DOCUMENTATION

OF

DOCTOR APPOINTMENT SYSTEM

BY

BHAVANI KUMAR KALAVAKUNTA (C314)

## 

## Table of Contents:

## 1.Introduction 4

## 1.1 Purpose

## 1.2 Scope of the Project

## 2.Requirement Analysis 5-6

## 2.1 Patient Registration

## 2.2 Doctor Registration

## 2.3 Appointment Scheduling

## 2.4 Appointment Management

2.5 Search

2.6 User Authentication

2.7 User Management

## 3.Non-Functional Requirement Analysis 7-8

## 3.1 performance

## 3.2 Security

## 3.3 Scalability

3.4 Reliability

## 4.System Design 9-10

## 4.1 ER Diagram

## 4.2 Use Case Diagram

## 4.3 Class Diagram

## 5.Implementation 11-16

## 6. Testing 17

## 7. Conclusion 18

## 8. References 19

**LIST OF FIGURES:**

fig 4.1 9

fig 4.2 9

fig 4.3 10

fig 4.4 10

fig 4.5 10

fig 4.6 10

**INTRODUCTION**

**1. Introduction**

The purpose of this document is to specify the requirements for the Doctor Appointment system, which will allow patients to schedule appointments with doctors online. The system will be developed using Java8, Spring Boot, Hibernate, and MySQL.

* 1. **Purpose**

The purpose of the Doctor Appointment system is to provide patients with an online platform to schedule appointments with doctors in a user-friendly and efficient manner. The system aims to simplify the process of booking appointments, reduce wait times, and increase patient satisfaction.

* 1. **Scope of the project**

The scope of the Doctor Appointment system project is to develop an online platform that will allow patients to schedule appointments with doctors. The system will be developed using Java8, Spring Boot, Hibernate, and MySQL.

The system will allow patients to create an account, search for doctors based on their specialization, location, and availability, and schedule appointments with them. Patients will be able to view their upcoming appointments, cancel them, and receive SMS notifications as reminders.

Doctors will be able to create an account, manage their appointments, and mark them as complete. Doctors will also be able to view their appointment schedule and cancel appointments as needed.

The system will support user authentication to ensure the security of patient and doctor information. The system must be scalable, reliable, and able to handle a large number of concurrent users.

Overall, the Doctor Appointment system will provide a user-friendly and efficient way for patients to schedule appointments with doctors, and for doctors to manage their appointments.

**CHAPTER -2**

**FUNCTIONAL REQUIREMENT ANALYSIS**

**2. Functional Requirements**

The Doctor Appointment system must provide the following functionality:

**2.1 Patient Registration**

Patients must be able to create an account with the system. The registration process should include the following fields:

* First Name: Patient should provide Patient’s First Name while registering.
* Last Name: Patient should provide Patient’s Last Name while registering.
* Email Address: Patient should provide Patient’s Email Address while registering.
* Password: Patient should provide Patient’s Password while registering.
* Phone Number: Patient should provide Patient’s phone number while registering.

**2.2 Doctor Registration**

Doctors must be able to create an account with the system. The registration process should include the following fields:

* First Name: Doctor should register with his First Name.
* Last Name: Doctor should register with his Last Name.
* Email Address: Doctor should register with his Email address.
* Password: Doctor should register with his Password.
* Phone Number: Doctor should register with his Phone number.
* Specialization: Doctor should register with his Specialization.

**2.3 Appointment Scheduling**

Patients must be able to schedule appointments with doctors. The scheduling process should include the following steps:

* Select a Doctor
* The patient can select a doctor based on the specialization and the disease/illness they are facing.
* Choose an Available Time Slot
* Patient can have the ability to choose the availability of time slot based on the selection of the doctor.
* Confirm Appointment
* After selection and choosing the available time slot the patient can book his appointment with confirm appointment.

**2.4 Appointment Management**

Admin must be able to manage their appointments. The management process should include the following functionality:

* View Appointments
* Admin can have the privilege for this Feature.
* Admin can view the appointments that are booked for the doctor.
* Mark Appointments as Complete
* Admin can mark the appointment as complete after the completion of the timeslot.

**2.5 Search**

Patients must be able to search for doctors by the following criteria:

* Specialization
* Patients can search the doctor’s specialization.
* Availability
* Patients can search the doctor’s availability to book the time slot.

**2.6 User Authentication**

The system must support user authentication to ensure the security of patient and doctor information.

* Patient Login
* Patient can login into the application with email and password which are given at the time of registration.
* Doctor Login
* Doctor can login into the application with email and password which are given at the time of registration.
* Admin Login
* Admin can login into the application with email and password which the hospital/Organization assigned or the registered.

