HEALTHCARE APPOINTMENT MANAGEMENT SYSTEM

23CS503 - APPLICATION DEVELOPMENT

A PROJECT REPORT

Submitted by

JAIRUS RAJ SINGH S - 727823TUCS116

in partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

SRI KRISHNA COLLEGE OF TECHNOLOGY
An Autonomous Institution | Accredited by NAAC with 'A' Grade
Affiliated to Anna University | Approved by AICTE
KOVAIPUDUR, COIMBATORE 641042

AUGUST 2025

SRI KRISHNA COLLEGE OF TECHNOLOGY
An Autonomous Institution | Accredited by NAAC with 'A' Grade
Affiliated to Anna University | Approved by AICTE
KOVAIPUDUR, COIMBATORE 641042

BONAFIDE CERTIFICATE

Certified that this project report "**HEALTHCARE APPOINTMENT MANAGEMENT SYSTEM**" is the bonafide work of **"JAIRUS RAJ SINGH S"** who carried out the project work under my supervision.

SIGNATURE

Dr. M. KAVITHA MARGRET

SUPERVISOR

Assistant Professor,
Department of Computer Science and Engineering
Sri Krishna College of Technology,
Coimbatore - 641042.

SIGNATURE

Dr. M. UDHAYAMOORTHI

HEAD OF THE DEPARTMENT

Associate Professor,
Department of Computer Science and Engineering
Sri Krishna College of Technology,
Coimbatore - 641042.

Certified that the candidates were examined by us in the Project Viva Voce examination held on_____at Sri Krishna College of Technology, Kovaipudur, Coimbatore - 641042

INTERNAL EXAMINER **EXTERNAL EXAMINER**

ACKNOWLEDGEMENT

First and foremost, we thank the **Almighty** for being our light and for showering his gracious blessings throughout the course of this project.

We express our gratitude to our beloved Principal, **Dr. M.G. Sumithra,**
for providing all facilities.

With the grateful heart, our sincere thanks to our Head of the Department **Dr. M. Udhayamoorthi**, Department of Computer Science and Engineering for the motivation and all support to complete the project work.

We are greatly indebted to our Industry Mentor **Ms.S.Nandhini** for her valuable guidance and suggestions in all aspects that aided us to ameliorate our skills.

We thank **Dr. M. Kavitha Margret,** Department of Computer Science and Engineering, for her motivation and support.

We are thankful to all the **Teaching and Non-Teaching Staff** of Department of Computer Science and Engineering and to all those who have directly and indirectly extended their help to us in completing this project work successfully.

We extend our sincere thanks to our family members and our beloved friends, who had been strongly supporting us in all our endeavor.

ABSTRACT

The Healthcare Appointment Management System harnesses modern web technologies like React JS, Spring Boot, and REST API to deliver a comprehensive platform that transforms the way medical appointments are managed. It provides an intuitive and user-friendly interface for patients, doctors, and administrators to efficiently handle registrations, appointment bookings, scheduling, and status updates. Core functionalities include patient and doctor management, appointment requests, approval workflows, and real-time status tracking, ensuring a smooth and transparent healthcare process.

Administrators benefit from a feature-rich dashboard to seamlessly monitor bookings, track doctor availability, and generate insightful reports on appointment history and service demand trends. Doctors can easily view their assigned appointments, approve or reject requests, and update statuses such as approved, rejected, or completed through a responsive interface. Patients can conveniently book appointments, view confirmations, and receive immediate notifications regarding approvals or schedule changes. Automated validation and conflict prevention ensure accuracy and reliability, while streamlined workflows reduce errors and enhance trust.

By leveraging React JS's dynamic frontend and Spring Boot's robust RESTful APIs, the system enhances efficiency by automating routine tasks and optimizing scheduling processes. The scalable design supports seamless expansion to accommodate growing healthcare organizations, while advanced reporting empowers administrators to make data-driven decisions for better resource utilization. In conclusion, the Healthcare Appointment Management System is a state-of-the-art solution designed to meet the evolving needs of modern healthcare, providing a reliable, efficient, and patient-focused service platform.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

Abbreviation	Acronym
HTML	HYPERTEXT MARKUP LANGUAGE
CSS	CASCADING STYLESHEET
JS	JAVASCRIPT
SDLC	SOFTWARE DEVELOPMENT LIFE CYCLE

CHAPTER 1

INTRODUCTION

In this chapter, we will present the problem statement, provide an overview, and outline the main objectives of the Healthcare Appointment Management System.

1.1 PROBLEM STATEMENT

Manual healthcare appointment booking causes delays, errors, and miscommunication between patients and doctors. A digital system is needed to make booking easy for patients, help doctors manage schedules, and allow administrators to track appointments efficiently.

1.2 OVERVIEW

The Healthcare Appointment Management System is a web application that helps patients book appointments online, doctors manage their schedules, and administrators track all bookings in one place. Built with React JS and Spring Boot, it provides easy registration, appointment booking, approval, and status updates. The system reduces errors, saves time, and improves communication between patients and doctors.

1.3 OBJECTIVE

The primary objective of this project is to develop a Healthcare Appointment Management System that provides patients, doctors, and administrators with a centralized platform for managing medical appointments. The system seeks to:

- Simplify patient and doctor registration.
- Enable easy and secure online appointment booking.
- Enhance communication between patients and doctors through real-time updates.
- Improve healthcare service delivery by reducing errors and saving time.

CHAPTER 2
SYSTEM SPECIFICATION

In this chapter, we are going to see the software that we have used to build the website. This chapter gives you a small description about the software used in the project.

2.1 VS CODE

Visual Studio Code is a source code editor developed by Microsoft for Windows, Linux, and macOS. It includes support for debugging, embedded Git control, syntax highlighting, intelligent code completion, snippets, and code refactoring. It is also customizable, so users can change the editor's theme, keyboard shortcuts,and preferences.

VS Code is an excellent code editor for React projects. It is lightweight, customizable, and has a wide range of features that make it ideal for React development. It has built-in support for JavaScript, JSX, and TypeScript, and enables developers to quickly move between files and view detailed type definitions. It also has a built-in terminal for running tasks. Additionally, VS Code has an extensive library of extensions that allow developers to quickly add features like code snippets, debugging tools, and linting supportto their projects.

2.2 LOCAL STORAGE

Local storage is a type of web storage for storing data on the client side of a web browser. It allows websites to store data on a user's computer, which can then be accessed by the website again when the user returns. Local storage is a more secure alternative to cookies because it allows websites to store data without having to send it back and forth with each request. Local storage is a key-value pair storage mechanism, meaning it stores data in the form of a key and corresponding value. It is similar to a database table in that it stores data in columns and rows, except that local storage stores the data in the browser rather than in a database. Local storage is often used to store user information such as

preferences and settings, or to store data that is not meant to be shared with other websites. It is also used to cache data to improve the performance of a website. Local storage is supported by all modern web browsers, including Chrome,

Firefox, Safari, and Edge. It is accessible through the browser's JavaScript API. Local storage is a powerful tool for websites to store data on the client side. It is secure, efficient, and can be used to store data that does not need to be shared with other websites.

Local Storage is a great way to improve the performance of a website by caching data. Local storage in web browsers allows website data to be stored locally on the user's computer. It is a way of persistently storing data on the client side, which is not sent to the server with each request. This allows users to store data such as preferences, login information, and form data without needing to send it to a server. It is typically stored in a browser's cookie file, but it can also be stored in other locations such as HTML5 Local Storage and Indexed DB. The data stored in local storage is persistent and can be accessed by the website even if the user closes the browser or navigates to another page. It is a great way for websites to store user-specific data, as it is secure, reliable, and fast. It is also a great way for developers to store data that does not need to be sent to the server with each request.

One of the key benefits of using local storage is its reliability. Unlike server-side storage, which can be affected by network outages or other server issues, local storage is stored locally on the user's machine, and so is not affected by these issues. Another advantage of local storage is its speed. Because the data is stored locally, it is accessed quickly, as there is no need to send requests to a server. This makes it ideal for storing data that needs to be accessed quickly, such as user preferences or session data. Local storage is also secure, as the data is stored on the user's machine and not on a server. This means that the data is not accessible by anyone other than the user, making it a good choice for storing sensitive information.

CHAPTER 3

PROPOSED SYSTEM

This chapter provides a brief description of the proposed concept and functionality behind the development of the Healthcare Appointment Management System.

3.1 PROPOSED SYSTEM

The Healthcare Appointment Management System provides a centralized digital platform that streamlines the process of scheduling, managing, and tracking medical appointments. It enables patients, doctors, and administrators to interact through a user-friendly interface, ensuring transparency and efficiency in healthcare services. Patients can register, browse available doctors, and book appointments online, while doctors can manage their schedules, approve or reject bookings, and update appointment status in real time.

The system automates key processes such as validation of patient/doctor details, appointment conflict checks, and real-time status updates. Once an appointment request is made, it is automatically routed to the selected doctor, who can take action immediately. Administrators can monitor all appointments, track doctor availability, and generate reports on appointment history and service demand.

By replacing manual scheduling with an automated solution, the system reduces delays, prevents booking conflicts, and ensures accurate record-keeping. Patients benefit from easy access to appointment booking and instant confirmation, doctors gain efficient schedule management tools, and administrators have complete oversight of operations. Additionally, the system securely stores patient and appointment records, ensuring data integrity and compliance with healthcare standards.

An added advantage of the system is **scalability**, allowing it to support growing numbers of patients, doctors, and healthcare organizations without compromising performance.

