PROJECT REPORT

REPORT SUBMITTED FOR THE PARTIAL FULFILMENT OF THE

REQUIREMENT FOR THE THREE-YEAR DIPLOMA IN

“Computer Science/Information Technology”

SUBMITTED TO

“Government Polytechnic Mohammadi Kheri”

Mohammadi Kheri

Team Member:

Kushal Chauhan (E24273738900004)

Rahul Raj (E23273735500049)

Amit Kumar (E23273735500042)

Vishal Jayswal (E23273735500029)

Under the Guidance of:-

Mayank Kumar

Lecturer of

Computer Science and Engineering

CERTIFICATE

This is to certify that the project report entitled "**Doctor Appointment System**" submitted by Kushal Chauhan, Rahul Raj, Vishal Jayswal and Amit Kumar in partial fulfillment of the requirements for the award of Diploma in Computer Science and Engineering from Government Polytechnic Mohammadi Kheri is a record of bonafide work carried out by them under my guidance and supervision.

_____

Mr. Mayank Kumar (Lect.)

Government Polytechnic
Mohammadi Kheri

(Project Guide)

DECLARATION

We, hereby, declare that the Project Report submitted to Govt. Polytechnic Mohammadi Kheri for the award of three-year Diploma in Computer Science and Engineering was done under the guidance of Lect. Mayank Kumar.

I also declare that the information in this report is correct as per my knowledge and I bear any responsibilities for any error or omission, if any. The matter embodied in this project work has not been submitted earlier for award of any degree or diploma to the best of my knowledge and belief.

ACKNOWLEDGEMENT

We would like to express our sincere gratitude to all those who helped us during the course of this project work. First and foremost, we are grateful to our project guide Mr. Mayank Kumar for his valuable guidance, constant encouragement, and support throughout the development of this project. His suggestions and feedback helped us understand the practical aspects of software development.

We are thankful to the Head of Department, Computer Science and Engineering, Government Polytechnic Mohammadi Kheri, for providing us with the necessary facilities and infrastructure to complete this project successfully.

Team Members:

Kushal Chauhan

Rahul Raj

Vishal Jayswal

Amit Kumar

INDEX

Introduction

The Doctor Appointment System is a web-based application designed to streamline the process of booking and managing medical appointments. Traditional appointment methods rely heavily on manual coordination, which often leads to scheduling conflicts, long waiting times, and inefficient communication between patients, doctors, and administrative staff. This project aims to eliminate these challenges by providing an automated, user-friendly, and secure digital platform.

The system enables **patients** to register, log in, browse available doctors, check their schedules, and book appointments conveniently from any device. **Doctors** can manage their availability, view appointment requests, and approve or reject bookings in real time. **Administrators** have centralized control to manage doctors, patients, and appointments, ensuring smooth operations across the entire platform.

Technology Used :-

The Doctor Appointment System is built using the MERN stack, which integrates four core technologies: MongoDB, Express.js, React.js, and Node.js. This stack enables end-to-end development using JavaScript, ensuring seamless communication between client, server, and database layers. The following technologies and tools form the backbone of the system:

1. MongoDB (Database Layer) :

MongoDB is a NoSQL, document-oriented database used to store the application's structured and semi-structured data. It uses collections and documents instead of traditional relational tables, making it highly flexible and scalable.

Key Features and Role in the Project:

- Stores patient profiles, doctor details, appointment data, and admin records in BSON documents.
- Supports dynamic data structures, allowing the system to grow without requiring complex schema migrations.
- Provides high performance with fast read/write operations, suitable for real-time appointment updates.
- Works seamlessly with Node.js through Mongoose for efficient database query handling.

2. Express.js (Backend Framework) :-

Express.js is a lightweight, flexible framework for building backend services and RESTful APIs. It simplifies the process of handling server-side logic and communication between frontend and database.

Key Features and Role in the Project:

- Manages all API routes such as appointment booking, user login, doctor approval, and admin actions.
- Handles HTTP methods (GET, POST, PUT, DELETE) efficiently.
- Supports middleware for input validation, authentication, and error handling.
- Ensures secure data flow between React frontend and MongoDB database.

3. React.js (Frontend Framework):-

React.js is used to develop the user interface of the Doctor Appointment System. It allows creation of reusable UI components, improving development efficiency and maintaining consistency throughout the application.

Key Features and Role in the Project:

- Provides a responsive and interactive UI for patients, doctors, and administrators.
- Uses state management and lifecycle methods to reflect real-time updates such as appointment confirmation or rejection.
- Virtual DOM improves performance by updating only required components.
- Supports routing and navigation for different user dashboards.

4. Node.js (Server-Side Runtime Environment)

Node.js is a JavaScript runtime built on Chrome's V8 engine. It allows developers to run JavaScript on the server, enabling the full stack to be written in a single language.

Key Features and Role in the Project:

- Handles backend logic, API execution, routing, and integration with external libraries.
- Event-driven architecture ensures high performance, especially during simultaneous user operations.
- Manages asynchronous operations such as database queries and authentication processes.
- Provides a scalable environment for hosting the appointment management system.

5. Supporting Tools, Libraries, and Technologies

5.1 Mongoose (ODM Library)

Mongoose is used to structure and validate MongoDB data using schemas.

- Defines models for Patients, Doctors, Admins, and Appointments.
- Simplifies query execution and data manipulation.
- Ensures data consistency and prevents malformed entries.

5.2 JWT (JSON Web Tokens)

JWT is used for secure user authentication and session management.

- Enables role-based login: Patient, Doctor, Admin.
- Protects secured routes and prevents unauthorized access.
- Provides encrypted tokens stored on the client side.

5.3 bcrypt.js (Password Hashing Library)

bcrypt.js is used to secure user credentials.

- Hashes user passwords before storing them in the database.
- Prevents exposure of plain-text passwords in case of a breach.
- Ensures strong salted encryption for data safety.

5.4 Axios / Fetch API

These libraries are used for making HTTP requests between the frontend and backend.

- Sends API requests for login, appointment booking, approval, rejection, etc.
- Supports asynchronous operations and error handling in React.