**2.7 User Management**

Admin must be able to manage the users. The management process should include the following functionality.

* Remove Patients: Admin can remove Patients.
* Remove Doctors: Admin can remove Doctors.
* Add Doctors: Admin can ­­­­­­Add Doctors.
* Add/Update Specialization: Admin can Add/Update Doctors Specialization.
* Add Patients: Admin can add Patients.

**CHAPTER -3**

**NON-FUNCTIONAL REQUIREMENT ANALYSIS**

**3. Non-Functional Requirements**

The Doctor Appointment system must meet the following non-functional requirements:

**3.1 Performance**

The system must be able to handle a large number of concurrent users and provide a responsive user experience.

**3.2 Security**

The system must be secure and protect patient and doctor information from unauthorized access.

**3.3 Scalability**

The system must be scalable and able to accommodate future growth.

**3.4 Reliability**

The system must be reliable and provide high availability to users.

**Technical Requirements**

The Doctor Appointment system must be developed using the following technologies:

• Java8

• Spring Boot

• Hibernate

• MySQL

**Constraints**

The Doctor Appointment system must adhere to the following constraints:

* The system must be developed with compatibility of following constraints
* Hardware
  + Processor: Pentium 2.4 GHz or above.
  + Memory: 256 MB RAM or above.
  + Hard disk: At least 3GB or above [At least 3 GB free space is required]
* Software
  + Operating System: windows 7/8/10/11 , Linux/Ubuntu/Centos e.t.c.
  + Web Browser: Mozilla Firefox/ Google Chrome e.t.c with updated Versions.

**Assumptions and Dependencies**

The Doctor Appointment system is dependent on the following assumptions:

• Patients and doctors will have access to a device with an internet connection.

• Patients and doctors will be able to use the system without any additional training.

• The system will be integrated with an SMS notification service to remind patients of their appointments.

CHAPTER -4

**SYSTEM DESIGN**

ER Diagram:

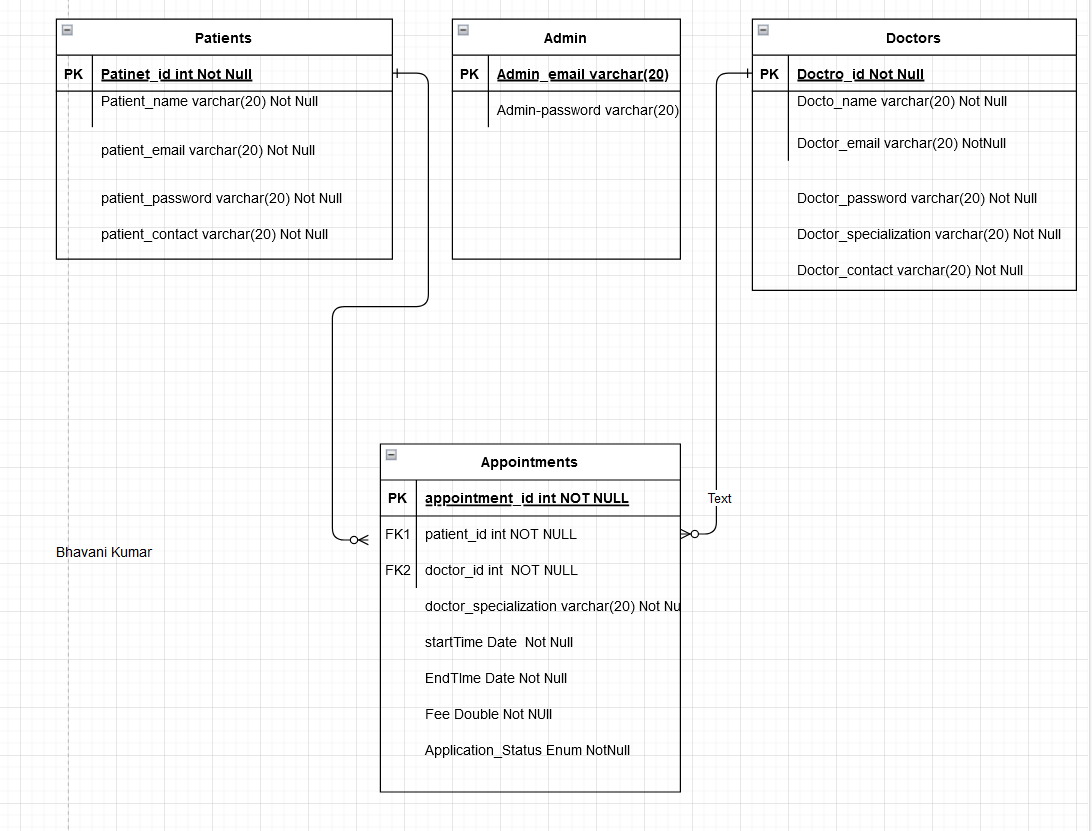


Fig 4.1

**USE CASE DIAGRAM:**

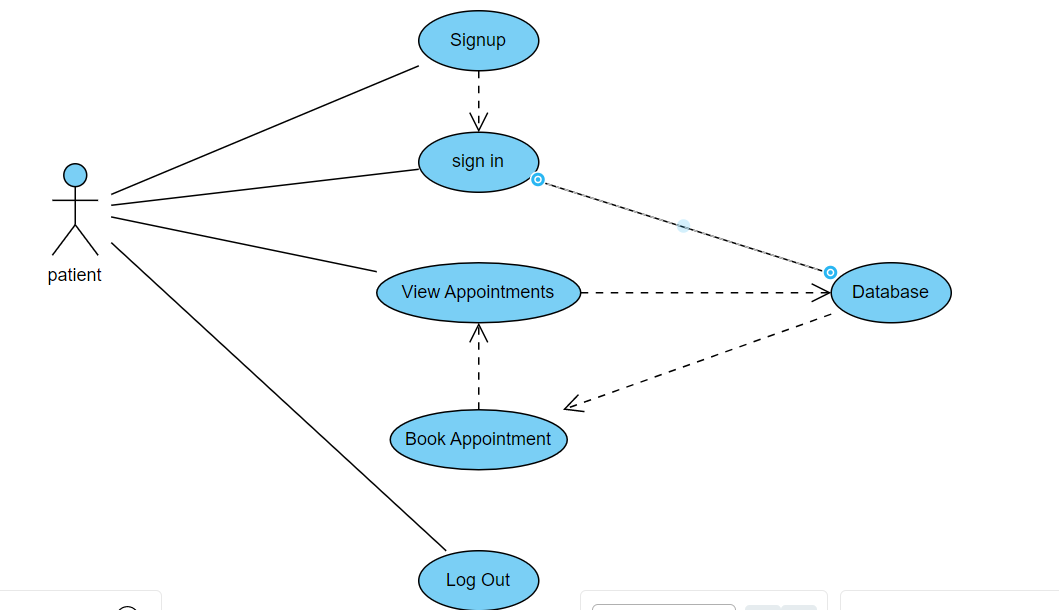


Fig 4.2

**CLASS DIAGRAM:**

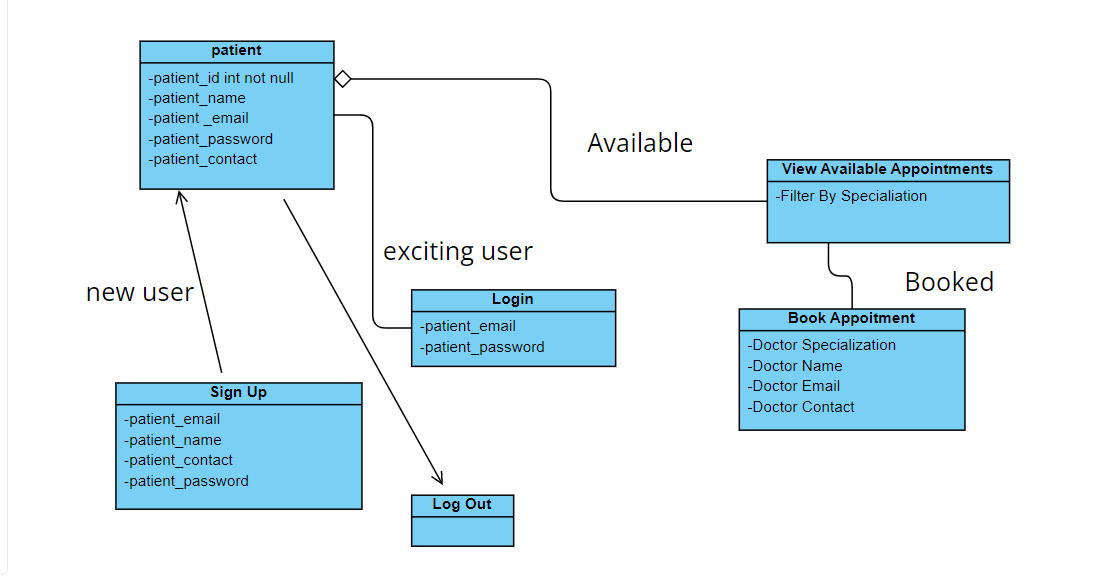


Fig 4.3

MODULES :

● Login: Login system for all types of roles (admin, doctor, patient).

