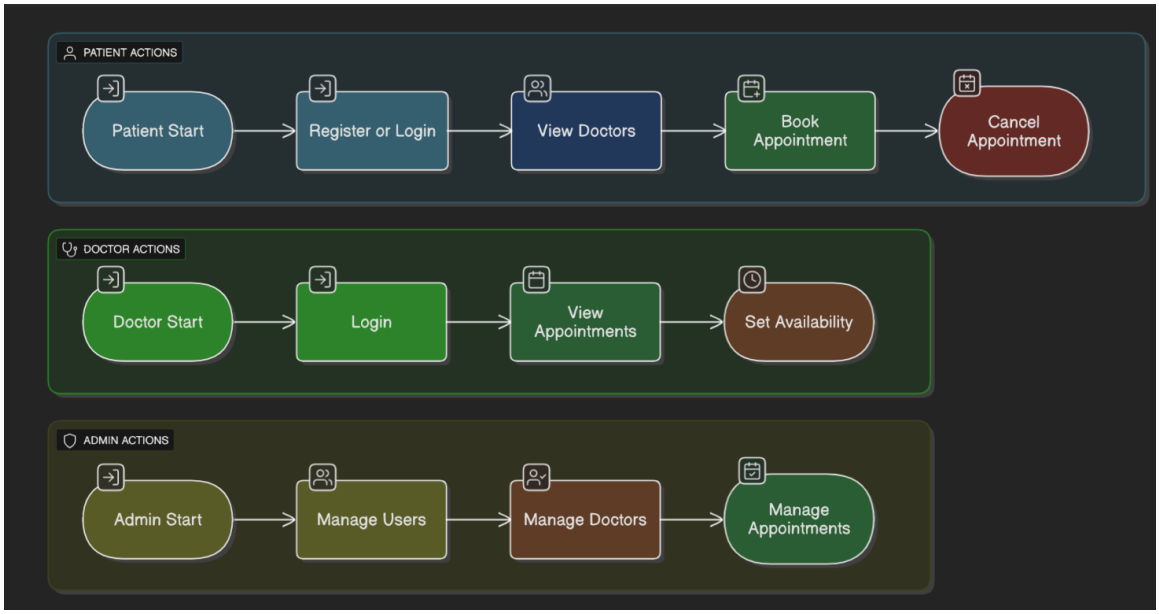


10. Data Models

- ER Diagram for database



- Use-Case Diagram for user interactions



- DFD (Level 0 & 1) for process flow

11. System Design

- Low-fidelity Wireframes via Figma
- Component-based design for React frontend
- REST API structure

11.1 Low-Fidelity Wireframes.

To visualize the application's user flow and interface layout, low-fidelity wireframes were created using **Figma**. These wireframes served as blueprints for frontend development, enabling early feedback and UI/UX refinement.

Key Screens Designed:

- Login / Signup Page
- Patient Dashboard
- Doctor Dashboard
- Appointment Booking Page
- Admin Panel
- Appointment History

11.2 Component-Based Design (React Frontend)

The frontend was developed using **React.js** with a **component-based architecture**, promoting reusability and modularity.

Main Components:

- AuthForm: Handles login and signup
- Navbar: Dynamic navigation for all user roles
- AppointmentForm: Used by patients to book appointments
- Dashboard: Separate dashboards for doctor, patient, and admin views
- AppointmentList: Displays upcoming and past appointments
- UserProfile: Profile and settings section for all users

12. Modularisation Details

Modules:
- Auth Module
- Appointment Module
- Doctor Management
- Admin Dashboard
- Patient Profile

12.1 Auth Module

Purpose:
Handles user registration, authentication, and role-based access control.

Key Features:

•Signup and Login (JWT-based authentication)

•Password hashing using bcrypt

Role-based redirection (Patient, Doctor, Admin) •

•

Middleware to protect private routes **Key Routes:**

- POST /auth/register
- POST /auth/login
- GET /auth/profile (Protected)

12.2 Appointment Module

Purpose:
Manages all functionalities related to booking, viewing, and canceling appointments.

Key Features:

•Book appointment (select doctor, date, and time) •View

upcoming and past appointments

•Cancel/reschedule appointments (if allowed) •

Doctor view for managing schedules **Key Routes:**

- POST /appointments
- GET /appointments/user/:id
- DELETE /appointments/:id

12.3 Doctor Management Module

Purpose:
Allows admin to manage doctor profiles and availability.

Key Features:

•Add/edit/remove doctor details

•Assign specialties and working hours

View doctor activity and appointments •

Key Routes:

- GET /doctors
- POST /doctors
- PUT /doctors/:id
- DELETE /doctors/:id

12.4 Admin Dashboard Module

Purpose:
Provides system-wide control and analytics to the admin.

Key Features:

•View all users and doctors

•Monitor system appointments and usage

•Approve or manage users

•

Generate basic reports **Key Routes:**

- GET /admin/users
- GET /admin/appointments
- GET /admin/stats

12.5 Patient Profile Module

Purpose:
Allows patients to manage their personal information and appointment history.