3.2 ADVANTAGES

- Efficiency: The system automates patient registration, doctor scheduling, and appointment booking, reducing time and effort for both staff and patients.
- Real-Time Updates: Patients and doctors receive instant notifications about booking confirmations, approvals, or changes, improving communication and trust.
- Booking Flexibility: Patients can book appointments anytime, schedule future visits, or request urgent consultations, ensuring convenience.
- Conflict Prevention: Automated validation prevents double bookings and overlapping schedules, resulting in smooth operations.
- Accessibility: Patients, doctors, and administrators can access the system from any location with internet access via a web interface.
- Transparency: The platform maintains clear records of patients, doctors, and appointments, ensuring accountability and service quality.
- Data-Driven Decisions: Built-in reporting and analytics allow administrators to analyze appointment trends, doctor availability, and service demand.
- Scalability: The system can easily expand to handle more patients, doctors, and healthcare centers as the organization grows.

CHAPTER 4

METHODOLOGIES

System Workflow

1. **Appointment Management (Doctor/Admin)**
    1. Doctors and admins create, update, or cancel appointments by inputting detailed information: patient ID, doctor ID, date, time, duration, reason for visit, and status (e.g., requested, approved, completed).
    2. The backend performs real-time validation (e.g., checking for overlapping schedules, valid patient/doctor IDs) and assigns a unique appointment ID before saving to the database.
    3. APIs provide functionalities to list all appointments with pagination, filter by date/status, or retrieve detailed views including patient history and doctor notes.
2. **Patient Registration**
    1. Patients access the system to browse available doctors and time slots, selecting an appointment by entering their patient ID and preferred details.
    2. The backend conducts multiple checks:
        1. Capacity: If the doctor's schedule is full, it returns "Event is at full capacity."
        2. Duplicates: If the patient is already registered for that slot, it displays "Student already registered for this event."
        3. Validity: If the patient or doctor ID is invalid, it shows "Not Found."
    3. Upon success, the registration is timestamped and stored, with a confirmation sent to the patient (e.g., via email or dashboard notification).
3. **Patient Dashboard**
    1. Patients log in with their ID to view a personalized dashboard listing all booked appointments, including dates, times, doctors, and statuses.
    2. If no appointments exist, a clear message "No appointments found" is displayed, encouraging new bookings.
    3. Additional features may include rescheduling options or viewing past visit summaries.

Technical Methodologies

- **Backend (Spring Boot)**
    - Implements RESTful APIs for CRUD operations on appointments, doctors, and patients.
    - Defines entities (e.g., Patient with fields: ID, name, contact; Doctor with ID, name, specialization; Appointment with ID, patientID, doctorID, dateTime, status) using JPA for relational mapping.
    - Employs Hibernate Validator for input constraints (e.g., date formats, required fields) and centralized exception handling to return meaningful error messages (e.g., JSON responses with error codes).
    - Supports authentication and authorization to restrict access (e.g., only doctors can approve appointments).
- **Frontend (React.js)**
    - Utilizes hooks like useState for form state and useEffect for API calls, ensuring dynamic updates.
    - Integrates Fetch API (or Axios if refactored) for HTTP requests, with error handling displayed via toast notifications or modals.
    - Features a responsive UI with Tailwind CSS for styling, including card-based appointment lists, dropdown filters (e.g., by doctor or date), and a detailed view page with a prominent "Book Appointment" button.
    - Implements client-side validation to enhance user experience before API calls.
- **Integration & Security**

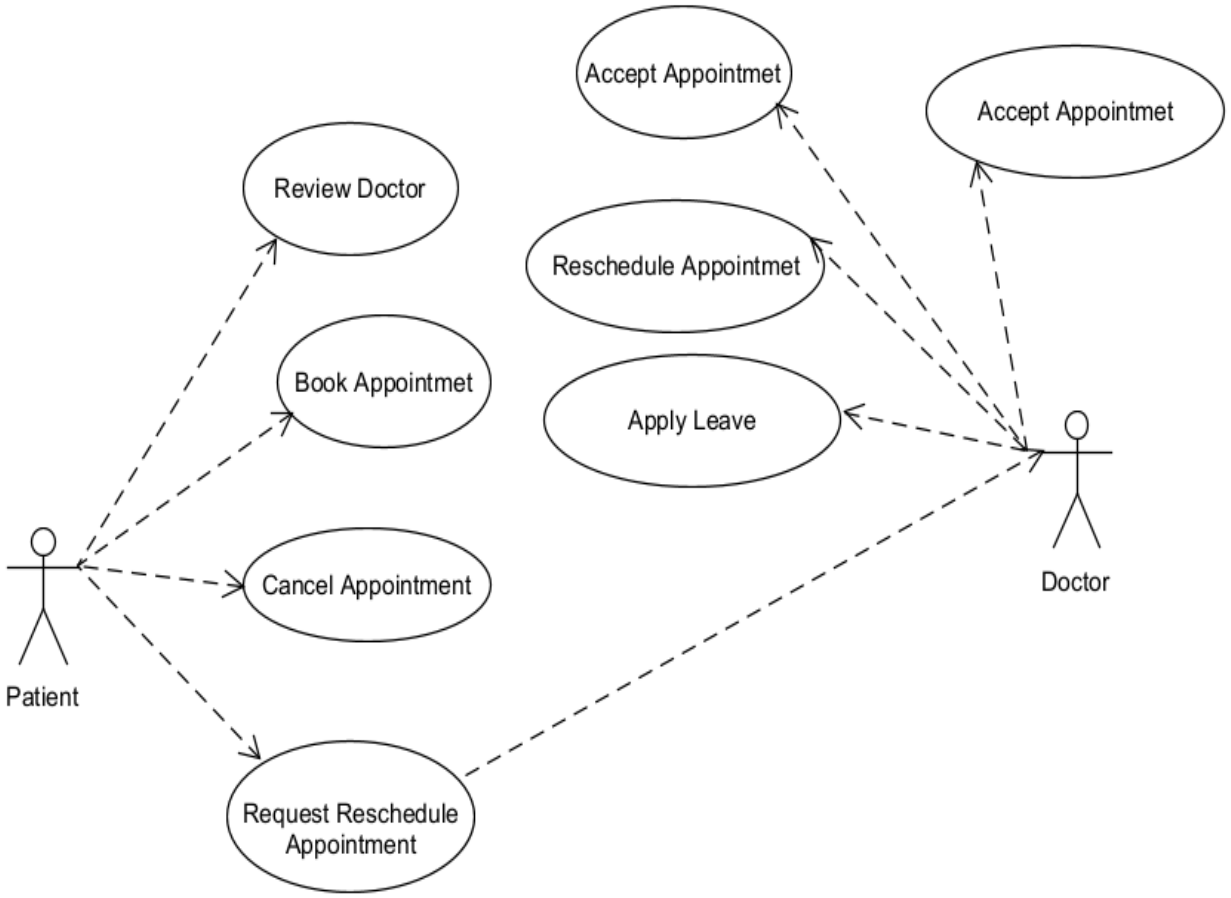
- Enables CORS to facilitate secure communication between the frontend (Port 8081) and backend (Port 8080).
- Combines client-side (JavaScript) and server-side (Spring Boot) validation to prevent invalid data submission.
- Provides user-friendly error messages (e.g., "Invalid time slot") and logs errors server-side for debugging.

Use Case Diagram:

· Defines functional scope with actors (patients, doctors).

· Highlights roles: patients book appointments, doctors manage them.

· Serves as a blueprint for development and stakeholder understanding.



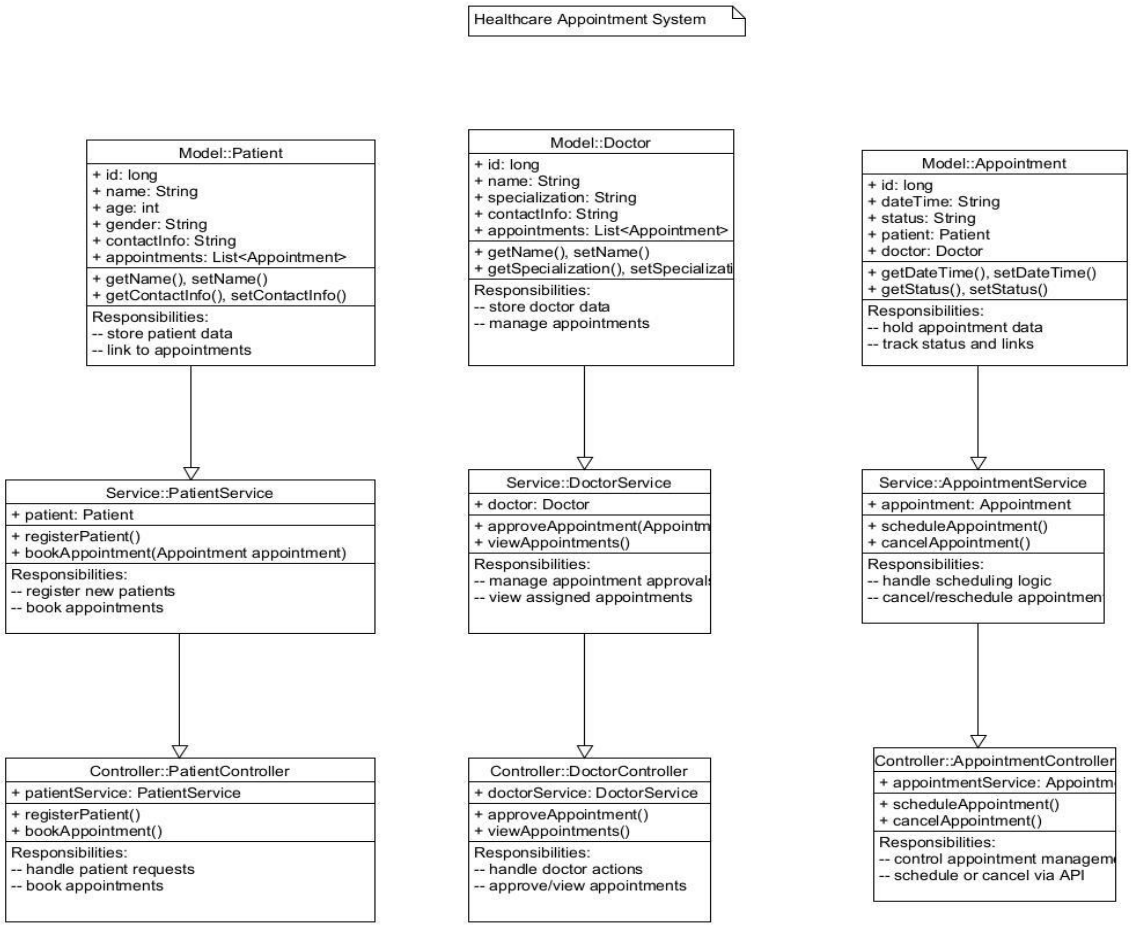
4.2 Use Case Diagram

Class Diagram:

· Defines classes: Patient (attributes: ID, name, contact; methods: bookAppointment), Doctor (ID, name, specialization; methods: approveAppointment), Appointment (ID, patient, doctor, dateTime, status), and Feedback (ID, rating, comment).

· Establishes relationships: One patient to many appointments, one doctor to many appointments, and one appointment to one feedback.

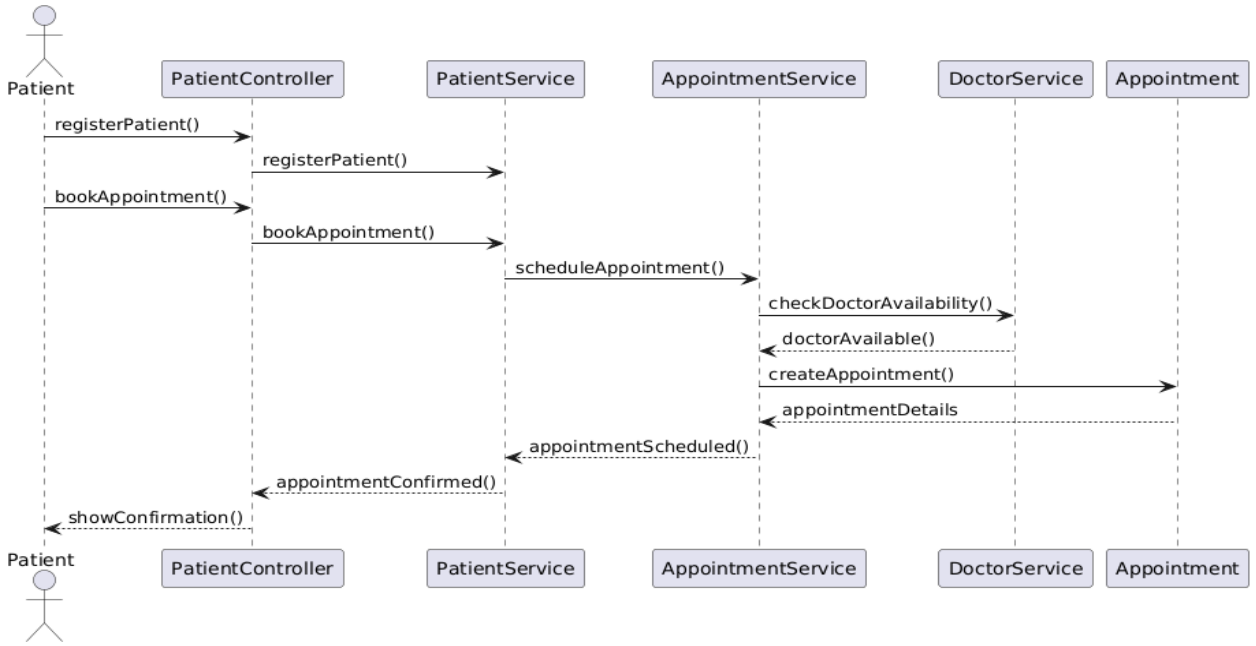
· Ensures structured data management, supporting complex queries (e.g., doctor availability).



4.3 Class Diagram

Sequence Diagram:

A patient logs in via the frontend, which sends credentials to the backend for database validation. For viewing appointments, the frontend requests data, the backend queries the database, and the response is displayed. For booking, the frontend submits a request, the backend processes it (with validation), saves to the database, and confirms. Doctors follow a similar flow for managing appointments, with the backend handling updates and notifications.

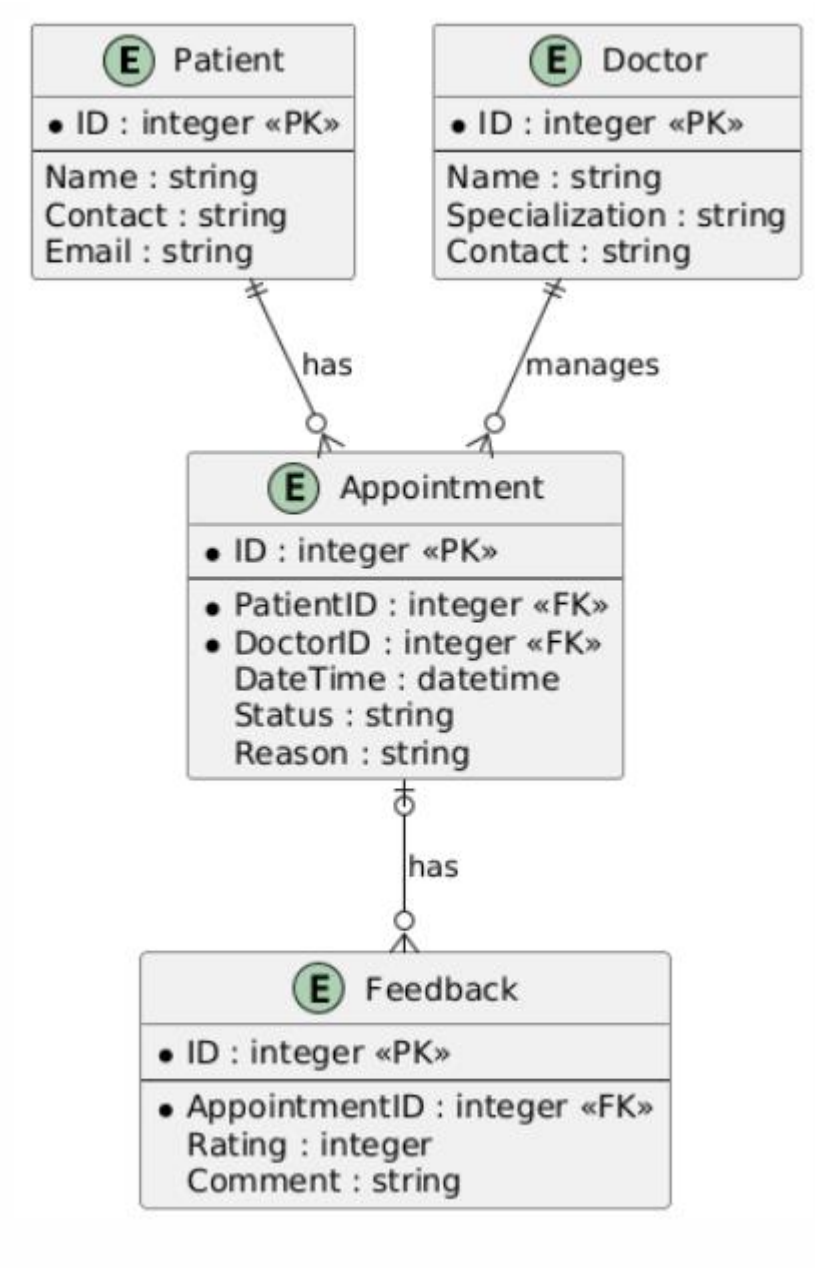


4.4 Sequence Diagram

ER Diagram:

The **ER (Entity-Relationship) Diagram** for the Healthcare Appointment Management System provides a concise representation of its data structure. It includes key entities such as Patient, with attributes like ID (primary key), Name, Contact, and Email, and Doctor, featuring ID (primary key), Name, Specialization, and Contact. The Appointment entity connects these, with attributes including ID (primary key), PatientID and DoctorID (foreign keys), DateTime, Status, and Reason, while Feedback, linked to Appointment, contains ID (primary key), AppointmentID (foreign key), Rating, and Comment.

The relationships are straightforward yet effective: a Patient can have multiple Appointments in a one-to-many relationship, and a Doctor can similarly manage multiple Appointments. Additionally, each Appointment can have one Feedback entry, forming a one-to-one relationship. These connections are enforced with primary and foreign keys to ensure data integrity, while an index on the DateTime field in Appointments optimizes query performance. This diagram serves as the backbone for managing appointments, tracking patient-doctor interactions, and capturing feedback efficiently.