5.5 Git & GitHub (Version Control)

- Used for code tracking, branching, and collaboration.
- Maintains version history and backup of the entire project.
- Facilitates smooth deployment workflows.

5.6 Postman (API Testing Tool)

Postman is used to test backend APIs individually.

- Tests all routes (auth, appointment, doctor actions, admin features).
- Ensures each function works correctly before frontend integration.
- Speeds up debugging and development.

About the Project:-

The Doctor Appointment System is a comprehensive web-based application designed to digitize and streamline the process of scheduling medical appointments. The project addresses the limitations of traditional manual booking systems, such as long waiting times, communication gaps, scheduling conflicts, and difficulties in managing patient and doctor records. By leveraging the MERN stack, the system provides an efficient, scalable,and user-friendly solution for patients, doctors, and administrators.

This project enables **patients** to create accounts, log in securely, view available doctors, check their specialties and timings, and book appointments online without visiting the hospital physically. Each appointment request is forwarded to the respective doctor, who can review, approve, or reject the booking based on availability. The system ensures real-time updates and transparent communication between the patient and doctor.

For **doctors**, the portal offers a dedicated dashboard to manage their schedules, review appointment requests, update availability, and maintain patient consultation records. This improves time management and reduces administrative workload.

The **admin module** provides full control over the system. Administrators can add or remove doctors, manage patient accounts, handle appointment records, and oversee the system's overall operation. The admin interface ensures that the platform remains organized, secure, and up to date.

The project demonstrates the use of modern web technologies to solve real-world healthcare challenges. It includes essential features such as user authentication, role-based access control, secure data management, responsive UI, and automated workflows. By combining efficiency, accuracy, and simplicity, the Doctor Appointment System significantly enhances the healthcare service experience and improves hospital–patient interactions.

MODULES

Module Name	MongoDB Collections	Description
Patient Module	users (Patient), appointments	Handles patient interactions & bookings
Doctor Module	users (Doctor), doctor_schedules, appointments	Manages doctor info & availability
Admin Panel	users (Admin), appointments, audit_logs	Controls system operations
Appointment Module	appointments, doctor_schedules	Central scheduling workflow

Objectives

The objective of the **Doctor Appointment System (MERN-based)** is to build a secure, reliable, and user-friendly healthcare scheduling platform where:

1. **Streamlined Appointment Management:** Enable patients to easily register, browse doctors, check availability, and book appointments without manual intervention, ensuring a smooth scheduling experience.

2. **Optimized Doctor Scheduling:** Provide doctors with an efficient interface to manage their availability, update time slots, review upcoming appointments, and handle patient interactions.

3. **Centralized Patient & Doctor Data:** Maintain an integrated database for storing user profiles, medical appointments, schedules, and system activity, ensuring quick access and organized record-keeping.

4. **Real-Time Availability & Conflict Prevention:** Display accurate doctor availability and prevent double-bookings through automated schedule validation and real-time slot updates.

5. **Secure Authentication & Role Management:** Implement a robust user authentication and authorization mechanism to safeguard patient data and differentiate functionalities for Admin, Doctor, and Patient roles.

6. **Enhanced Patient Engagement:** Facilitate improved doctor–patient communication using appointment notifications, reminders, and status updates for a transparent healthcare experience.

7. **Efficient Administrative Control:** Equip administrators with tools to monitor system performance, manage users, oversee appointments, and maintain operational accuracy across the healthcare workflow.

8. **Reporting & Insights:** Provide analytical insights such as appointment frequency, patient visit trends, doctor utilization rate, and cancellations to support data-driven decisions.

9. **Scalability & Performance:** Design the platform to scale with growing users, additional clinics, expanded doctor categories, and increasing appointment volumes without compromising performance.

10. **Improved Healthcare Accessibility:** Make healthcare services more accessible by reducing waiting time, minimizing physical queues, and enabling patients to seek medical care anytime from any device.

SYSTEM ANALYSIS

This Project is a Doctor Appointment System. The analysis steps of project are given below: -

- **Feasibility Study**

Feasibility study is the measure of how beneficial or practical the development of an information system will be to an organization. The Feasibility analysis is a cross life cycle activity and should be continuously performed throughout the system life cycle.

- **Operational Feasibility: -**

By providing the web-based application, all the users will get a very good facility of accessing the service to fulfil their requirements. All the user information, information sharing and selection process is done properly.
Users will feel comfortable by reduction of their work. The system will make handling of large databases easy. Losing of records will be avoided. Considering all these factors, we can conclude that all the users and end users will be satisfied by the system.

- **Technical Feasibility: -**

For the design and development of the system, several software products have been accommodated.

- Database design – MongoDB
- Interface design – React, Tailwind CSS.
- Coding – Mern Stack

The technology (Mern Stack) has enough efficiency for the development of the system. Therefore, the project is technically feasible.

- **Schedule Feasibility: -**

The duration of time required for the project has been planned appropriately and it is the same as the duration of time expected by the client. Therefore, the application can be delivered to the client within the expected time duration, satisfying the client. Hence the project is feasible in scheduling.

- **Economic Feasibility: -**

According to the resources available and the project scheduling process it is estimated that the expenses allocated for the web application to be developed, by the client is sufficient enough. Hence the economical factor has been considered feasible.

- **Project Planning & Scheduling: -**

Planning is very important part of any software development. In the planning phase we decide which features are to be included in the system to make a good system, how much time do we need to complete the project, what will the cost of the system etc…

A Software Life Cycle or software process is a series of identifiable stages that a software product undergoes during its development. A software product development effort usually starts with a project identification and selection stage and then requirements analysis; design, coding, testing, implementation and maintenance are undertaken.

SOFTWARE REQUIREMENTS FOR DEVELOPMENT

User Interface Designing	React Tailwind CSS
Programming Language	Mern Stack
Database	MongoDB
IDE	VSCode

Modules in Project

Below is the project module list converted into the same concise, module-oriented format you provided. Each module is described for a MERN-stack implementation (MongoDB, Express, React, Node), listing responsibilities, related collections, and key backend routes / frontend actions.