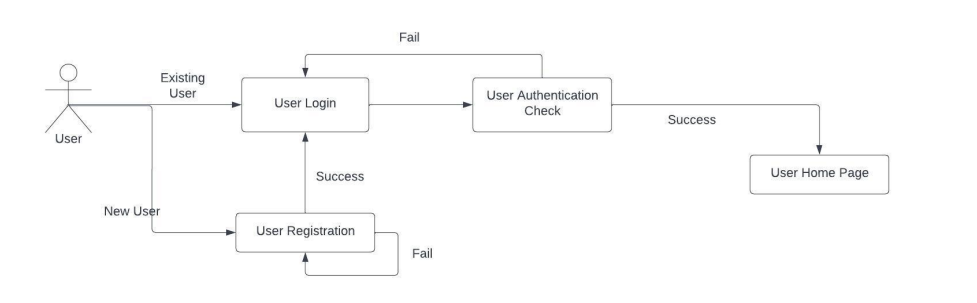


Fig 4.4

● Appointment booking: Users can book appointments for their required date and time.

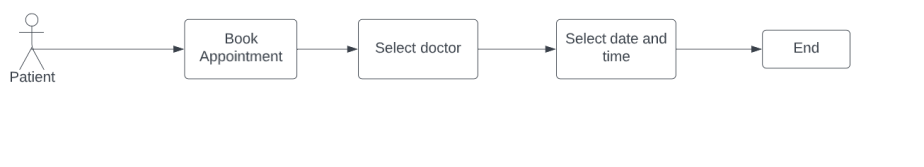


Fig 4.5

● Admin View Reports: Admin can view patient and doctor data based on certain filters.

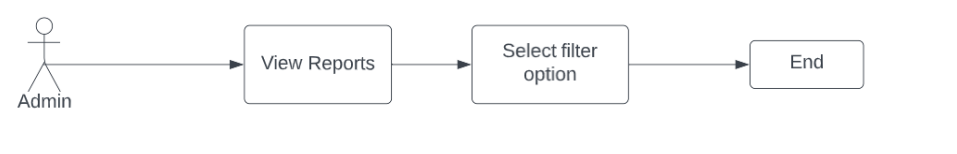


Fig 4.6

**CHAPTER -5**

**IMPLEMENTATION**

**Implementation of the project :**

**Entities :**

In Spring Boot, an entity usually refers to a Java class that represents a database table. The entity class typically has instance variables that correspond to the columns in the database table and methods that provide access to these variables.

Spring Boot uses the concept of Object Relational Mapping (ORM) frameworks such as **Hibernate or JPA** to map the entity class to the corresponding database table. By doing this, developers can interact with the database using objects rather than writing SQL queries directly.

To define an entity in Spring Boot, you typically use annotations such as **@Entity, @Table, @Id,** **@Column**, and so on. These annotations provide metadata about the class and its properties, which is used by the ORM framework to generate the necessary SQL statements for **CRUD (create, read, update, delete) operations.**

In addition to database entities, Spring Boot also supports other types of entities, such as RESTful web service entities, which are defined using annotations such as **@RestController** and **@RequestMapping**. These entities allow developers to expose APIs for their applications, which can be consumed by other systems or applications.

**1.Patient Entity**

@Entity

@Table(name="Patient")

**public** **class** Patient {

@Id

@GeneratedValue(strategy = GenerationType.***AUTO***)

**private** Integer patient\_id;

@Column(unique = **true**)

@NonNull

**private** String email;

**private** String password;

**private** String name;

**private** String contact;

@OneToMany(mappedBy = "patient")

**private** List<Appointment> appointments;

**public** Patient() {

**super**();

// **TODO** Auto-generated constructor stub

}

**public** Patient(Integer patient\_id, String email, String password, String name, String contact) {

**super**();

**this**.patient\_id = patient\_id;

**this**.email = email;

**this**.password = password;

**this**.name = name;

**this**.contact = contact;

}

**public** Integer getPatient\_id() {

**return** patient\_id;

}

**public** **void** setPatient\_id(Integer patient\_id) {

**this**.patient\_id = patient\_id;

}

**public** String getEmail() {

**return** email;

}

**public** **void** setEmail(String email) {

**this**.email = email;

}

**public** String getPassword() {

**return** password;

}

**public** **void** setPassword(String password) {

**this**.password = password;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getContact() {

**return** contact;

}

**public** **void** setContact(String contact) {

**this**.contact = contact;

}

@Override

**public** String toString() {

**return** "Patient [patient\_id=" + patient\_id + ", email=" + email + ", password=" + password + ", name=" + name

+ ", contact=" + contact + "]";

}

}

2. Doctor Entity

3. Admin Entity

4. Appointment Enitty

**Controllers:**

In Spring Boot, a controller is a Java class that handles incoming HTTP requests and returns an appropriate response. The controller acts as an intermediary between the view (the user interface) and the model (the data).