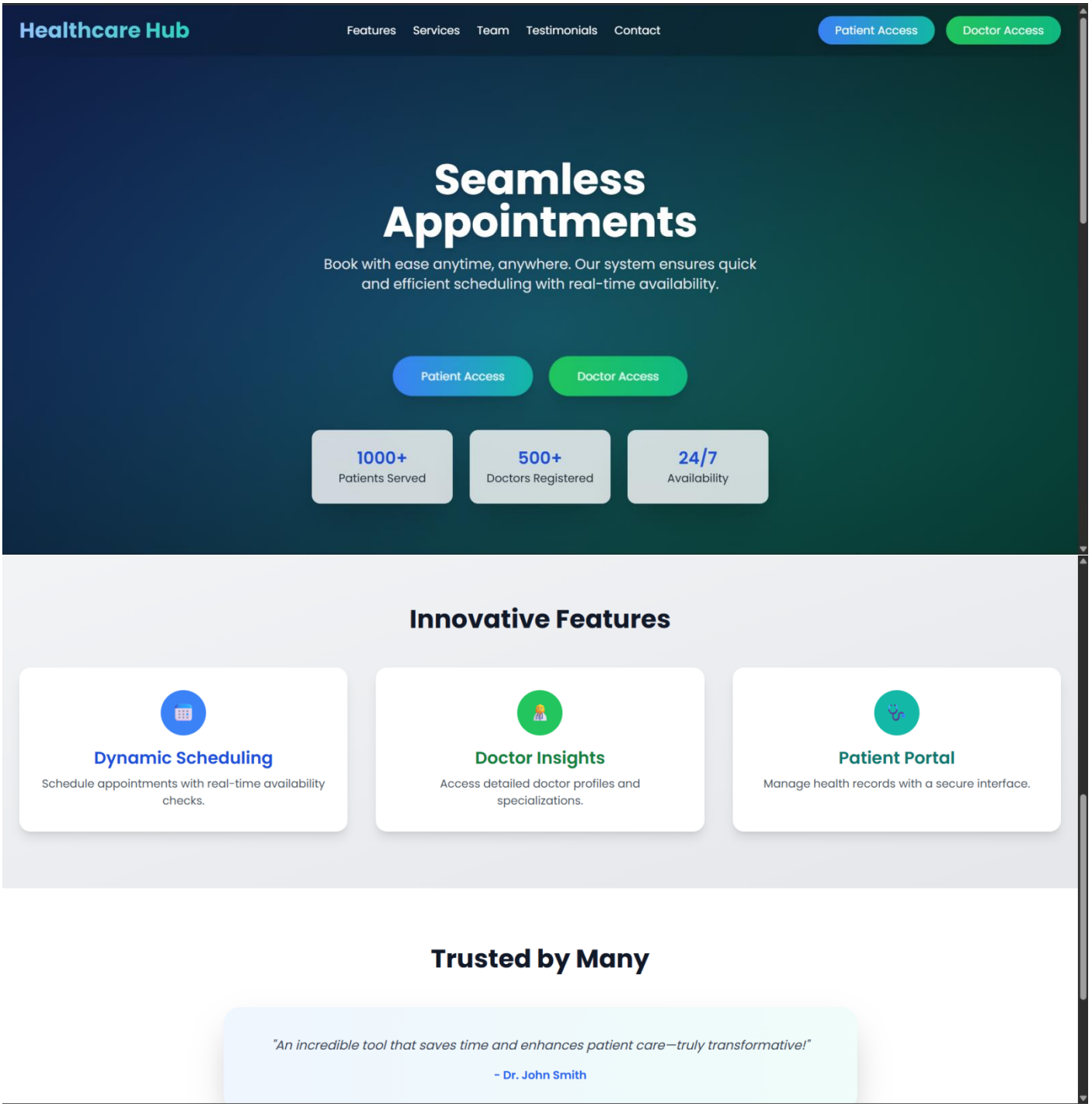
4.5 Entity Relationship Diagram

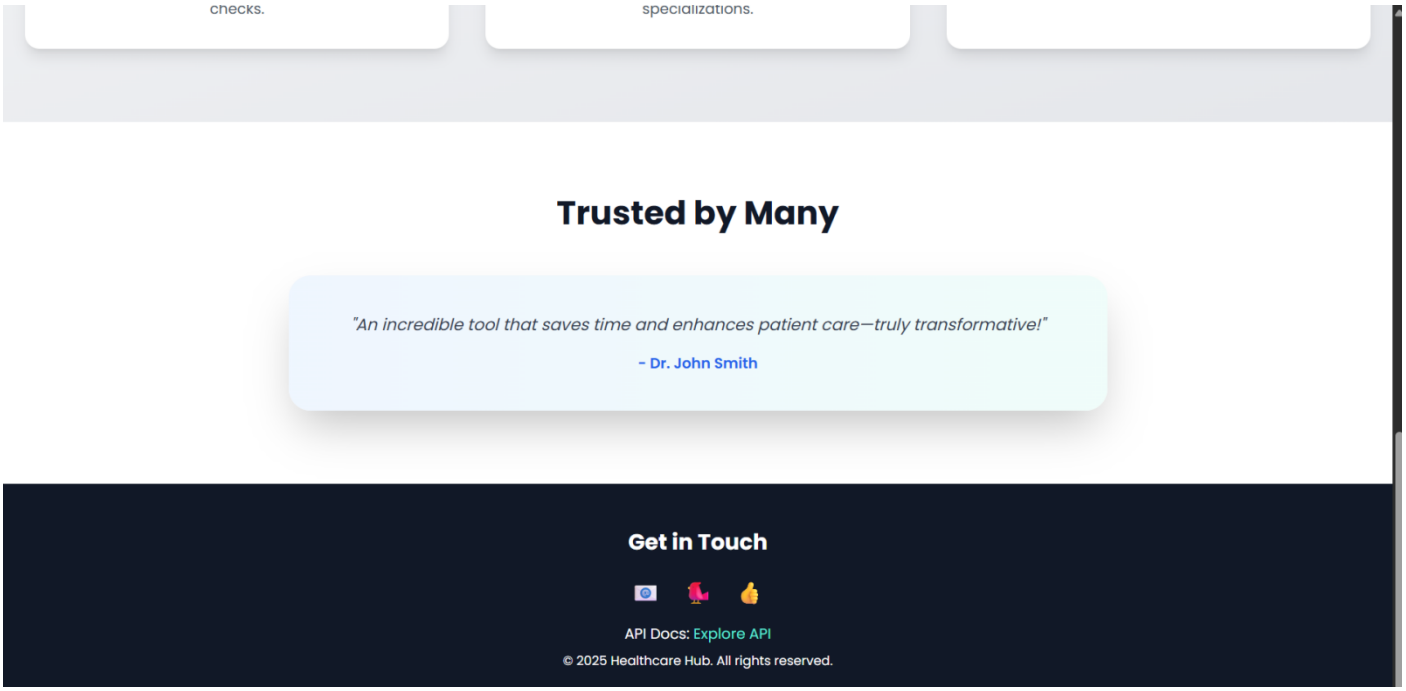
CHAPTER 5

IMPLEMENTATION AND RESULT

This chapter gives a description about the output that we produced by developing the website of our idea.

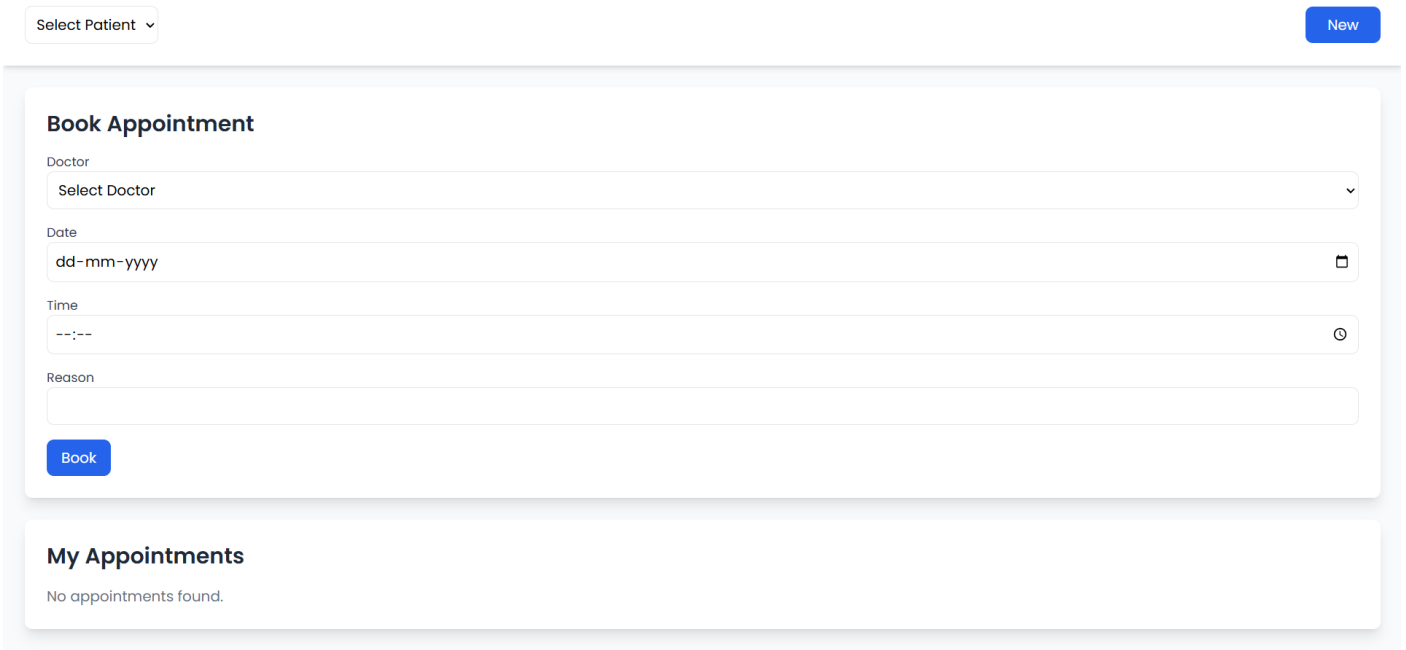
5.1 Home page:





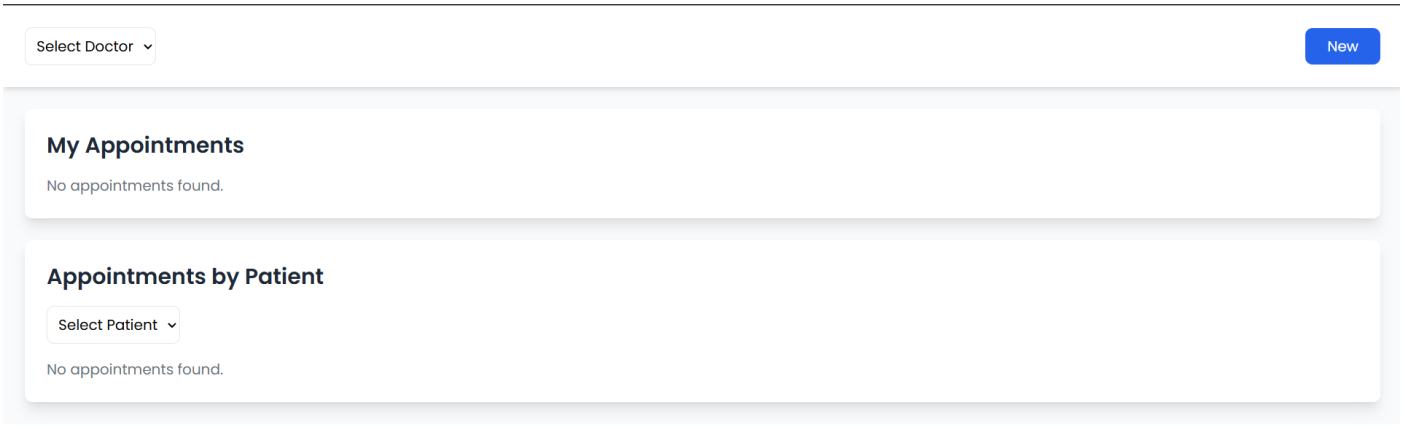
5.1 Home Page

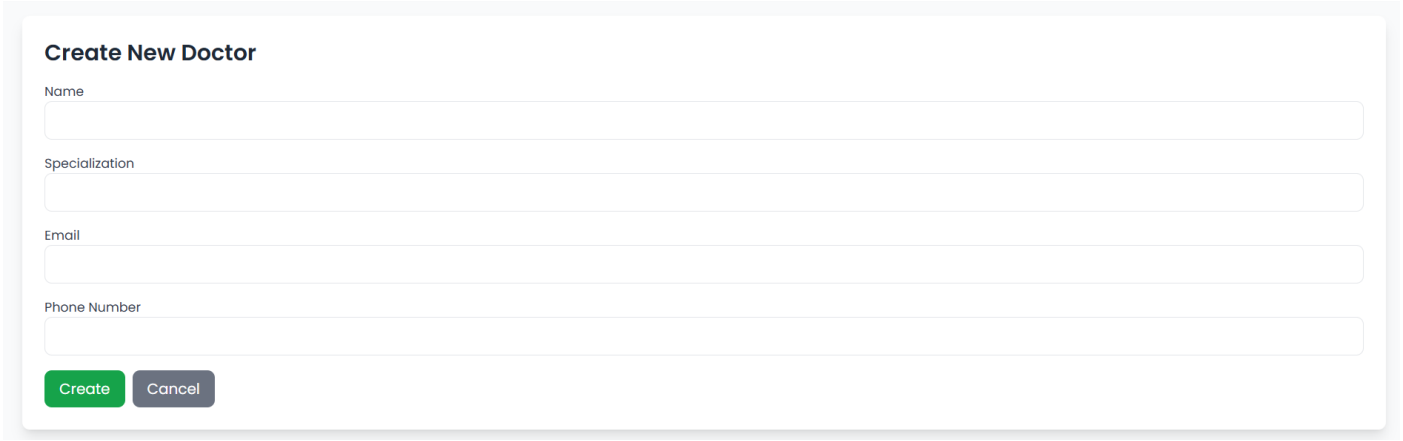
5.2 Patient Acess Page:



5.2 Patient Acess Page

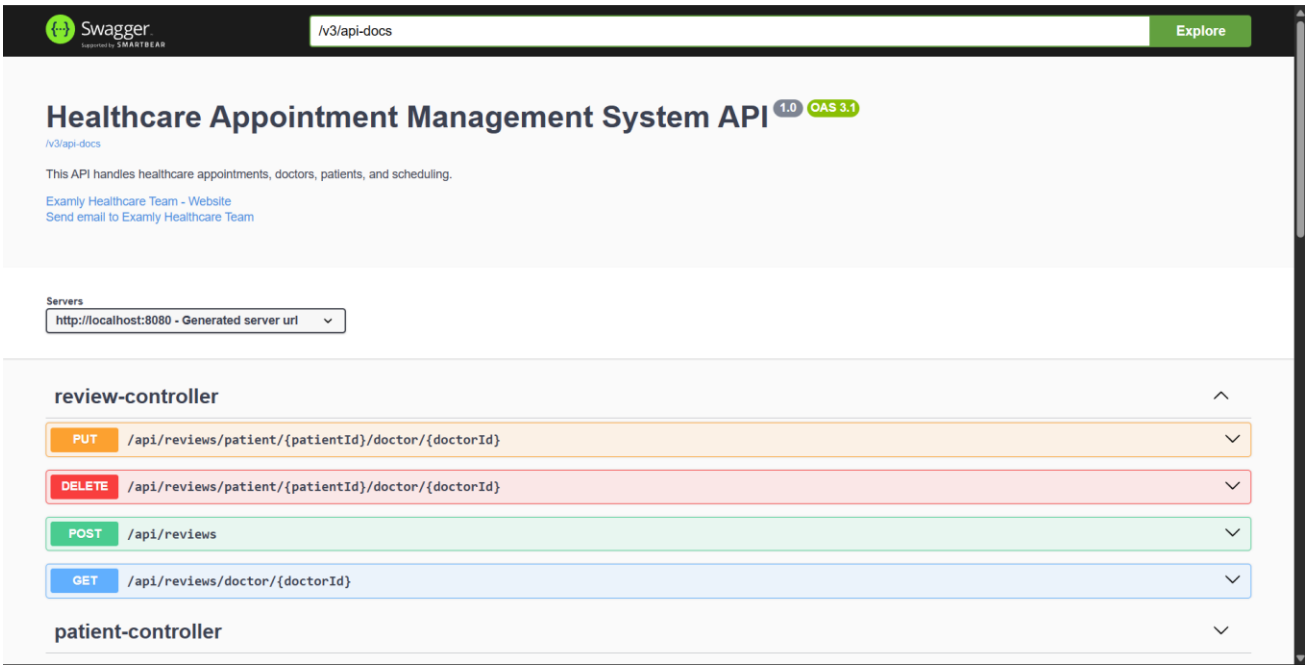
5.3 Doctor Access Page

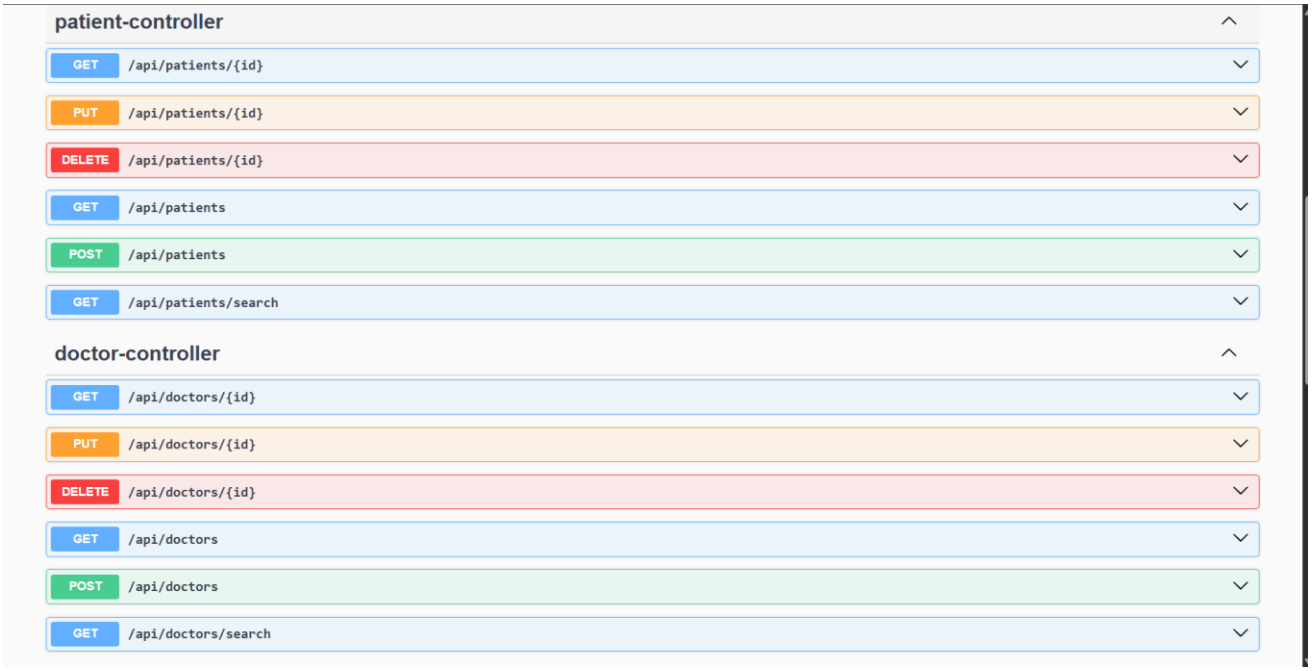




5.3 Doctor Acess Page

5.4 Swagger


Swagger
/v3/api-docs
Explore
Healthcare Appointment Management System API
1.0
OAS 3.1
/v3/api-docs
This API handles healthcare appointments, doctors, patients, and scheduling.
Examly Healthcare Team - Website
Send email to Examly Healthcare Team
Servers
http://localhost:8080 - Generated server url
review-controller
PUT /api/reviews/patient/{patientId}/doctor/{doctorId}
DELETE /api/reviews/patient/{patientId}/doctor/{doctorId}
POST /api/reviews
GET /api/reviews/doctor/{doctorId}
patient-controller


patient-controller
GET /api/patients/{id}
PUT /api/patients/{id}
DELETE /api/patients/{id}
GET /api/patients
POST /api/patients
GET /api/patients/search
doctor-controller
GET /api/doctors/{id}
PUT /api/doctors/{id}
DELETE /api/doctors/{id}
GET /api/doctors
POST /api/doctors
GET /api/doctors/search