Key Features:

- Edit profile (name, contact, etc.)
- View personal appointment history
- Change password

Key Routes:

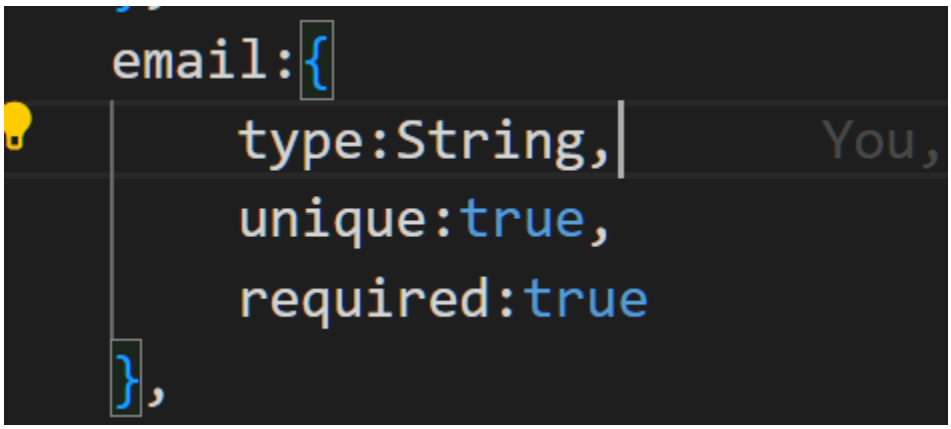
- GET /users/:id
- PUT /users/:id
- GET /appointments/user/:id (reused from appointment module)

13. Data Integrity and Constraints

- Email must be unique
- Date fields restricted to future dates
- Passwords hashed using bcrypt

13.1 Unique Email Constraint

- **Purpose:** Prevent duplicate user registrations.
- **Implementation:**
    - Enforced at the database level using a **unique index** on the email field.
    - Also validated in the backend before creating a new user account.



13.2 Date Fields Restricted to Future Dates

•**Purpose:** Prevent users from booking appointments in the past.

- **Implementation:**
    - Frontend date pickers limit selection to current date + future dates.
    - Backend validation checks if the selected appointment date is greater than or equal to the current date.

13.3 Password Hashing with Bcrypt

•**Purpose:** Enhance user security by not storing plain-text passwords.

- **Implementation:**
    - Passwords are hashed using **bcrypt.js** before being saved in the database.
    - During login, bcrypt compares the hashed version of the entered password with the stored hash.

```
// After that encrypt the password and save in database
// Encryption password.
// First, we have to generate one salt to hash the password.
//Hashing doctor password.
const salt = await bcrypt.genSalt(10); // genSalt method - here we pass
10, it is number of round.
// After that create variable for hash password.
const hashedPassword = await bcrypt.hash(password, salt);
// After that we will get one encrypted password in the "hashedPassword"
variable.
  You, last month • multer …
//Next, we have to upload the image file on the cloudinary so that we
will get image url.
```

14. Database Design

- MongoDB collections: Users, Doctors, Appointments
- Use of indexes for fast lookup

14.1 Collections Overview

1.**Users Collection**

- Stores information for all system users (patients, doctors, admins).

- Common fields: name, email, password, role, createdAt.
- Role-based access is controlled via a role field (patient, doctor, admin).

Shema

```
{
  _id: ObjectId,
  name: String,
  email: String (unique),
  password: String (hashed),
  role: String,
  createdAt: Date
}
```

2.**Doctors Collection**

- Contains doctor-specific data such as specialty and available timings.
- Linked to Users collection via a userId reference.

```
{
  _id: ObjectId,
  userId: ObjectId, // Reference to Users collection
  specialty: String,
  availability: [Date],
  rating: Number
}
```

3.**Appointments Collection**

- Stores details about appointments between patients and doctors.

- Linked to both Users and Doctors collections via references.

```
{
  _id: ObjectId,
  patientId: ObjectId, // Reference to Users
  doctorId: ObjectId,  // Reference to Doctors
  appointmentDate: Date,
  status: String, // e.g., pending, confirmed, canceled
  createdAt: Date
}
```

14.2 Relationships and Data Access

- **One-to-One:**
  Each doctor has one linked user account (Doctors.userId → Users._id).

- **One-to-Many:**
  A patient can have multiple appointments (Appointments.patientId → Users._id).
  A doctor can also have multiple appointments.

- **Referential Integrity:**
  While MongoDB is schema-less, these relationships are enforced logically in the application layer using Mongoose references

15. User Interface Design

- Responsive UI with Tailwind CSS
- Optimized for mobile & desktop
- Accessibility features included

15.1 Responsive UI with Tailwind CSS

- The frontend was styled using **Tailwind CSS**, a utility-first CSS framework.

- Tailwind enabled rapid prototyping with consistent spacing, typography, colors, and responsive utilities.

- Mobile-first design principles were followed using Tailwind's responsive breakpoints (sm, md, lg, xl).

15.2 Optimized for Mobile & Desktop

- All critical pages (e.g., login, booking, dashboards) were tested across devices using Chrome DevTools and mobile emulators.
- Flexbox and grid systems were used to create dynamic layouts that adjust gracefully to available screen real estate.
- Buttons, inputs, and navigation elements are touch-friendly and appropriately sized for mobile interaction.

15.3 Accessibility Features Included

Accessibility was considered during UI development to ensure the application is usable by all, including those with disabilities.

Key Accessibility Features:

- **Keyboard Navigation:** All interactive elements can be accessed using the Tab key.
- **Semantic HTML:** Use of proper tags (<header>, <main>, <section>, <button>, etc.) improves screen reader support.
- **ARIA Labels:** Added for non-text content (e.g., icons, buttons) to describe functionality to screen readers.
- **Color Contrast:** Maintained recommended contrast ratios for text and UI components for readability.

16. Coding

- MERN Stack
- Modular Codebase
- Key Features: Auth, CRUD for appointments, Doctor availability

16.1 MERN Stack

- **MongoDB** – NoSQL database for storing user, doctor, and appointment data.
- **Express.js** – Backend framework to define RESTful APIs.
- **React.js** – Frontend library for building reusable components and managing UI state.
- **Node.js** – JavaScript runtime for server-side development.

This full-stack JavaScript setup allowed seamless data flow from frontend to backend and efficient real-time UI updates.