• User Authentication Module

Description: Handles secure sign-up, sign-in, session management and role-based access for Admin, Doctor, and Patient.
Related Collections: `users` (with `role: ['Admin','Doctor','Patient']`)
Key Responsibilities / Features:

- Registration (patient/doctor/admin seeds) and login (JWT or session).
- Password hashing (bcrypt) and password reset flow.
- Role-based middleware (protect routes, authorize actions).
  Typical API Routes: `POST /api/auth/register`, `POST /api/auth/login`, `POST /api/auth/forgot`, `POST /api/auth/reset`
  Frontend Actions: Sign-up, Login, Logout, Role-specific routing & protected pages.

- **Patient Management Module (Patient Module)**

Description: Manages patient profiles and patient-facing features: profile editing, view history, book appointments.

Related Collections: users (patients), appointments, patient_profiles (optional)

Key Responsibilities / Features:

- Patient profile CRUD, view appointment history, cancel appointments.
- Search and filter doctors by specialty, date, or location.
- Receive notifications (booking confirmations, reminders).
  Typical API Routes: GET /api/patients/:id, PUT /api/patients/:id, GET /api/doctors, POST /api/appointments

  Frontend Actions: Patient dashboard, doctor search, appointment booking form, history page.

• **Doctor Management Module (Doctor Module)**

Description: Manages doctor profiles, availability, and appointment approvals/rejections.

Related Collections: users (doctors), doctor_schedules, appointments

Key Responsibilities / Features:

- Doctor profile (specialization, qualifications) management.
- Set and edit availability/time slots; block/unblock slots.
- View incoming appointment requests; approve or reject with comments.
- View schedule/calendar of confirmed appointments.
  Typical API Routes: GET /api/doctors/:id/schedule, PUT /api/doctors/:id/schedule, GET /api/appointments/doctor/:id, PATCH /api/appointments/:id/status

  Frontend Actions: Doctor dashboard, calendar view, approve/reject UI, availability editor.

• **Admin Panel (Admin Module)**

Description: Central administration: add/remove doctors, manage users, monitor appointments, system settings.

Related Collections: users, appointments, audit_logs (optional)

Key Responsibilities / Features:

- CRUD operations for doctors and patients (Admin can add/remove/modify doctors).
- Monitor and search appointments across the system; override status if needed.
- Generate reports and view system metrics.
  Typical API Routes: GET /api/admin/users, POST /api/admin/doctors, DELETE /api/admin/doctors/:id, GET /api/admin/reports.

  Frontend Actions: Admin dashboard, user management pages, reporting UI.

Appointment Module (Appointment Management)

Description: Core scheduling: creates, validates, and tracks appointment lifecycle (requested → approved/ rejected → completed → cancelled).

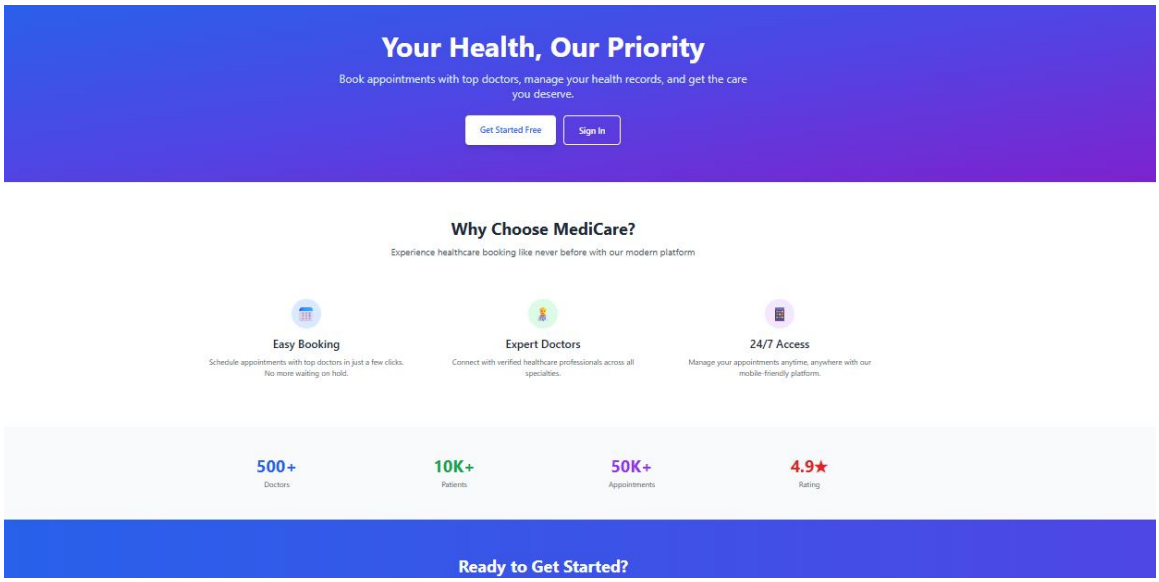
Related Collections: appointments, doctor_schedules, notifications

Key Responsibilities / Features:

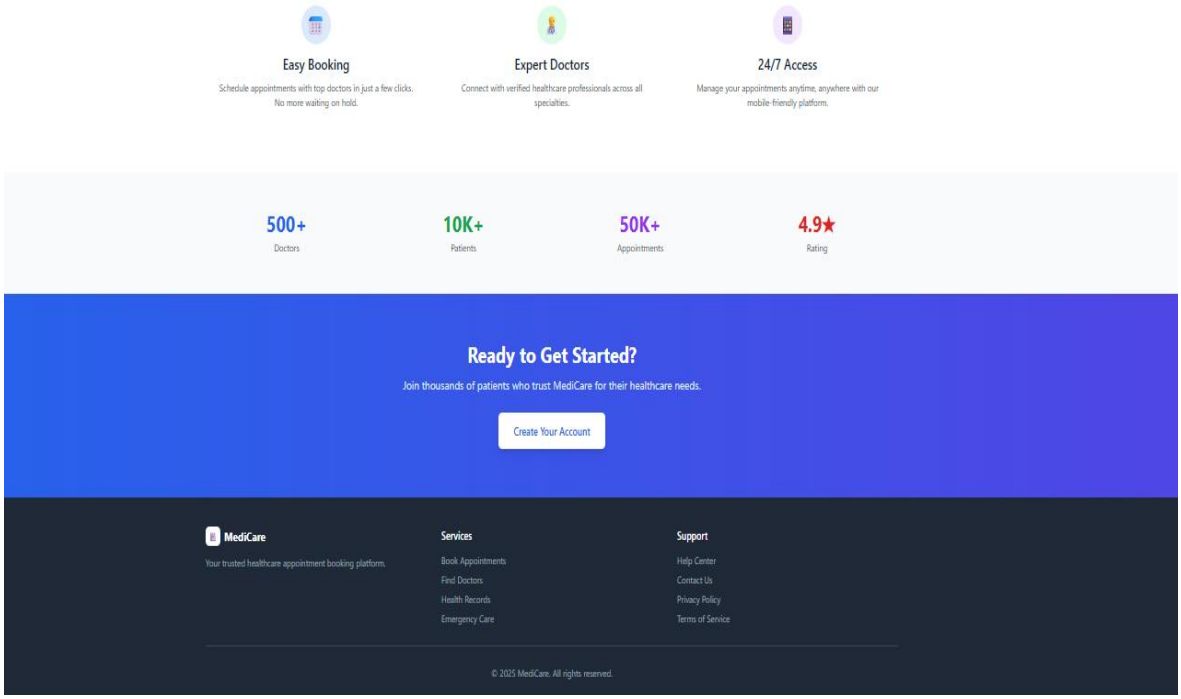
- Create appointment requests from patients.
- Real-time conflict checks against doctor_schedules to prevent double-booking.
- Status transitions (Pending → Confirmed/Rejected → Completed/Cancelled).
- Link appointments to patient and doctor profiles; store reason, attachments, timestamps.
  Typical API Routes: POST /api/appointments, GET /api/appointments/:id, PATCH /api/appointments/:id/status
-
  Frontend Actions: Booking flow, appointment lists, status badges, cancel/ reschedule.