To define a controller in Spring Boot, you typically use the **@Controller** or **@RestController** annotation. The @Controller annotation is used to indicate that the class is a controller, while the @RestController annotation is used for controllers that are designed to return JSON or XML responses instead of HTML.

Controllers typically have methods that are annotated with request mapping annotations such as **@RequestMapping**, **@GetMapping**, **@PostMapping**, etc. These annotations define the URI path that the method will handle and the HTTP method that will be used for the request. Within the controller method, developers can use Spring's dependency injection to access other components such as services or repositories to perform business logic or retrieve data from a database. This separation of concerns allows for more modular and maintainable code. Once the controller has processed the request and generated a response, it sends the response back to the client using either a view template or directly returning a response object.

1.Doctor Controllers

1. Add Appointment
2. Registeration
3. Login

**2. Patient Controllers**

@RestController

@RequestMapping("/patient")

@CrossOrigin

**public** **class** PatientController {

@Autowired

**private** PatientService patientService;

**private** **final** Logger logger=LoggerFactory.*getLogger*(DoctorAppointmentApplication.**class**);

@PostMapping("/register")

**public** ResponseEntity<Patient> createPatient(@RequestBody Patient patient) {

Patient createdPatient = patientService.createPatient(patient);

logger.info("new Patient Registered");

**return** **new** ResponseEntity(createdPatient, HttpStatus.***CREATED***);

}

@PostMapping("/login")

**public** ResponseEntity<Object> login(@RequestBody PatientLoginRequest login) {

Patient patient = patientService.login(login.getEmail(), login.getPassword());

**if** (patient != **null**) {

logger.info("Patient Loggedin");

**return** ResponseEntity.*ok*(patient);

} **else** {

**return** ResponseEntity.*status*(HttpStatus.***UNAUTHORIZED***).body("Invalid email or password");

}

}

@GetMapping("/appointments/available")

**public** List<AppointmentAvailableResponse> getAppointments(@RequestParam(required=**false**, name="specialization") String specialization,

@RequestParam(required=**false**,name="startTime") @DateTimeFormat(iso = DateTimeFormat.ISO.***DATE\_TIME***) LocalDateTime startTime,

@RequestParam(required=**false**,name="endTime") @DateTimeFormat(iso = DateTimeFormat.ISO.***DATE\_TIME***) LocalDateTime endTime){

logger.info("Logged in patient checking available appointements");

**return** patientService.getAppointments(specialization, startTime, endTime);

}

@PostMapping("/appointments/available/book")

**public** PatientBookResponse bookAppointment(@RequestBody AppointmentRequest appointmentRequest) {

logger.info("Patient Booked Appointment");

System.***out***.println(appointmentRequest);

**return** patientService.bookAppointment(appointmentRequest);

}

}

* Login Patient
* Add Appointment
* Register Patient

3.Admin Controller:

* View All patients
* View All Doctors
* View All Appointments
* Login Admin

**Services:**

In Spring Boot, a service is a Java class that contains the business logic of an application. The service layer sits between the controller (which handles incoming requests) and the repository (which interacts with the database). Services typically represent a set of related operations and encapsulate the details of their implementation. They are responsible for processing and manipulating data, performing validations, and coordinating interactions between different components of the application.

To define a service in Spring Boot, you can use the **@Service** annotation. This annotation tells Spring that the class contains business logic and should be managed as a bean by the Spring container. Within the service class, developers can use dependency injection to access other components such as repositories or external services. This allows the service to delegate responsibilities to these components and focus on its own specific functionality.

Services also play an important role in maintaining separation of concerns in an application. By separating business logic into a separate layer, controllers become lighter and easier to read and maintain. Additionally, services can be reused across different controllers or even in different applications, making them a valuable component of any Spring Boot application.