16.2 Modular Codebase

The code is organized into modules and follows the **MVC architecture**:

```
/server
|
├── /controllers → Business logic (e.g., appointmentController.js)
├── /models → Mongoose schemas (e.g., User.js, Doctor.js)
├── /routes → API endpoints (e.g., authRoutes.js, doctorRoutes.js)
├── /middleware → Auth, error handling, validators
├── /config → DB connection, environment config
└── server.js → Entry point for the backend server

/client
|
├── /components → Reusable UI blocks (e.g., Navbar, AppointmentCard)
├── /pages → Route-specific pages (e.g., LoginPage, DashboardPage)
├── /services → API interaction (e.g., authService.js)
├── /context → Global state management using Context API
└── App.js → Main application router and layout
```

16.3 Key Features Implemented

Authentication (JWT-based)

- Signup/Login with role-based access control
- Protected routes via middleware
- Bcrypt hashing for password security

CRUD for Appointments

- Patients can book, view, and cancel appointments
- Doctors can view and manage their appointments
- Admin has access to all appointments for monitoring

Doctor Availability Management

- Admin can create/edit doctors and set available time slots
- Patients can only book from available slots
- Doctors can update their own availability (optional)

17. SQL Commands

While MongoDB doesn't use SQL, here are equivalents:
- Collection creation with schema constraints
- Insert many documents
- Role-based access (handled via logic in Node.js middleware)

While this project uses MongoDB, a NoSQL database that doesn't use SQL syntax, many common database tasks such as creating collections, inserting data, and enforcing constraints still apply. Below are MongoDB-native commands with references to their SQL counterparts — included for clarity and academic comparison.

17.1 Creating Collections with Constraints

MongoDB collections are usually created automatically when a document is inserted. However, with **Mongoose**, schema constraints can be defined explicitly.

```
const mongoose = require("mongoose");

const UserSchema = new mongoose.Schema({
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true },
  role: { type: String, enum: ["patient", "doctor", "admin"], default: "patient"
});

module.exports = mongoose.model("User", UserSchema);
```

17.2 Insert Multiple Documents

```
db.doctors.insertMany([
  { name: "Dr. Arjun", specialty: "Orthopedics" },
  { name: "Dr. Leela", specialty: "Pediatrics" }
]);
```

17.3 Role-Based Access (Application Level)

MongoDB doesn't natively support role-based UI access. Instead, it's handled in the **Node.js backend** using middleware.

```
function authorizeRoles(...roles) {
  return (req, res, next) => {
    if (!roles.includes(req.user.role)) {
      return res.status(403).json({ message: "Access denied" });
    }
    next();
  };
}
```

18. Code Quality

- Followed coding standards (ESLint + Prettier)
- Secure API endpoints with JWT
- Reusable components

18.1 Followed Coding Standards (ESLint + Prettier)

- **ESLint** was used to statically analyze JavaScript code and identify problematic patterns or deviations from best practices.
- **Prettier** was used to auto-format code to a consistent style across the codebase.
- A shared .eslintrc.json and .prettierrc configuration was used across both the client and server folders.

Benefits:

- Eliminates inconsistent formatting
- Encourages clean, readable, and uniform code
- Prevents common bugs and anti-patterns

18.2 Secure API Endpoints with JWT

- All private routes are protected using **JSON Web Tokens (JWT)**.
- Users must include a valid token in the Authorization header to access protected resources.
- Middleware checks token validity and extracts user info for role-based access control.

Example Middleware

```
function authMiddleware(req, res, next) {
  const token = req.headers.authorization?.split(" ")[1];
  if (!token) return res.status(401).json({ message:
"Unauthorized" });

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch {
```

```
    res.status(403).json({ message: "Invalid token" });
  }
}
```

18.3 Reusable Components (React)

- UI was broken down into small, reusable React components to follow the **DRY (Don't Repeat Yourself)** principle.
- Components like Navbar, AuthForm, AppointmentCard, UserProfile, and DoctorList were created for reusability across pages.

Benefits:

- Reduces code duplication
- Improves readability and debugging
- Makes testing and refactoring easier

Example

```
// AppointmentCard.jsx
function AppointmentCard({ doctor, date, status }) {
  return (
    <divclassName="p-4 rounded shadow bg-white">
      <h3>{doctor}</h3>
      <p>{date}</p>
      <spanclassName={`badge${status}`}>{status}</span>
    </div>
  );
}
```

19. Testing

- Unit Testing: Jest
- Integration Testing: Postman

- Manual testing for UI
- Bug report log maintained

19.1 Unit Testing (Jest)

- **Tool Used:**Jest
- Focused on testing individual functions, utilities, and backend logic in isolation.
- Key targets included authentication logic, middleware functions, and appointment handlers.

Example

```
describe("Token Verification", () => {
  it("should return a decoded user object for a valid token", () => {
    const token = generateTestToken({ id: "user123", role: "doctor" });
    const user = jwt.verify(token, process.env.JWT_SECRET);
    expect(user.id).toBe("user123");
  });
});
```

19.2 Integration Testing (Postman)

- **Tool Used:** Postman
- All REST API endpoints were tested for:
  - Valid input and expected output
  - Authentication and authorization
  - Error handling and edge cases
- Test collections were created and saved for repeated use.

Tested Scenarios:

- User signup/login
- Token expiration and invalid JWTs

- Appointment creation and cancellation
- Access control for patients, doctors, and admins

19.3 Manual Testing for UI

- **The React frontend was manually tested on:**
    - Desktop (Chrome, Firefox)
    - Mobile view (using Chrome DevTools and real Android device)
- **Checked for:**
    - Responsiveness
    - Navigation flow
    - Visual consistency
    - Accessibility issues

20. System Security

- JWT-based Authentication: JSON Web Tokens (JWT) were used to securely transmit user identity after login. Tokens are signed and verified using a secret key, ensuring that only authenticated users can access protected routes.