Snapshots

HOME PAGE -1

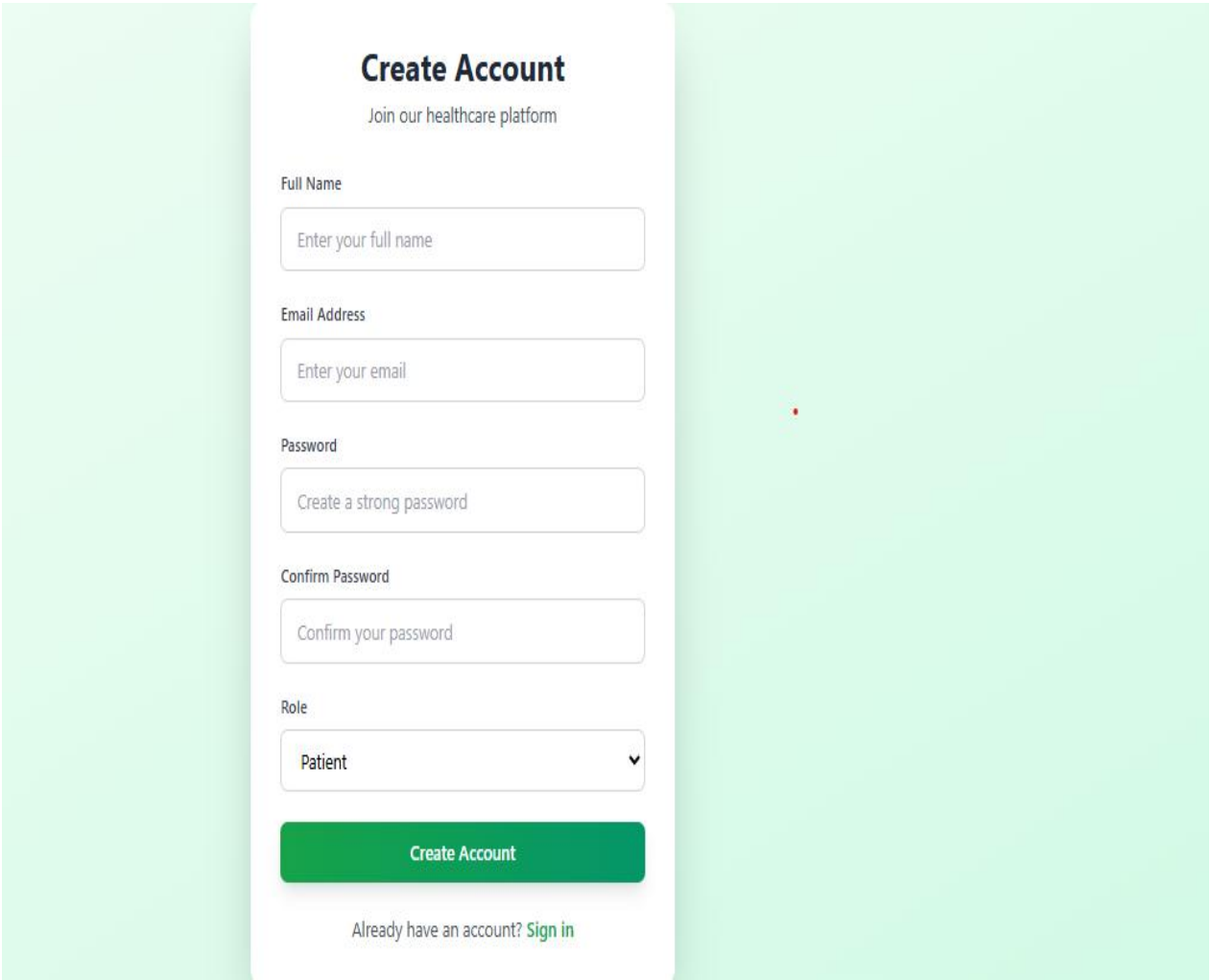


HOME PAGE -2

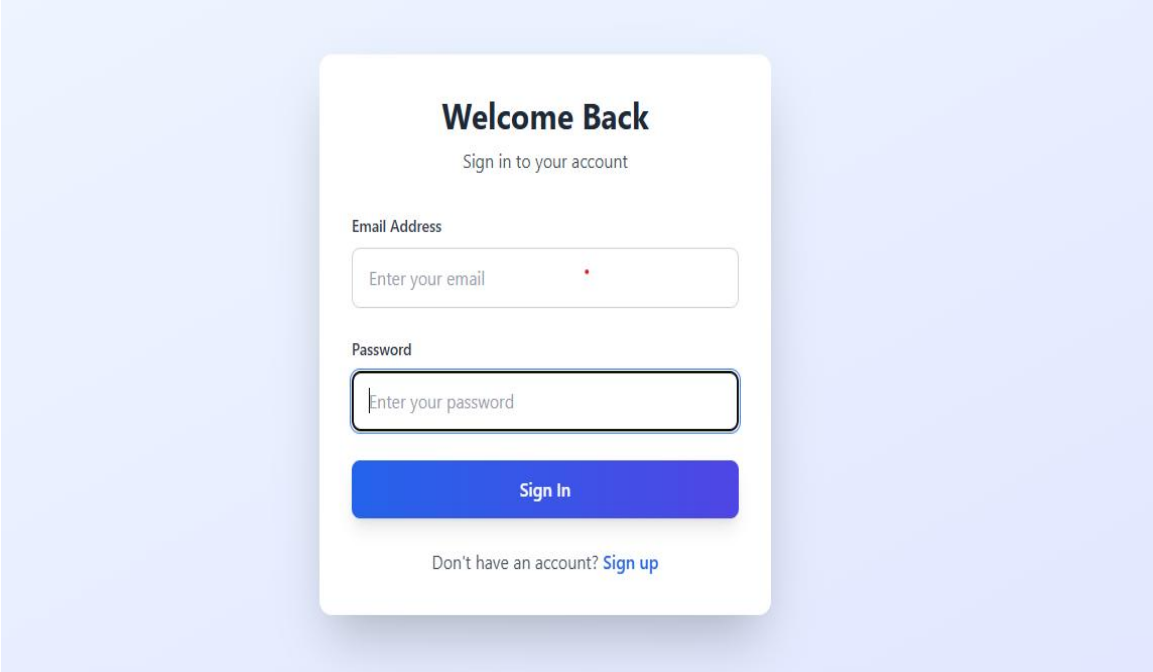


HOME PAGE -3

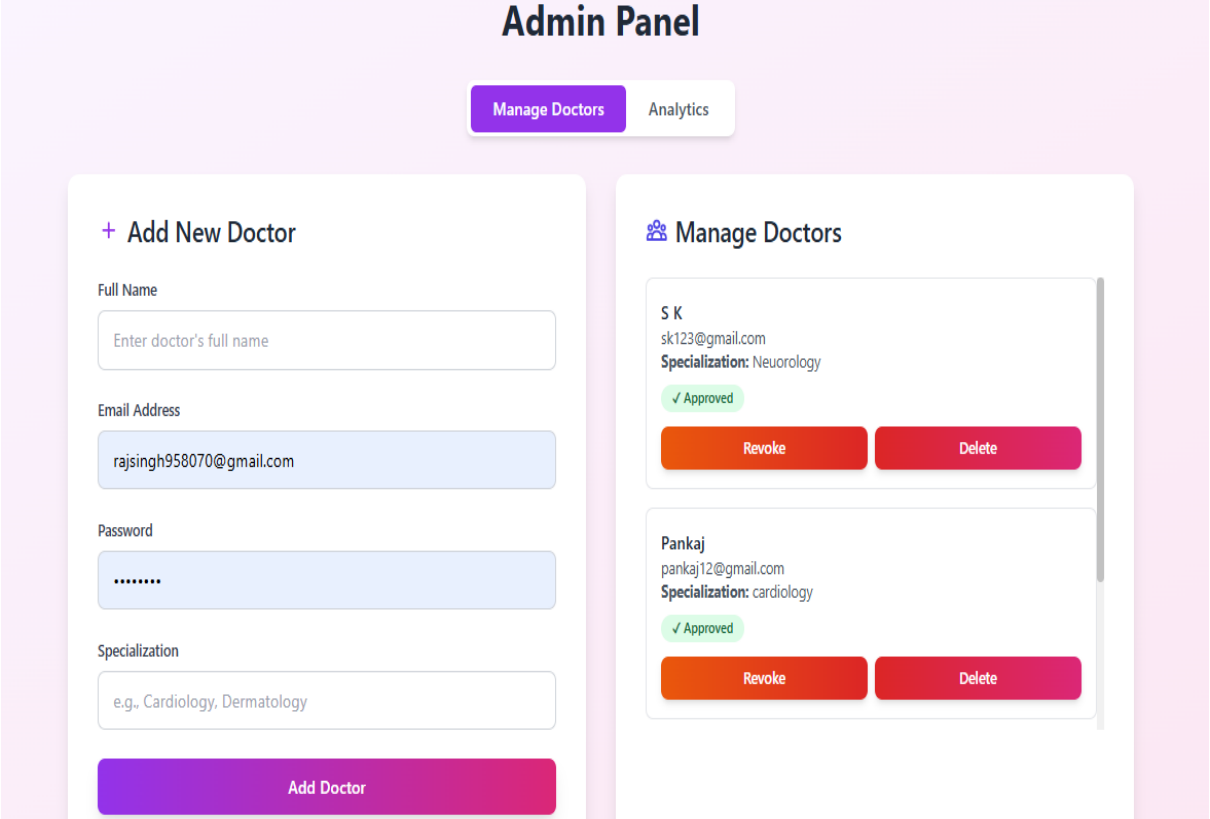
Registration Page



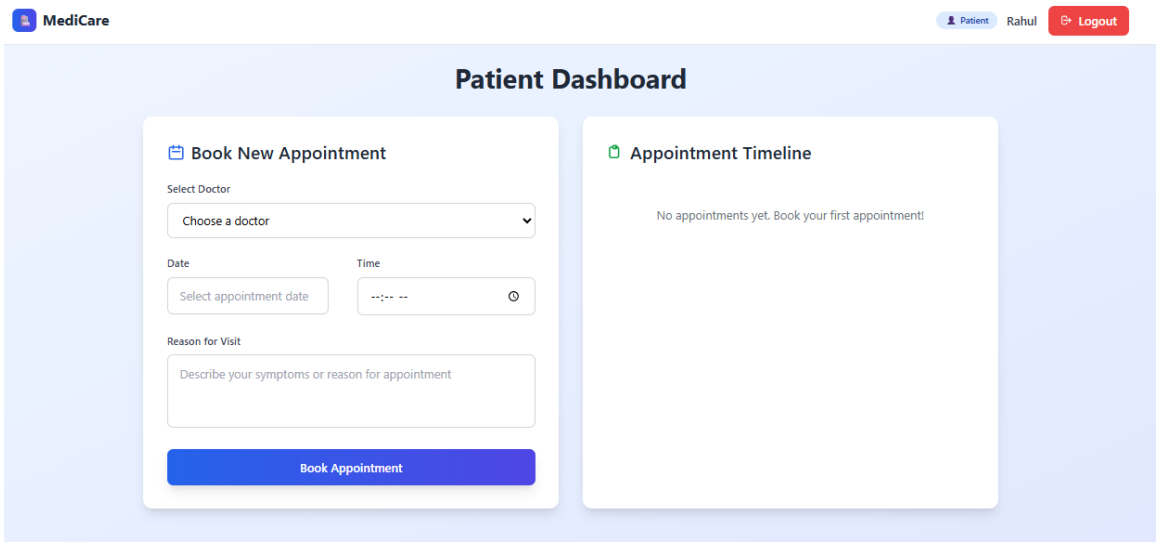
LOGIN PAGE



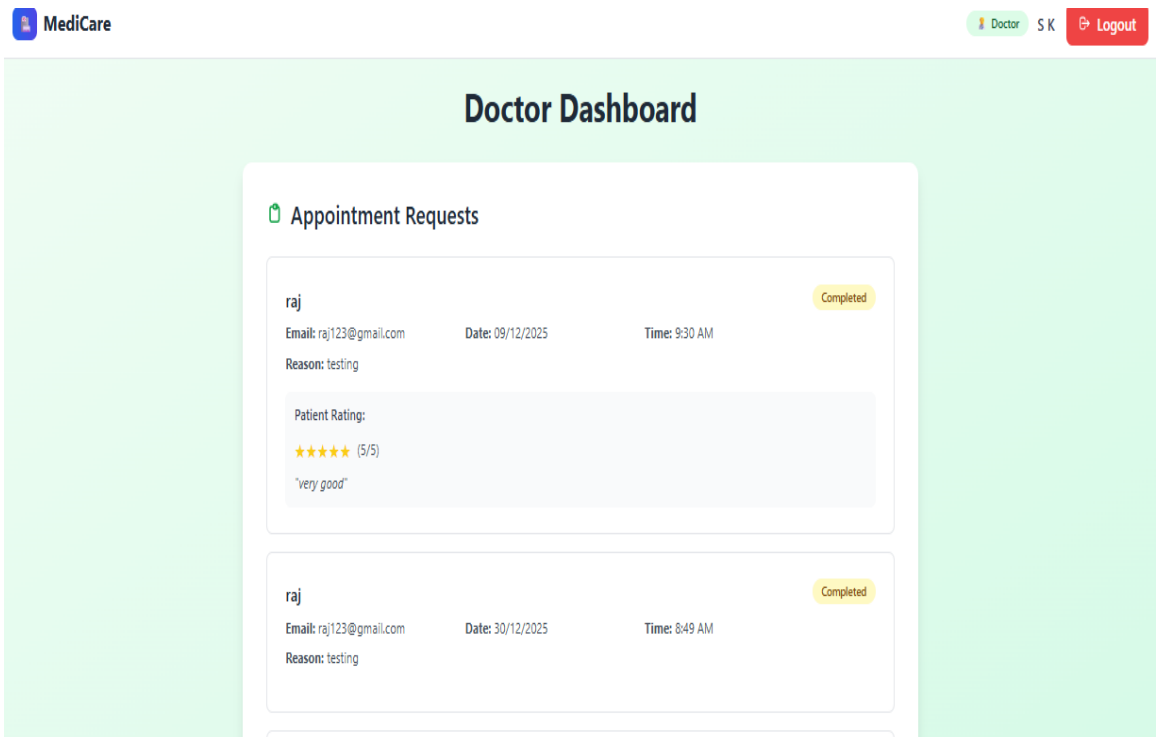
ADMIN DASHBOARD PAGE



PATIENT DASHBOARD PAGE



DOCTOR DASHBOARD PAGE



Tech Stack & Dependencies

```
{
"dependencies": {
"bcryptjs": "^2.4.3",                    // Password hashing
"cors": "^2.8.5",                        // Cross-Origin Resource Sharing
"dotenv": "^16.0.3",                     // Environment variables
"express": "^4.18.2",                    // Web framework
"express-validator": "^7.0.0",           // Input validation
"jsonwebtoken": "^9.0.0",                // JWT authentication
"mongoose": "^7.0.0"                     // MongoDB ODM
"nodemon": "^2.0.0"                      // Auto-restart on file changes
},
```

Coding

Models

Appointments.js:-

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

const appointmentSchema = new mongoose.Schema({
  patient: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },
  doctor: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },
  date: { type: Date, required: true },
  time: { type: String, required: true },
  status: { type: String, enum: ['pending', 'approved', 'rejected', 'completed'], default: 'pending' },
  reason: { type: String },
  rating: { type: Number, min: 1, max: 5 },
  review: { type: String },
  reviewedAt: { type: Date },
  createdAt: { type: Date, default: Date.now }
});

module.exports = mongoose.model('Appointment', appointmentSchema);
```

User.js:-

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

const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true },
  role: { type: String, enum: ['patient', 'doctor', 'admin'], default: 'patient' },
  specialization: { type: String }, // for doctors
  isApproved: { type: Boolean, default: false }, // doctors need approval, patients are auto-approved
  createdAt: { type: Date, default: Date.now }
});

module.exports = mongoose.model('User', userSchema);
```