POST /api/doctors
GET /api/doctors/search
auth-controller
POST /api/auth/signup/patient
POST /api/auth/signup/doctor
POST /api/auth/logout
POST /api/auth/login
appointment-controller
GET /api/appointments
POST /api/appointments
PATCH /api/appointments/{id}/status
GET /api/appointments/{id}
GET /api/appointments/patient/{patientId}
GET /api/appointments/doctor/{doctorId}

5.4 Swagger

5.5 Coding:

Frontend:

AppointmentForm.js:

```
import React, { useEffect, useState } from "react";
import {
   fetchPatients,
   fetchDoctors,
   createAppointment,
} from "../utils/api";

import "./AppointmentForm.css";
function normalizeTimeToHMS(t) {
   if (!t) return t;

   if (/^\d{2}:\d{2}$/.test(t)) return `${t}:00`;
   return t;
}



export default function AppointmentForm() {
   const [patients, setPatients] = useState([]);
   const [doctors, setDoctors] = useState([]);

   const [loading, setLoading] = useState(true);
  const [form, setForm] = useState({

      patient: "",

      doctor: "",

      date: "",

      time: "",

      reason: "",

            doctor: "",

            date: "",

            time: "",
            reason: "", });
      } catch (e2) {

         setServerError(e2?.message || "Server error");
```

```
    }};

if (loading) {
    return <div>Loading...</div>;  }

return (
     <div  className="appointment-form-container">
       <h2 className="form-title">Book Appointment</h2>
       <form  onSubmit={handleSubmit}  className="appointment-form">
          <div  className="form-group">
             <label  htmlFor="patient-select"  className="form-label">Patient</label>
             <input
                id="patient-select"
                data-testid="patient-select"
                name="patient"
                value={form.patient}
               onChange={handleChange}
               placeholder="Type patient name"
                className="form-input" />
             {errors.patient && <div className="form-error">{errors.patient}</div>}
          </div> <div className="form-group">
             <label  htmlFor="doctor-select"  className="form-label">Doctor</label>
             <select
});

const [errors, setErrors] = useState({});
const [message, setMessage] = useState("");
 const [serverError, setServerError] = useState("");

useEffect(() => {
    let mounted = true;
    (async () => {
       try {
          const [p, d] = await Promise.all([fetchPatients(), fetchDoctors()]);
          if (mounted) {
             setPatients(p || []);
```

```
            setDoctors(d || []);
          }

        } catch (_) {

          if (mounted) {
            setPatients([]);
            setDoctors([]);

          }

        } finally {

          if (mounted) setLoading(false);

        }  })();
  return () => {
mounted = false; }  }, []);


  const handleChange = (e) => {
    const { name, value } = e.target;
     setForm((prev) => ({ ...prev, [name]: value }));

    setErrors((prev) => ({ ...prev, [name]: "" }));
    setMessage("");

    setServerError("");  };


  const validate = () => {
    const err = {};
    if (!form.patient) err.patient = "Patient is required";
    if (!form.doctor) err.doctor = "Doctor is required";
    if (!form.date) err.date = "Date is required";
    if (!form.time) err.time = "Time is required";

    if (!form.reason || !form.reason.trim()) err.reason = "Reason is required";
    return err;   };

  const handleSubmit = async (e) => {
    e.preventDefault();
    setMessage("");
    setServerError("");
    const err = validate();
    setErrors(err);
    if (Object.keys(err).length > 0) return;
    const payload = {
     patientId: parseInt(form.patient, 10),
    doctorId: parseInt(form.doctor, 10),
    appointmentDate: form.date,
   appointmentTime: normalizeTimeToHMS(form.time),
reason: form.reason.trim(), };
```

```
                id="doctor-select"

                data-testid="doctor-select"
                name="doctor"
                value={form.doctor}
                onChange={handleChange}
                className="form-select" >
                <option value="">Select doctor</option>

                {doctors.map((d) => (

                  <option key={d.id} value={String(d.id)}>

                    {d.name} {d.specialization ? `${d.specialization}` : ""}

                  </option>   ))}

              </select>

              {errors.doctor && <div className="form-error">{errors.doctor}</div>}

            </div>


            <div className="form-group">

              <label htmlFor="date-input" className="form-label">Date</label>

              <input

                id="date-input"

                data-testid="date-input"
                type="date"
                name="date"
               value={form.date} onChange={handleChange}
 className="form-input  />

 {errors.date && <div className="form-error">{errors.date}</div>      </div>


 <div  className="form-group">

  <label htmlFor="time-input" className="form-label">Time</label>  <input
    id="time-input"
    data-testid="time-input"

                type="time"
                name="time"
                value={form.time}
                onChange={handleChange}
                className="form-input"/>
              {errors.time && <div className="form-error">{errors.time}</div>}

            </div>
```

```
        <div className="form-group">

          <label htmlFor="reason-input" className="form-label">Reason</label>

          <input

            id="reason-input"

            data-testid="reason-input"
            type="text"
            name="reason"
            value={form.reason}
            onChange={handleChange}
            placeholder="Enter  reason"
            className="form-input" />
          {errors.reason && <div className="form-error">{errors.reason}</div>}

        </div>

 <button type="submit" className="submit-btn">Book Appointment</button> </form>
{message && <div className="form-message success-message">{message}</div>}

        {serverError && <div className="form-message error-message">{serverError}</div>}

      </div>  );}
```

ApplicationList.js:

```
import React, { useEffect, useState } from "react";

import { fetchPatients, fetchAppointmentsByPatient } from "../utils/api";
import './AppointmentList.css';

export default function AppointmentList() {
   const [patients, setPatients] = useState([]);
   const [loadingPatients, setLoadingPatients] = useState(true);


   const [appointmentsByPatient, setAppointmentsByPatient] = useState({});
   const [loadingAppointments, setLoadingAppointments] = useState(false);

   useEffect(() => {


      let mounted = true;


      (async () => { try {
```

```
            const p = await fetchPatients();


            if (mounted) setPatients(p || []);
        if (mounted) {

            setLoadingAppointments(true);

                const appointmentsData = await Promise.all(

                    (p || []).map((patient) => fetchAppointmentsByPatient(patient.id))  );


                if (mounted) {
                    const map = {};
                    (p || []).forEach((patient, index) => {

                        map[patient.id] = appointmentsData[index] || [];

                    });

                    setAppointmentsByPatient(map);}    }

            } catch (e) {

                if (mounted) {
                    setPatients([]);
                    setAppointmentsByPatient({});  }

            } finally {

                if (mounted) {
                    setLoadingPatients(false);
                    setLoadingAppointments(false);  }     }  })();


        return () => {

            mounted = false;  };     }, []);


    if (loadingPatients || loadingAppointments) {

        return <div className="loading-container">Loading...</div>;}


    const allPatientsHaveNoAppointments = patients.length > 0 && patients.every(patient =>
(appointmentsByPatient[patient.id] || []).length === 0);
  return (<div className="appointment-list-container">
 <h2 className="list-title">Appointments</h2>
```

```
<div style={{ display: 'none' }}>
            <label htmlFor="patient-filter" className="filter-label">Filter by Patient</label>
            <select id="patient-filter" data-testid="patient-filter" disabled>
               <option value="">Select patient</option>
               {patients.map((p) => (
                  <option key={p.id} value={String(p.id)}>
                     {p.name}
                  </option> ))}
            </select>
         </div>

         <table className="appointments-table" border="1" cellPadding="5" style={{ width:
'100%', borderCollapse: 'collapse' }}>
            <thead>
<tr>
                  <th>Patient Name</th>
                  <th>Appointment ID</th>
                  <th>Date & Time</th>
                  <th>Status</th>
                  <th>Reason</th>
               </tr>
            </thead>
            <tbody>
               {patients.length === 0 && (
                  <tr>
                     <td colSpan="5" style={{ textAlign: "center" }}>No patients found</td>
                  </tr>    )}
```

```
                {patients.map((patient) => {

                  const appts = appointmentsByPatient[patient.id] || [];
                  if (appts.length === 0) {
                    return (

                      <tr key={`no-appt-${patient.id}`}>

                        <td>{patient.name}</td>

                        <td colSpan="4" style={{ textAlign: "center" }}>
                          No appointments found      </td>    </tr>       }
                  return appts.map((a, idx) => (

                    <tr key={a.id}>

                      {idx === 0 && (

                        <td rowSpan={appts.length} style={{ verticalAlign: "middle" }}>

                          {patient.name}

                        </td>   )}

                      <td>{a.id}</td>

                      <td>{a.appointmentDate} {a.appointmentTime}</td>

                      <td>{a.status}</td>       <td>{a.reason}</td>       </tr>
   ));   })}
  </tbody>
  </table> </div> ); }
```