- Role-Based Access Control: The system distinguishes between different user roles (e.g., Admin, Doctor, Patient) and enforces permission boundaries accordingly. Access to APIs and functionalities is controlled based on user roles to prevent unauthorized operations.

- **HTTPS deployment:** The application was deployed on a secure HTTPS endpoint, ensuring encrypted communication between client and server. This protects against data interception and man-in-the-middle attacks.

- **MongoDB Atlas IP whitelisting:** The MongoDB database was hosted on MongoDB Atlas, with IP whitelisting configured to allow access only from specific, trusted IP addresses. This adds a layer of network-level protection to the database.

21. Cost Estimation

The **COCOMO (Constructive Cost Model)** Basic Model was used for estimating the effort and development time of the project. The estimation was based on academic-level data and assumptions suitable for student or prototype-level projects.

Model Used: COCOMO Basic Model

Project **Type:** Organic (suitable for small teams, familiar technology)

Estimated **Size:** ~x KLOC (example: 2 KLOC – Key Lines of Code)

Effort **Applied (Person-Months):** E = a × (KLOC)^b

For Organic projects, typically:

•a = 2.4, b = 1.05

•

Example: E = 2.4 × (2)^1.05 ≈ 4.92 PM Development **Time (TDEV):** D

= c × (E)^d

Hosting and Tools

•**Tools Used:** VS Code, GitHub, MongoDB Atlas, Render, and Draw.io

•**Hosting:** Deployed using **Render** and **MongoDB Atlas**, both in free-tier plans.

- **Cost Incurred:** ₹0 (due to usage of free-tier services)

22. Future Scope

- Video Consultation integration
- Payment Gateway (Razorpay)
- SMS/Email Reminders
- Analytics Dashboard