Routes:-

Admin.js:-

```
const express = require('express');

const bcrypt = require('bcryptjs');

const auth = require('../middleware/auth');

const User = require('../models/User');

const Appointment = require('../models/Appointment');


const router = express.Router();


// Get all doctors

router.get('/doctors', auth, async (req, res) => {

  if (req.user.role !== 'admin') return res.status(403).json({ message: 'Access denied' });


  try {

    const doctors = await User.find({ role: 'doctor' });

    res.json(doctors);

  } catch (err) {

    res.status(500).json({ message: err.message });

  }

});
```

```
// Add doctor
router.post('/doctors', auth, async (req, res) => {
  if (req.user.role !== 'admin') return res.status(403).json({ message: 'Access denied' });

  const { name, email, password, specialization } = req.body;
  try {
    const existingUser = await User.findOne({ email });
    if (existingUser) return res.status(400).json({ message: 'User already exists' });

    const hashedPassword = await bcrypt.hash(password, 10);
    const doctor = new User({ name, email, password: hashedPassword, role: 'doctor', specialization, isApproved: false });
    await doctor.save();
    res.json(doctor);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});

// Approve/Reject doctor
```

```
router.put('/doctors/:id/approve', auth, async (req, res) => {
  if (req.user.role !== 'admin') return res.status(403).json({ message: 'Access denied' });

  const { isApproved } = req.body;
  try {
    const doctor = await User.findById(req.params.id);
    if (!doctor || doctor.role !== 'doctor') return res.status(400).json({ message: 'Doctor not found' });

    doctor.isApproved = isApproved;
    await doctor.save();

    res.json({ message: isApproved ? 'Doctor approved' : 'Doctor rejected', doctor });
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});

// Delete a doctor
router.delete('/doctors/:id', auth, async (req, res) => {
```

```
  if (req.user.role !== 'admin') return res.status(403).json({ message: 'Access denied' });


  try {
    const doctor = await User.findById(req.params.id);
    if (!doctor) {
      return res.status(404).json({ message: 'Doctor not found' });
    }

    if (doctor.role !== 'doctor') {
      return res.status(400).json({ message: 'User is not a doctor' });
    }

    // Delete all appointments for this doctor
    await Appointment.deleteMany({ doctorId: req.params.id });

    // Delete the doctor
    await doctor.deleteOne();
    res.json({ message: 'Doctor deleted successfully' });
  } catch (err) {
    console.error('Delete error:', err.message);
    res.status(500).json({ message: err.message });
```

```
  }
});


// Get dashboard analytics
router.get('/analytics', auth, async (req, res) => {
  if (req.user.role !== 'admin') return res.status(403).json({ message: 'Access denied' });

  try {
    const totalPatients = await User.countDocuments({ role: 'patient' });
    const totalDoctors = await User.countDocuments({ role: 'doctor' });
    const totalAppointments = await Appointment.countDocuments();
    const pendingAppointments = await Appointment.countDocuments({ status: 'pending' });
    const approvedAppointments = await Appointment.countDocuments({ status: 'approved' });
    const completedAppointments = await Appointment.countDocuments({ status: 'completed' });

    // Monthly appointments for the last 6 months
    const sixMonthsAgo = new Date();
    sixMonthsAgo.setMonth(sixMonthsAgo.getMonth() - 6);
```

```
const monthlyAppointments = await Appointment.aggregate([
  { $match: { createdAt: { $gte: sixMonthsAgo } } },
  {
    $group: {
      _id: {
        year: { $year: '$createdAt' },
        month: { $month: '$createdAt' }
      },
      count: { $sum: 1 }
    }
  },
  { $sort: { '_id.year': 1, '_id.month': 1 } }
]);

// Popular specializations
const popularSpecializations = await User.aggregate([
  { $match: { role: 'doctor' } },
  { $group: { _id: '$specialization', count: { $sum: 1 } } },
  { $sort: { count: -1 } },
  { $limit: 5 }
]);
```

```
// Average ratings by doctor
const doctorRatings = await Appointment.aggregate([
 { $match: { rating: { $exists: true } } },
 {
  $group: {
   _id: '$doctor',
   avgRating: { $avg: '$rating' },
   totalReviews: { $sum: 1 }
  }
 },
 {
  $lookup: {
   from: 'users',
   localField: '_id',
   foreignField: '_id',
   as: 'doctor'
  }
 },
 { $unwind: '$doctor' },
 { $project: { name: '$doctor.name', avgRating: 1, totalReviews: 1 } },
 { $sort: { avgRating: -1 } },
 { $limit: 10 }
```

```
    ]);

    res.json({
      totalPatients,
      totalDoctors,
      totalAppointments,
      pendingAppointments,
      approvedAppointments,
      completedAppointments,
      monthlyAppointments,
      popularSpecializations,
      doctorRatings
    });
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});

module.exports = router;
```