1.Admin Service

* getAllDoctor()
* getSpecializations()
* getAllPatient()
* Login()
* getAllAppointment()

**2.Patient Service**

@Service

**public** **class** PatientService {

@Autowired

**private** PatientRepository patientRepository;

@Autowired

**private** AppointmentRepository appointmentRepository;

**public** Patient createPatient(Patient patient) {

Patient p = patientRepository.save(patient);

System.***out***.println(p);

**return** p;

}

**public** Patient login(String email, String password) {

Patient patient = patientRepository.findByEmail(email);

**if** (patient != **null** && patient.getPassword().equals(password)) {

**return** patient;

}

**return** **null**;

}

**public** List<AppointmentAvailableResponse> getAppointments(String specialization, LocalDateTime startTime, LocalDateTime endTime) {

List<Appointment> appointments = appointmentRepository.findAll();

**if**(specialization != **null**)

appointments = appointments.stream().filter(a -> a.getSpecialization().equalsIgnoreCase(specialization)).toList();

**if**(startTime != **null** )

appointments = appointments.stream().filter(a -> a.getStartTime().equals(startTime)).toList();

**return** appointments.stream().map(AppointmentAvailableResponse::**new**).toList();

}

**public** PatientBookResponse bookAppointment(AppointmentRequest appointmentRequest){

Appointment appointment = appointmentRepository.findById(appointmentRequest.getAppointment\_id()).get();

appointment.setStatus(AppointmentStatus.***BOOKED***);

appointment.setPatient(patientRepository.findById(appointmentRequest.getPatient\_id()).get());

Appointment app = appointmentRepository.save(appointment);

**return** **new** PatientBookResponse(app.getPatient(), app.getDoctor(), app) ;

}

}

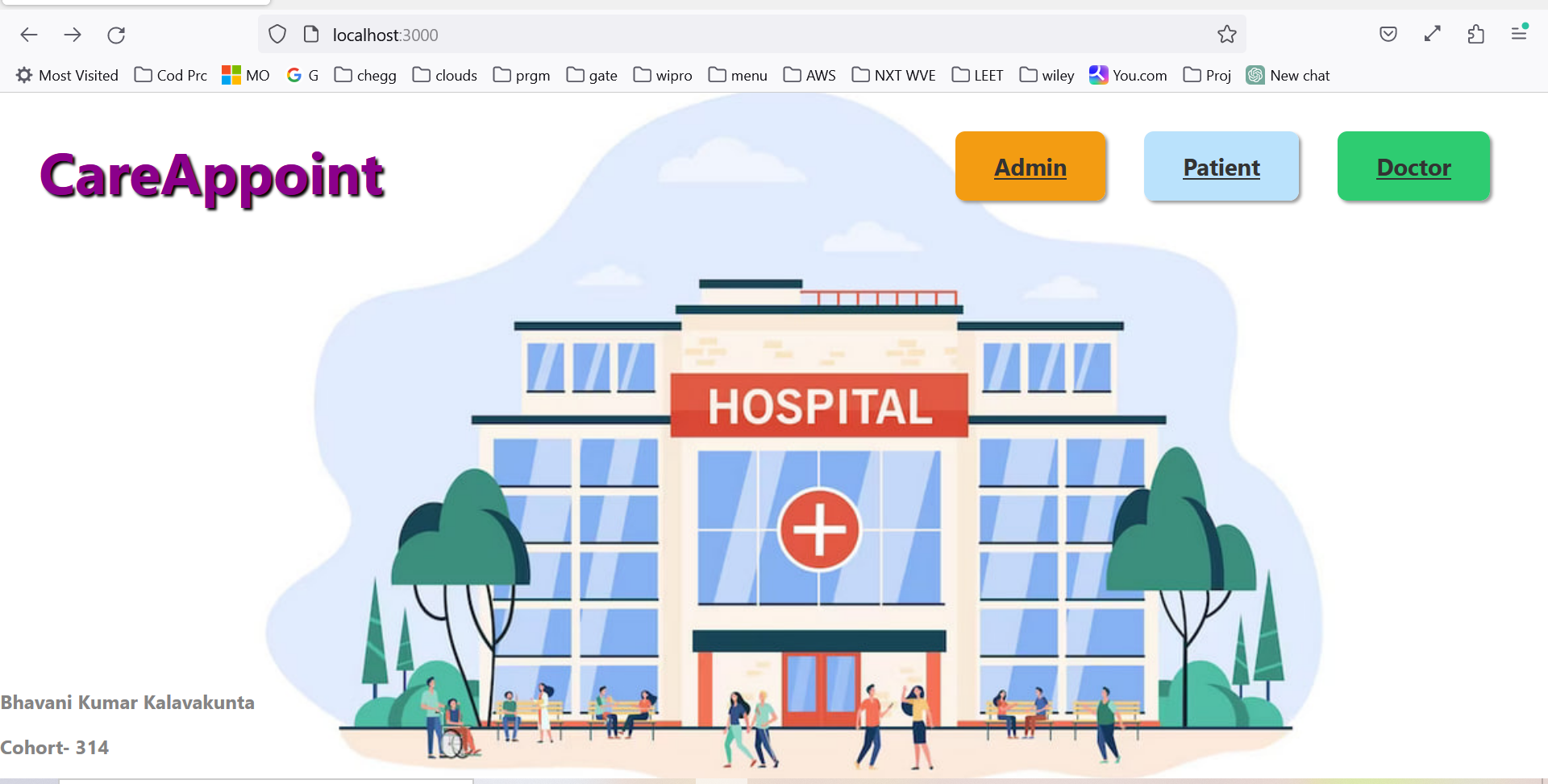
* LoginPatient()
* RegisterPatient()
* GetAvailable Appointments() By filtering specialization
* Book Available Appointments()