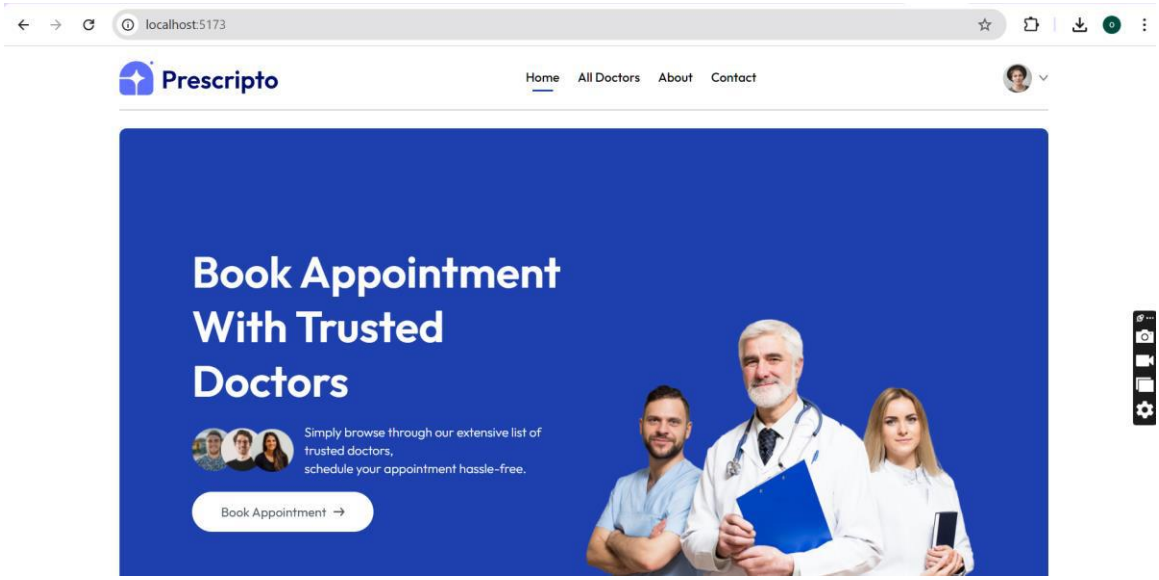
23. Bibliography

- www.practo.com
- www.mongodb.com
- www.reactjs.org
- www.nodejs.org

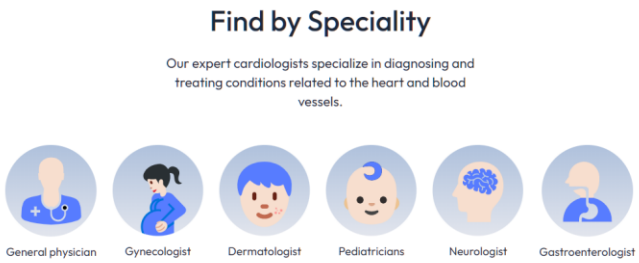
24. Appendices

- Screenshots of UI