appointments.js:-

```
const express = require('express');
const { body, validationResult } = require('express-validator');
const auth = require('../middleware/auth');
const Appointment = require('../models/Appointment');
const User = require('../models/User');


const router = express.Router();


// Get all doctors (for patients to book appointments)
router.get('/doctors', auth, async (req, res) => {
  try {
    const doctors = await User.find({ role: 'doctor' }).select('name email specialization');
    res.json(doctors);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});


// Book appointment (patient)
router.post('/book', auth, [
```

```
  body('doctorId').notEmpty(),
  body('date').isISO8601(),
  body('time').notEmpty(),
  body('reason').optional()
], async (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) return res.status(400).json({ errors: errors.array() });

  if (req.user.role !== 'patient') return res.status(403).json({ message: 'Only patients can book appointments' });

  const { doctorId, date, time, reason } = req.body;
  try {
    const doctor = await User.findById(doctorId);
    if (!doctor || doctor.role !== 'doctor') return res.status(400).json({ message: 'Invalid doctor' });

    const appointment = new Appointment({ patient: req.user.id, doctor: doctorId, date, time, reason });
    await appointment.save();

    res.json(appointment);
  } catch (err) {
```

```
    res.status(500).json({ message: err.message });
  }
});

// Get appointments for user
router.get('/', auth, async (req, res) => {
  try {
    let query = {};
    if (req.user.role === 'patient') query.patient = req.user.id;
    else if (req.user.role === 'doctor') query.doctor = req.user.id;
    else query = {}; // admin sees all

    const appointments = await Appointment.find(query).populate('patient',
'name email').populate('doctor', 'name email specialization');
    res.json(appointments);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});

// Update appointment status (doctor)
router.put('/:id/status', auth, async (req, res) => {
```

```
  if (req.user.role !== 'doctor') return res.status(403).json({ message: 'Only doctors can update appointments' });


  // Check if doctor is approved
  const doctor = await User.findById(req.user.id);
  if (!doctor.isApproved) return res.status(403).json({ message: 'Your account is not yet approved by admin' });


  const { status } = req.body;
  if (!['approved', 'rejected', 'completed'].includes(status)) return res.status(400).json({ message: 'Invalid status' });


  try {
    const appointment = await Appointment.findById(req.params.id).populate('patient', 'name email').populate('doctor', 'name');
    if (!appointment || appointment.doctor._id.toString() !== req.user.id) return res.status(404).json({ message: 'Appointment not found' });


    appointment.status = status;
    await appointment.save();


    res.json(appointment);
  } catch (err) {
```

```
    res.status(500).json({ message: err.message });
  }
});

// Submit rating and review (patient)
router.post('/:id/review', auth, async (req, res) => {
  if (req.user.role !== 'patient') return res.status(403).json({ message: 'Only patients can review appointments' });

  const { rating, review } = req.body;
  if (!rating || rating < 1 || rating > 5) return res.status(400).json({ message: 'Rating must be between 1 and 5' });

  try {
    const appointment = await Appointment.findById(req.params.id);
    if (!appointment || appointment.patient.toString() !== req.user.id) return res.status(404).json({ message: 'Appointment not found' });

    if (appointment.status !== 'completed') return res.status(400).json({ message: 'Can only review completed appointments' });

    appointment.rating = rating;
    appointment.review = review;
```

```
    appointment.reviewedAt = new Date();
    await appointment.save();


    res.json(appointment);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});


module.exports = router;
```

auth.js:-

```
const express = require('express');
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');
const { body, validationResult } = require('express-validator');
const User = require('../models/User');

const router = express.Router();

// Register
router.post('/register', [
  body('name').notEmpty(),
  body('email').isEmail(),
  body('password').isLength({ min: 6 }),
  body('role').optional().isIn(['patient', 'doctor', 'admin'])
], async (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) return res.status(400).json({ errors: errors.array() });

  const { name, email, password, role, specialization } = req.body;
  try {
    const existingUser = await User.findOne({ email });
```

```
    if (existingUser) return res.status(400).json({ message: 'User already exists' });

    const hashedPassword = await bcrypt.hash(password, 10);
    const isApproved = role === 'patient' || role === 'admin' ? true : false; // auto-approve patients and admins, doctors need approval
    const user = new User({ name, email, password: hashedPassword, role, specialization, isApproved });
    await user.save();

    const token = jwt.sign({ id: user._id, role: user.role }, process.env.JWT_SECRET);
    res.json({ token, user: { id: user._id, name, email, role, isApproved } });
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});

// Login
router.post('/login', [
  body('email').isEmail(),
  body('password').notEmpty()
], async (req, res) => {
```

```
  const errors = validationResult(req);
  if (!errors.isEmpty()) return res.status(400).json({ errors: errors.array() });

  const { email, password } = req.body;
  try {
    const user = await User.findOne({ email });
    if (!user) return res.status(400).json({ message: 'Invalid credentials' });

    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch) return res.status(400).json({ message: 'Invalid credentials' });

    const token = jwt.sign({ id: user._id, role: user.role },
process.env.JWT_SECRET);
    res.json({ token, user: { id: user._id, name: user.name, email, role:
user.role } });
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});

module.exports = router;
```

Index.js:-

```
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');
const dotenv = require('dotenv');

dotenv.config();

const app = express();
app.use(cors());
app.use(express.json());

mongoose.connect(process.env.MONGO_URI)
  .then(() => {
    console.log('MongoDB connected');
  })
  .catch(err => console.log('MongoDB connection error:', err));

app.use('/api/auth', require('./routes/auth'));
app.use('/api/appointments', require('./routes/appointments'));
app.use('/api/admin', require('./routes/admin'));
```

```
const PORT = process.env.PORT || 5000;

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

Future Scope of Project

1. **SMS / Email Reminders & Two-Way Notifications**
    - Rationale: Reduces no-shows and improves engagement.
    - Implement: Integrate SMTP/SendGrid and an SMS gateway (Twilio/Razorpay SMS). Use background jobs (Bull / Agenda) for scheduled reminders.
    - Metrics: % reduction in no-shows, open/response rate.

2. **Improved Role-based Dashboards**
    - Rationale: Faster workflows for doctors and admins.
    - Implement: Add calendar views (react-big-calendar), quick actions for approve/reject, and filters.
    - Metrics: Average time to approve/reject, user satisfaction.

3. **Appointment Rescheduling & Cancellation Workflow**
    - Rationale: User convenience and fewer manual changes.
    - Implement: Allow patient-initiated reschedules with notifications and automatic slot validation.
    - Metrics: % reschedules handled without admin intervention.

4. **Payment & Billing Integration**

    - Rationale: Monetization (consultation fees) and revenue tracking.
    - Implement: Integrate Stripe/PayPal/Razorpay with server-side verification & webhook handling; store transactions in `payments`.
    - Metrics: Payment success rate, average time to reconcile.