3.Dcotor Service

* Register Doctor()
* Login Doctor()
* Add Appointment()

**Front -END:**

**Home page:**

****

**CHAPTER -6**

**TESTING**

**1. Introduction to JUnit and Mockito**

JUnit is a popular open-source testing framework for Java applications that provides annotations and assertions to write unit tests. Mockito, on the other hand, is a mocking framework that allows developers to create mock objects to test interactions between components.

Mockito works well with JUnit tests because it allows us to easily create mock objects and execute test cases without having to worry about dependencies or external systems. By creating mock objects, we can isolate the code being tested and ensure that it behaves as expected in a controlled environment.

**2. Setting up a Spring Boot project for testing**

Before we start writing tests, we need to set up our Spring Boot project for testing. We can do this by adding the necessary dependencies to our `pom.xml` file:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>3.12.4</version>

<scope>test</scope>

</dependency>

The `spring-boot-starter-test` dependency provides all the necessary libraries for testing Spring Boot applications, including JUnit and Mockito. The `mockito-core` dependency provides the Mockito library.

**Test class:**

@SpringBootTest

**class** DoctorAppointmentApplicationTests {

@Mock

**private** AdminService adminService;

@InjectMocks

**private** AdminController adminController;

**private** List<PatientResponse> patients;

**private** List<DoctorResponse> doctors;

@Test

**void** contextLoads() {

}

**private** MockMvc mockMvc;

ObjectMapper objectMapper=**new** ObjectMapper();

ObjectWriter objectWriter=objectMapper.writer();

@Mock

**private** AdminRepository adminRepository;

**private** PatientRepository patientRepository;

Patient patient\_1=**new** Patient(1, "hello", "world", "how", "918522469");

Patient patient\_2=**new** Patient(2, "hell", "worl", "ho", "91852246");

@SuppressWarnings("deprecation")

@Before

**public** **void** setUp() {

MockitoAnnotations.~~initMocks~~(**this**);

**this**.mockMvc=MockMvcBuilders.*standaloneSetup*(adminController).build();

}

@BeforeEach

**public** **void** setup() {

patients = **new** ArrayList<>();

Patient patient1 = **new** Patient(1, "patient1@example.com", "password1", "John Doe", "1234567890");

Patient patient2 = **new** Patient(2, "patient2@example.com", "password2", "Jane Doe", "0987654321");

patients.addAll((Collection<? **extends** PatientResponse>) patient1);

patients.addAll((Collection<? **extends** PatientResponse>) patient2);

doctors = **new** ArrayList<>();

Doctor doctor1 = **new** Doctor(1, "doctor1@example.com", "password1", "Dr. John Smith", "1234567890", "Cardiology");

Doctor doctor2 = **new** Doctor(2, "doctor2@example.com", "password2", "Dr. Jane Smith", "0987654321", "Pediatrics");

doctors.addAll((Collection<? **extends** DoctorResponse>) doctor1);

doctors.addAll((Collection<? **extends** DoctorResponse>) doctor2);

}

@Test

**public** **void** testViewAllPatients() {

*when*(adminService.viewAllPatients()).thenReturn(patients);

ResponseEntity<List<PatientResponse>> responseEntity = (ResponseEntity<List<PatientResponse>>) adminController.viewAllPatients();

*assertEquals*(HttpStatus.***OK***, responseEntity.getStatusCode());

List<PatientResponse> patientResponses = responseEntity.getBody();

*assertEquals*(2, patientResponses.size());

PatientResponse patientResponse1 = patientResponses.get(0);

*assertEquals*("patient1@example.com", patientResponse1.getEmail());

*assertEquals*("John Doe", patientResponse1.getName());