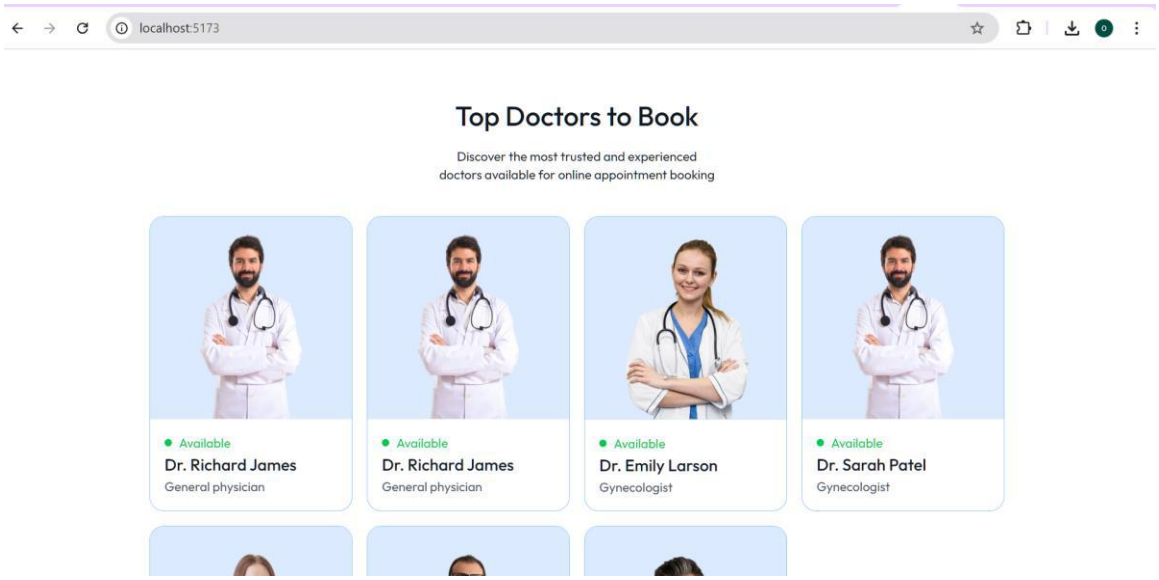
Home page



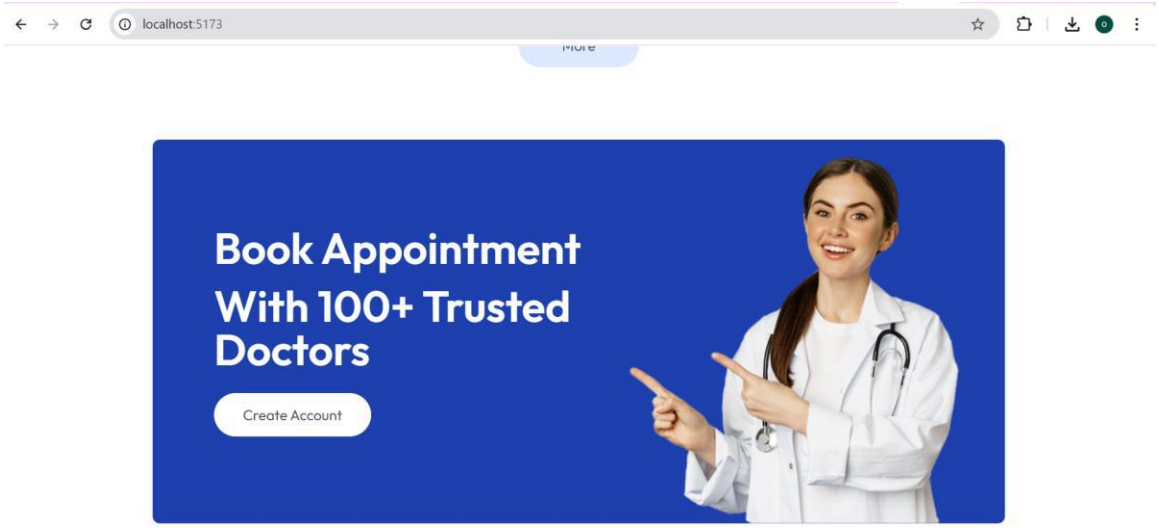
Find by Specialty page



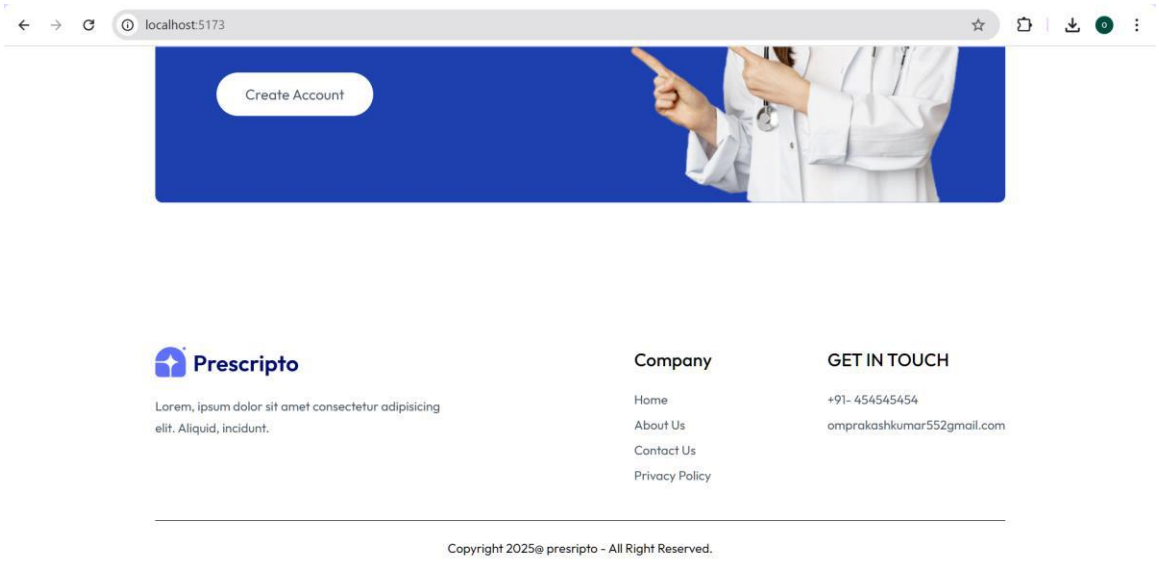
Top doctor page



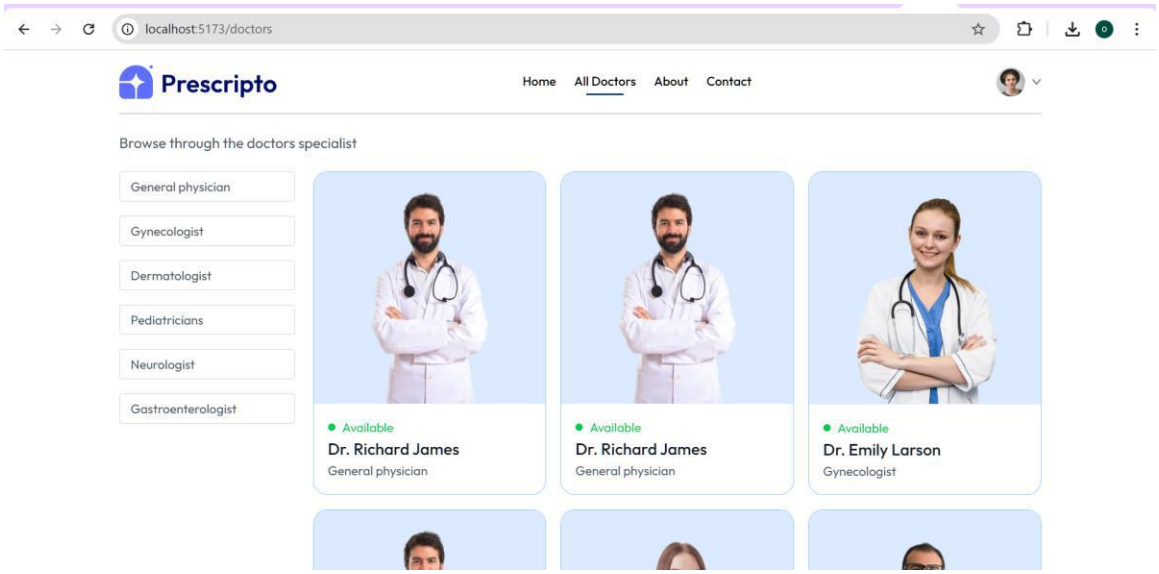