App.js:

```
import React, { useState } from "react";

import AppointmentForm from "./components/AppointmentForm";
import AppointmentList from "./components/AppointmentList";

import HomePage from "./components/HomePage";
import "./App.css";

export default function App() {

  const [activeTab, setActiveTab] = useState("home"); // default: Home page


  return (
```

```
  <div className="app-container">
    <header className="app-header">
      <h1 className="app-title">CureConnect</h1>
      <nav className="navbar">
        <button
          className={`nav-btn ${activeTab ===  "home" ? "active" : ""}`}
          onClick={() => setActiveTab("home")  >
          Home
        </button>
        <button
          className={`nav-btn ${activeTab === "form" ? "active" : ""}`}
          onClick={() => setActiveTab("form")}   >
  Book
    </button>
        <button
          className={`nav-btn ${activeTab === "list" ? "active" : ""}`}
          onClick={() => setActiveTab("list")}  >
          Appointments
        </button>
</nav>

import HomePage from "./components/HomePage";
import "./App.css";

export default function App() {

  const [activeTab, setActiveTab] = useState("home"); // default: Home page


  return (
    <div className="app-container">
      <header className="app-header">
        <h1 className="app-title">CureConnect</h1>
        <nav className="navbar">
```

```
        <button

          className={`nav-btn ${activeTab ===  "home" ? "active" : ""}`}
          onClick={() => setActiveTab("home")  >
          Home

        </button>

        <button

          className={`nav-btn ${activeTab === "form" ? "active" : ""}`}

          onClick={() => setActiveTab("form")}   >
  Book
    </button>

        <button

          className={`nav-btn ${activeTab === "list" ? "active" : ""}`}
          onClick={() => setActiveTab("list")}  >
          Appointments

        </button>
</nav>

  </header>


    <main className="app-main">

      {activeTab === "home" && <HomePage setActiveTab={setActiveTab} />}

      {activeTab === "form" && <AppointmentForm />}

      {activeTab === "list" && <AppointmentList />}

    </main>


    <footer className="app-footer">

      <p>© 2025 CureConnect. All rights reserved.</p>

    </footer>

   </div>  );}
```

\

Index.js:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>);
```

Backend:

Appointment.java:

```
package com.examly.springapp.model;
import jakarta.persistence.*;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.FutureOrPresent;
import lombok.*;

import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;

@Entity
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Appointment {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne

    @JoinColumn(name = "patient_id", nullable = false)
    private Patient patient;

    @ManyToOne
    @JoinColumn(name = "doctor_id", nullable = false)
    private Doctor doctor;

    @NotNull(message = "Appointment date is required")
    @FutureOrPresent(message = "Appointment date must be today or in the future")
    private LocalDate appointmentDate;
```

```
    @NotNull(message = "Appointment time is required")
    private LocalTime appointmentTime;

    @NotBlank(message = "Reason is required")
    private String reason;

    @Enumerated(EnumType.STRING)
    private AppointmentStatus status;
    private LocalDateTime createdAt;
    @PrePersist
    protected void onCreate() {
        createdAt = LocalDateTime.now();
        if (status == null) {
            status = AppointmentStatus.REQUESTED;} }}
```

AppointmentStatus.java:

```
package com.examly.springapp.model;
public enum AppointmentStatus {
    REQUESTED,
    APPROVED,
    COMPLETED,
    CANCELLED,
    RESCHEDULED}
```

Doctor.java:

```
package com.examly.springapp.model;

import com.fasterxml.jackson.annotation.JsonIgnore;
import jakarta.persistence.*;
import jakarta.validation.constraints.*;
import lombok.*;


import java.util.List;

@Entity
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Doctor {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank(message = "Name is required")
    @Size(min = 3, max = 50, message = "Name must be 3-50 characters")
    private String name;
```

```
    @NotBlank(message = "Specialization is required")
    private String specialization;

    @NotBlank(message = "Email is required")
    @Email(message = "Invalid email format")
    private String email;

    @NotBlank(message = "Phone number is required")
    @Pattern(regexp = "\\d{10}", message = "Phone number must be 10 digits")
    private String phoneNumber;

    @OneToMany(mappedBy = "doctor")
    @JsonIgnore
    private List<Appointment> appointments;}
```

Patient.java

```
package com.examly.springapp.model;
import com.fasterxml.jackson.annotation.JsonIgnore;
import jakarta.persistence.*;
import jakarta.validation.constraints.*;
import lombok.*;

import java.time.LocalDate;
import java.util.List;

@Entity
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Patient {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    @NotBlank(message = "Name is required")
    @Size(min = 3, max = 50, message = "Name must be 3-50 characters")
    private String name;

    @NotBlank(message = "Email is required")
    @Email(message = "Invalid email format")
    private String email;

    @NotBlank(message = "Phone number is required")
    @Pattern(regexp = "\\d{10}", message = "Phone number must be 10 digits")
    private String phoneNumber;

    @Past(message = "Date of birth must be in the past")
    @NotNull(message = "Date of birth is required")
    private LocalDate dateOfBirth;
```

```
    @OneToMany(mappedBy = "patient")
    @JsonIgnore
    private List<Appointment> appointments;}
```

AppointmentController.java:

```
package com.examly.springapp.controller;

import com.examly.springapp.model.Appointment;
import com.examly.springapp.model.AppointmentStatus;
import com.examly.springapp.service.AppointmentService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import jakarta.validation.Valid;
import java.time.LocalDate;
import java.time.LocalTime;
import java.util.List;
import java.util.Map;

@RestController
@RequestMapping("/api/appointments")
public class AppointmentController {

    @Autowired
    private AppointmentService appointmentService;
    @PostMapping

    public ResponseEntity<?> bookAppointment(@Valid @RequestBody Map<String,
Object> request) {
        Long patientId = Long.valueOf(request.get("patientId").toString());
        Long doctorId = Long.valueOf(request.get("doctorId").toString());
        LocalDate appointmentDate =
LocalDate.parse(request.get("appointmentDate").toString());
        LocalTime appointmentTime =
LocalTime.parse(request.get("appointmentTime").toString());
        String reason = request.get("reason").toString();

        Appointment savedAppointment = appointmentService.bookAppointment(
                patientId, doctorId, appointmentDate, appointmentTime, reason);
        return new ResponseEntity<>(savedAppointment, HttpStatus.CREATED);   }
    @PatchMapping("/{id}/status")
    public ResponseEntity<Appointment> updateAppointmentStatus(@PathVariable Long id,
            @RequestBody Map<String, String> status) {
        Appointment updatedAppointment = appointmentService.updateAppointmentStatus(
                id, AppointmentStatus.valueOf(status.get("status")));
        return ResponseEntity.ok(updatedAppointment);}
```

```
    @GetMapping("/patient/{patientId}")
    public ResponseEntity<List<Appointment>> getAppointmentsByPatient(@PathVariable
Long patientId) {
        List<Appointment> appointments =
appointmentService.getAppointmentsByPatientId(patientId);
        return ResponseEntity.ok(appointments);}

    @GetMapping("/doctor/{doctorId}")
    public ResponseEntity<List<Appointment>> getAppointmentsByDoctor(@PathVariable
Long doctorId) {
        List<Appointment> appointments =
appointmentService.getAppointmentsByDoctorId(doctorId);
        return ResponseEntity.ok(appointments); }
  @GetMapping("/{id}")
    public ResponseEntity<Appointment> getAppointmentById(@PathVariable Long id) {
        Appointment appointment = appointmentService.getAppointmentById(id);
        return ResponseEntity.ok(appointment); }}
```

DoctorController.java

```
package com.examly.springapp.controller;
import com.examly.springapp.model.Doctor;
import com.examly.springapp.service.DoctorService;
import jakarta.validation.Valid;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;


import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/doctors")
public class DoctorController {

    @Autowired
    private DoctorService doctorService;

    @PostMapping
    public ResponseEntity<Doctor> createDoctor(@Valid @RequestBody Doctor doctor) {
        Doctor savedDoctor = doctorService.createDoctor(doctor);
        return new ResponseEntity<>(savedDoctor, HttpStatus.CREATED);}

    @GetMapping
    public ResponseEntity<List<Doctor>> getAllDoctors() {
        List<Doctor> doctors = doctorService.getAllDoctors();
        return ResponseEntity.ok(doctors);}

    @GetMapping("/{id}")
    public ResponseEntity<Doctor> getDoctorById(@PathVariable Long id) {
```

```
    Doctor doctor = doctorService.getDoctorById(id);
    return ResponseEntity.ok(doctor);  }}
```

PatientController.java

```
package com.examly.springapp.controller;
import com.examly.springapp.model.Patient;
import com.examly.springapp.service.PatientService;
import jakarta.validation.Valid;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/patients")
public class PatientController {

   @Autowired
   private PatientService patientService;
   @PostMapping

   public ResponseEntity<Patient> createPatient(@Valid @RequestBody Patient patient) {
      Patient savedPatient = patientService.createPatient(patient);
      return new ResponseEntity<>(savedPatient, HttpStatus.CREATED); }

   @GetMapping
   public ResponseEntity<List<Patient>> getAllPatients() {
      List<Patient> patients = patientService.getAllPatients();
      return ResponseEntity.ok(patients); }