Book Appointment page



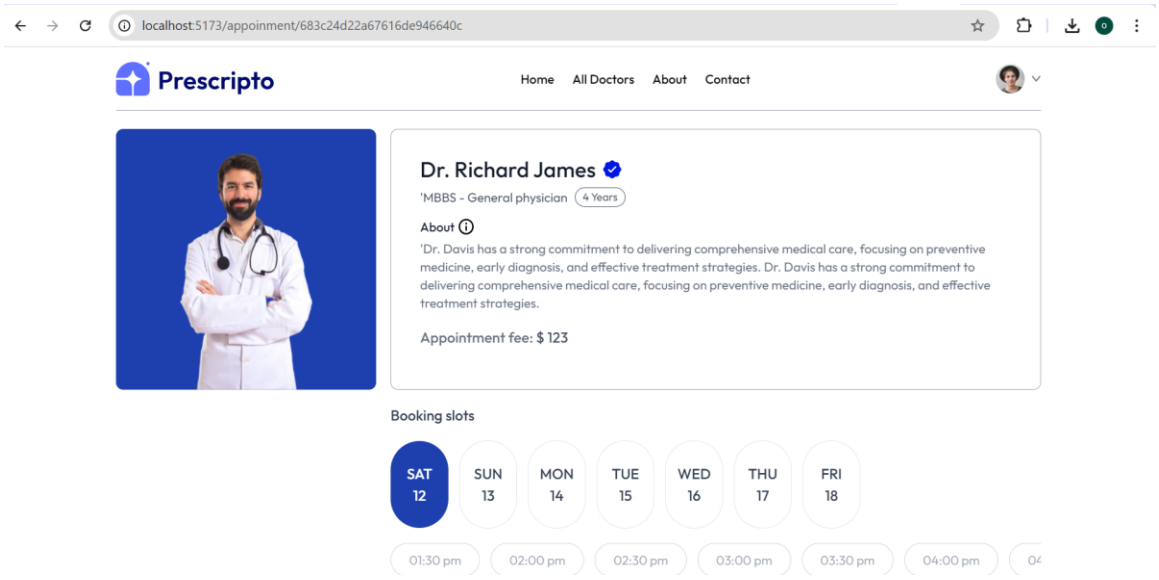
Footer page



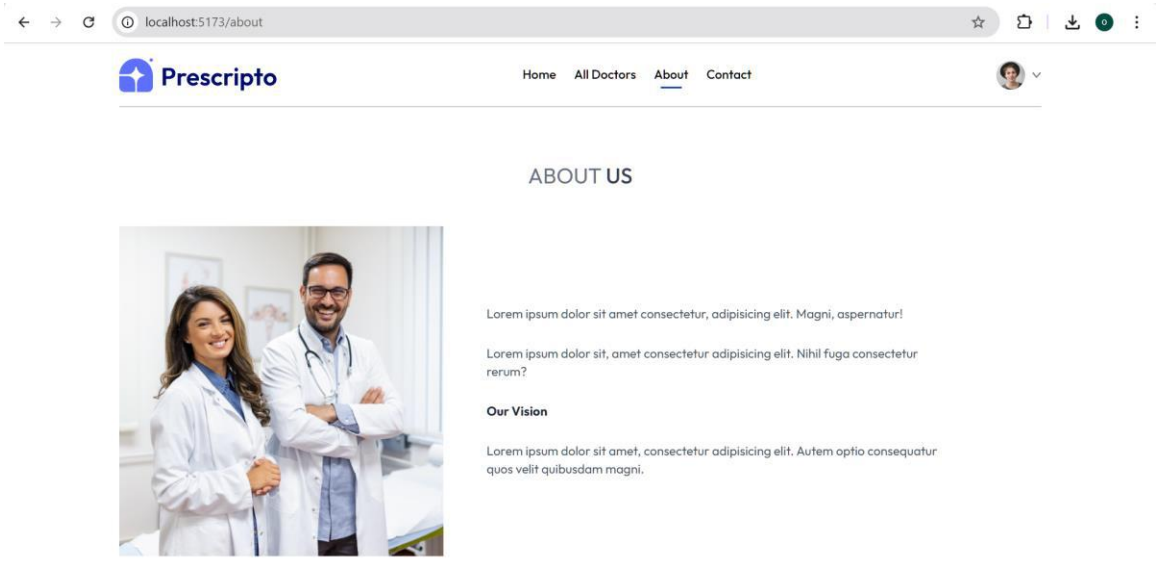
All Doctor page



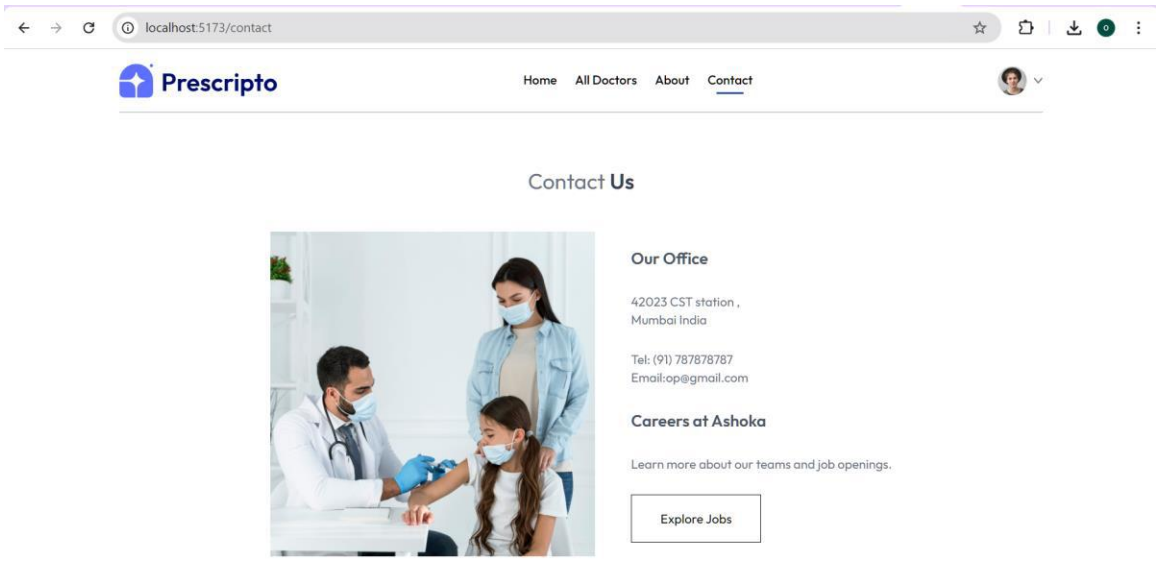
Individual doctor page with more information