   @GetMapping("/{id}")
   public ResponseEntity<Patient> getPatientById(@PathVariable Long id) {
      Patient patient = patientService.getPatientById(id);
      return ResponseEntity.ok(patient); }}
```

AppointmentService.java

```
package com.examly.springapp.service;

import com.examly.springapp.exception.AppointmentConflictException;
import com.examly.springapp.exception.ResourceNotFoundException;
import com.examly.springapp.model.Appointment;
import com.examly.springapp.model.AppointmentStatus;
import com.examly.springapp.model.Doctor;
import com.examly.springapp.model.Patient;
import com.examly.springapp.repository.AppointmentRepository;
import  org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
```

```
import  java.time.LocalDate;
import java.time.LocalTime;
import java.util.List;

@Service
public class AppointmentService {

    @Autowired
    private AppointmentRepository appointmentRepository;

    @Autowired
    private PatientService patientService;

    @Autowired
    private DoctorService doctorService;

    public Appointment bookAppointment(Long patientId, Long doctorId, LocalDate
appointmentDate,
            LocalTime appointmentTime, String reason) {

        Patient patient = patientService.getPatientById(patientId);
        Doctor doctor = doctorService.getDoctorById(doctorId);

        // Check for appointment conflicts
        if (appointmentRepository.existsByDoctorAndDateAndTime(doctor, appointmentDate,
appointmentTime)) {
            throw new AppointmentConflictException("The doctor already has an appointment at the
requested time");

        }

        Appointment appointment = Appointment.builder()
                .patient(patient)
                .doctor(doctor)
                .appointmentDate(appointmentDate)
                .appointmentTime(appointmentTime)
                .reason(reason)
                .status(AppointmentStatus.REQUESTED)
                .build();

        return appointmentRepository.save(appointment);
    }

    public Appointment updateAppointmentStatus(Long id, AppointmentStatus status) {
        Appointment appointment = getAppointmentById(id);
        appointment.setStatus(status);
        return appointmentRepository.save(appointment);
    }

    public Appointment getAppointmentById(Long id) {
        return appointmentRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("Appointment not found with id: " +
id));
    }

    public List<Appointment> getAppointmentsByPatientId(Long patientId) {
        if (!patientService.existsById(patientId)) {
```

```
            throw new ResourceNotFoundException("Patient not found with id: " + patientId);
        }
        return appointmentRepository.findByPatientId(patientId);
    }

    public List<Appointment> getAppointmentsByDoctorId(Long doctorId) {
        if (!doctorService.existsById(doctorId)) {
            throw new ResourceNotFoundException("Doctor not found with id: " + doctorId);
        }
        return appointmentRepository.findByDoctorId(doctorId);}}
```

```
package com.examly.springapp.service;

import com.examly.springapp.exception.AppointmentConflictException;
import com.examly.springapp.exception.ResourceNotFoundException;
import com.examly.springapp.model.Appointment;
import com.examly.springapp.model.AppointmentStatus;
import com.examly.springapp.model.Doctor;
import com.examly.springapp.model.Patient;
import com.examly.springapp.repository.AppointmentRepository;
import  org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import  java.time.LocalDate;
import java.time.LocalTime;
import java.util.List;

@Service
public class AppointmentService {

    @Autowired
    private AppointmentRepository appointmentRepository;

    @Autowired
    private PatientService patientService;

    @Autowired
    private DoctorService doctorService;

    public Appointment bookAppointment(Long patientId, Long doctorId, LocalDate
appointmentDate,
            LocalTime appointmentTime, String reason) {

        Patient patient = patientService.getPatientById(patientId);
        Doctor doctor = doctorService.getDoctorById(doctorId);

        // Check for appointment conflicts
        if (appointmentRepository.existsByDoctorAndDateAndTime(doctor, appointmentDate,
appointmentTime)) {
            throw new AppointmentConflictException("The doctor already has an appointment at the
requested time");
        }

        Appointment appointment = Appointment.builder()
                .patient(patient)
                .doctor(doctor)
                .appointmentDate(appointmentDate)
```

```
            .appointmentTime(appointmentTime)
            .reason(reason)
            .status(AppointmentStatus.REQUESTED)
            .build();

        return appointmentRepository.save(appointment);
    }

    public Appointment updateAppointmentStatus(Long id, AppointmentStatus status) {

        Appointment appointment = getAppointmentById(id);
        appointment.setStatus(status);
        return appointmentRepository.save(appointment);
    }

    public Appointment getAppointmentById(Long id) {
        return appointmentRepository.findById(id)
            .orElseThrow(() -> new ResourceNotFoundException("Appointment not found with id: " +
id));
    }

    public List<Appointment> getAppointmentsByPatientId(Long patientId) {
        if (!patientService.existsById(patientId)) {
            throw new ResourceNotFoundException("Patient not found with id: " + patientId);
        }
        return appointmentRepository.findByPatientId(patientId);
    }

    public List<Appointment> getAppointmentsByDoctorId(Long doctorId) {
        if (!doctorService.existsById(doctorId)) {
            throw new ResourceNotFoundException("Doctor not found with id: " + doctorId);
        }
        return appointmentRepository.findByDoctorId(doctorId);}}
```

DoctorService.java

```
package com.examly.springapp.service;

import com.examly.springapp.exception.ResourceNotFoundException;
import com.examly.springapp.model.Doctor;
import com.examly.springapp.repository.DoctorRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;
@Service
public class DoctorService {

    @Autowired
    private DoctorRepository doctorRepository;

    public Doctor createDoctor(Doctor doctor) {
        return doctorRepository.save(doctor);  }

    public List<Doctor> getAllDoctors() {
        return doctorRepository.findAll();  }
```

```
    public Doctor getDoctorById(Long id) {
        return doctorRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("Doctor not found with id: "
+ id));}  public boolean existsById(Long id) {
    return doctorRepository.existsById(id);  }}
```

PatientService.java

```
package com.examly.springapp.service;

import com.examly.springapp.exception.ResourceNotFoundException;
import com.examly.springapp.model.Patient;
import com.examly.springapp.repository.PatientRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;
@Service
public class PatientService {

    @Autowired
    private PatientRepository patientRepository;

    public Patient createPatient(Patient patient) {
        return patientRepository.save(patient);}

    public List<Patient> getAllPatients() {
        return patientRepository.findAll(); }

    public Patient getPatientById(Long id) {
        return patientRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("Patient not found with id: "
+ id));  }

    public boolean existsById(Long id) {
        return patientRepository.existsById(id); }}
```

AppointmentRepository.java

```
 package com.examly.springapp.repository;

import com.examly.springapp.model.Appointment;
import com.examly.springapp.model.Doctor;
import com.examly.springapp.model.Patient;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;

import  java.time.LocalDate;
import java.time.LocalTime;
import java.util.List;
```

```
@Repository
public interface AppointmentRepository extends JpaRepository<Appointment, Long> {
   List<Appointment> findByPatient(Patient patient);

   List<Appointment> findByDoctor(Doctor doctor);

   @Query("SELECT COUNT(a) > 0 FROM Appointment a WHERE a.doctor = :doctor
AND a.appointmentDate = :date " +
         "AND a.appointmentTime = :time")
   boolean existsByDoctorAndDateAndTime(Doctor doctor, LocalDate date, LocalTime
time);

   List<Appointment> findByPatientId(Long patientId);
   List<Appointment> findByDoctorId(Long doctorId);}
```

DoctorRepository.java

```
package com.examly.springapp.repository;
import com.examly.springapp.model.Doctor;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface DoctorRepository extends JpaRepository<Doctor, Long> {
      boolean existsByEmail(String email);
      }
```

PatientRepository.java

```
package com.examly.springapp.repository;
import com.examly.springapp.model.Patient;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface PatientRepository extends JpaRepository<Patient, Long> {
      boolean existsByEmail(String email);
}
```

CHAPTER 6

CONCLUSION

This chapter presents the conclusion derived from the Healthcare Appointment Management System project and the learning outcomes from developing this solution.

6.1 CONCLUSION

In conclusion, the proposed Healthcare Appointment Management System is designed to simplify patient registration, appointment scheduling, and doctor-patient communication while ensuring secure data management. The system enhances efficiency by reducing waiting times, minimizing scheduling conflicts, and providing real-time updates to both patients and healthcare providers. With features like automated reminders, digital records, and role-based access, it ensures transparency, reliability, and patient safety. By streamlining administrative tasks, improving resource utilization, and offering actionable insights, this system supports better healthcare delivery, strengthens patient trust, and contributes to overall operational excellence.

6.2 FUTURE SCOPE

1. **Integration of AI for Appointment Optimization:**
   Using AI to predict patient appointment trends, optimize doctor allocation, and reduce waiting times.
2. **Enhanced Security Measures:**
   Implementing advanced authentication, encrypted health records, and secure communication between patients and healthcare providers.
3. **Mobile App Development Enhancements:**
   Expanding app functionality to include features like live chat with doctors, appointment reminders, digital prescriptions, and health history tracking.
4. **Integration with Payment Gateways and Insurance:**
   Adding seamless integration with multiple payment options and health insurance processing for convenience.
5. **Feedback and Rating Mechanism:**
   Allowing patients to provide feedback on doctors and services to continuously improve care quality.

CHAPTER 7

REFERENCES

GitHub – Healthcare Appointment System Examples

TutorialsPoint – Appointment Scheduling and Management Systems

Medium – Building a Healthcare Appointment Platform

Stack Overflow – Appointment System Development Discussions

YouTube – Healthcare Appointment System Development Tutorials

CodeProject – Appointment Booking Applications

GitHub – Medical Appointment App Source Codes

Upwork – Healthcare App Development Projects