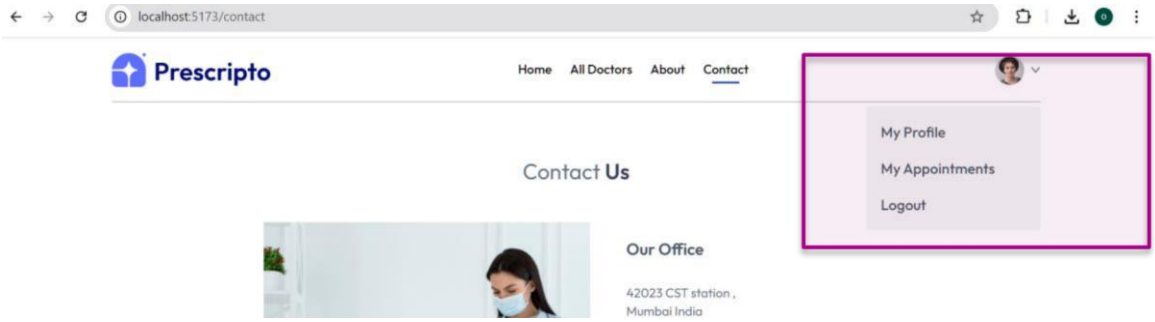
About page

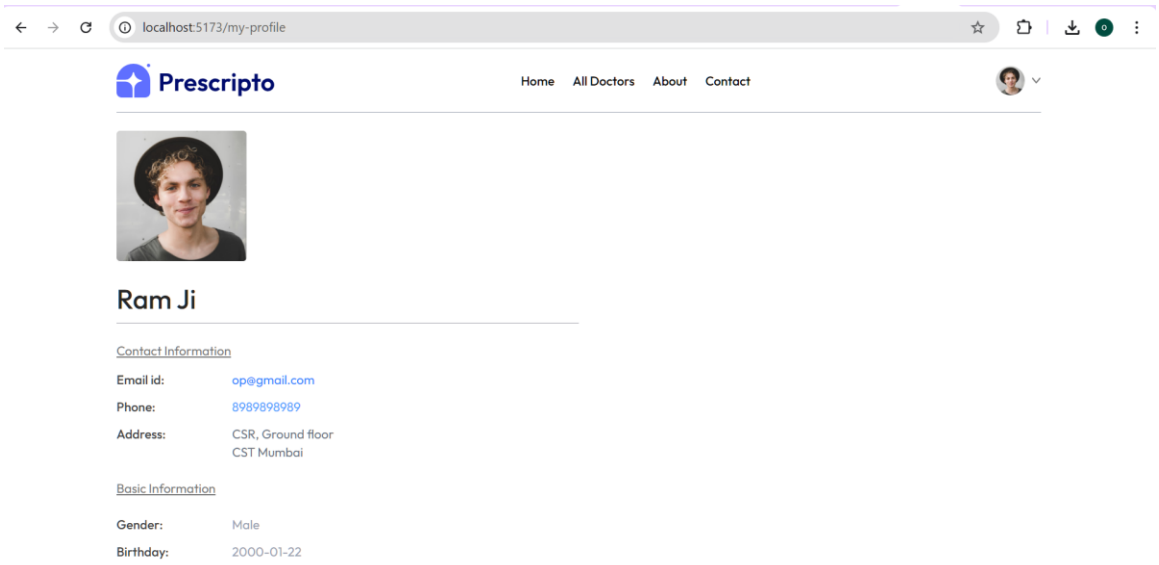


Contact page

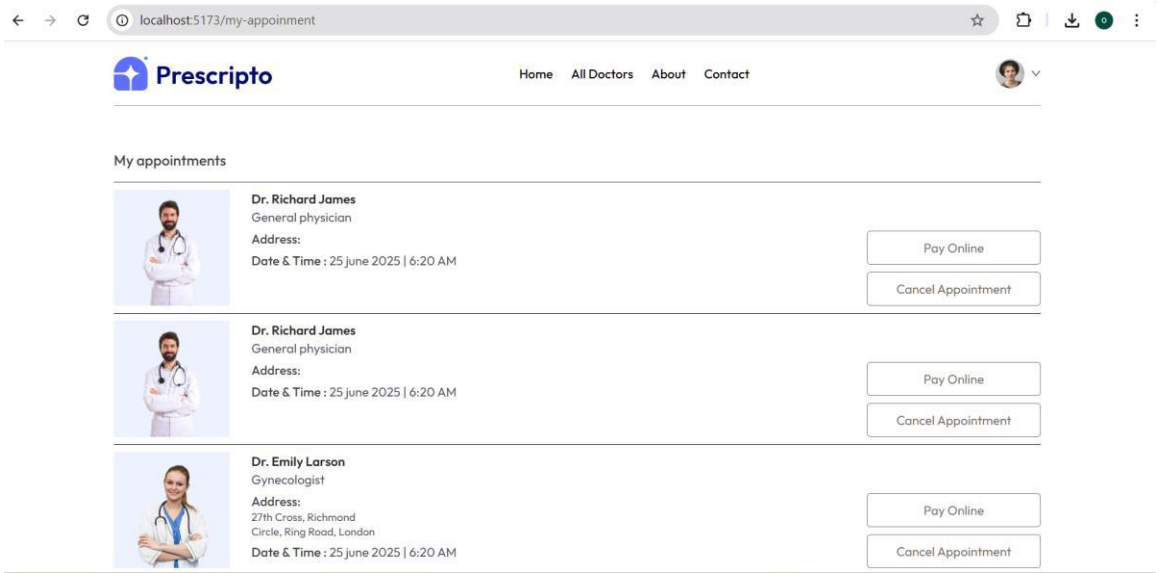

localhost:5173/contact
Prescripto
Home
All Doctors
About
Contact
Contact Us
Our Office
42023 CST station ,
Mumbai India
Tel: (91) 787878787
Email:op@gmail.com
Careers at Ashoka
Learn more about our teams and job openings.
Explore Jobs

Login page

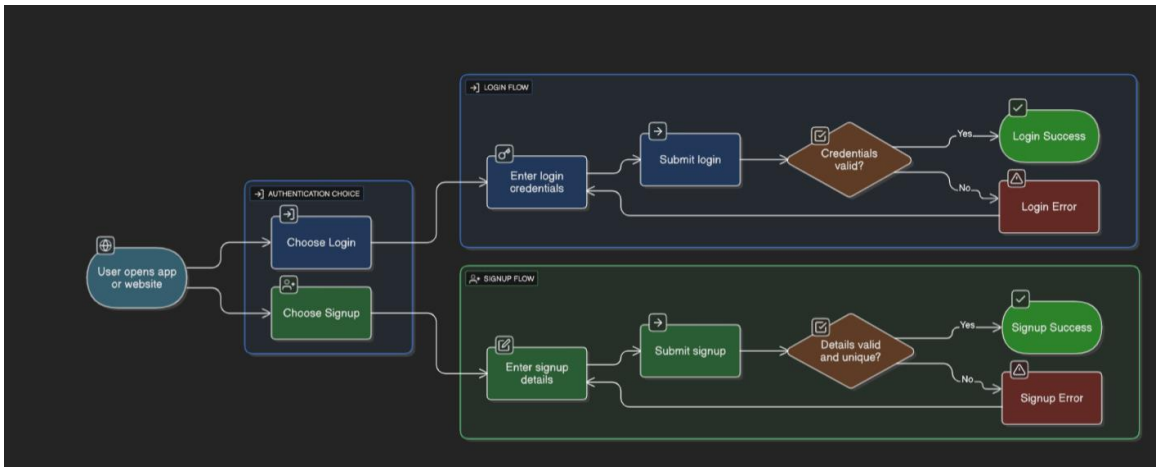

localhost:5173/contact
Prescripto
Home
All Doctors
About
Contact
My Profile
My Appointments
Logout
Contact Us
Our Office
42023 CST station ,
Mumbai India



My appointment page



- Login/Signup Flow Diagram



25. Data Dictionary

A	B	C
Field	**Type**	**Description**
userId	String	Unique identifier for the user
doctorName	String	Full name of the doctor
speciality	String	Doctor's area of specialization
appointmentDate	Date	Scheduled date and time of the appointment