*assertEquals*("1234567890", patientResponse1.getContact());

PatientResponse patientResponse2 = patientResponses.get(1);

*assertEquals*("patient2@example.com", patientResponse2.getEmail());

*assertEquals*("Jane Doe", patientResponse2.getName());

*assertEquals*("0987654321", patientResponse2.getContact());

}

@Test

**public** **void** getAllPatients\_sucess() **throws** Exception{

List<Patient> patients=**new** ArrayList<>();

patients.add(patient\_1);

patients.add(patient\_2);

Mockito.*when*(patientRepository.findAll()).thenReturn(patients);

mockMvc.perform(MockMvcRequestBuilders

.*get*("/admin/patients")

.contentType(MediaType.***APPLICATION\_JSON***))

.andExpect(*status*().isOk());

}

@Test

**public** **void** testViewAllDoctors() {

*when*(adminService.viewAllDoctors()).thenReturn(doctors);

List<DoctorResponse> responseEntity = adminController.viewAllDoctors(**null**);

*assertEquals*(HttpStatus.***OK***, ((ResponseEntity<List<DoctorResponse>>) responseEntity).getStatusCode());

List<DoctorResponse> doctorResponses = ((HttpEntity<List<DoctorResponse>>) responseEntity).getBody();

*assertEquals*(2, doctorResponses.size());

DoctorResponse doctorResponse1 = doctorResponses.get(0);

*assertEquals*("doctor1@example.com", doctorResponse1.getEmail());

*assertEquals*("Dr. John Smith", doctorResponse1.getName());

*assertEquals*("1234567890", doctorResponse1.getContact());

System.***out***.println("one test case passed");

DoctorResponse doctorResponse2 = doctorResponses.get(1);

*assertEquals*("doctor2@example.com", doctorResponse2.getEmail());

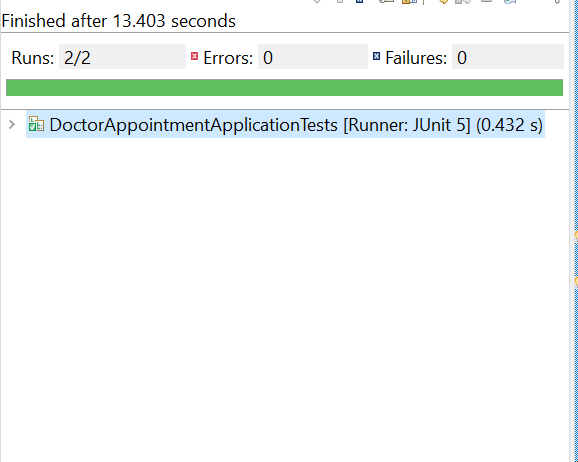
*assertEquals*("Dr. Jane Smith", doctorResponse2.getName());

*assertEquals*("0987654321", doctorResponse2.getContact());

}

}

**Result:**



**CHAPTER -7**

**CONCLUSION**

In conclusion, a doctor appointment system built using Spring Boot can provide an efficient way for patients to book appointments based on the availability of the doctor. The application can be designed to allow patients to view the availability of doctors and schedule appointments at their preferred time slots.

By leveraging the Spring Boot framework, developers can easily create RESTful APIs and microservices that can be integrated with other systems. This allows for seamless communication between different components of the application and improves the overall efficiency of the system.

Moreover, the application can also incorporate features such as patient profiles, appointment reminders, and feedback mechanisms to ensure a smooth and satisfactory experience for both patients and doctors. With proper implementation and testing, this system can reduce wait times, improve patient satisfaction, and streamline the process of scheduling appointments.

One of the main advantages of using Spring Boot for building this type of application is that it provides a framework for developing RESTful APIs and microservices. This allows developers to create modular components that can be easily integrated with other systems in the application. For instance, you might have separate microservices for the patient management system, doctor management system, and appointment scheduling system, which can be combined to form a complete application.

To start building the application, you would need to set up a backend database to store information about doctors, patients, and appointments. Spring Boot provides various libraries and tools to make this process easier, including Spring Data JPA and Hibernate.

Once the database is set up, you can create RESTful APIs that allow patients to view the availability of doctors and schedule appointments. For example, you might have an API that returns a list of available time slots for a particular doctor, and another API that allows patients to book an appointment at a specific time slot. These APIs can be secured using authentication and authorization mechanisms to ensure that only authorized users can access them.

Another important aspect of the doctor appointment system is user management. Patients should be able to create accounts, log in, and view their upcoming appointments. You might also want to implement features such as appointment reminders and cancellations.

In addition to patient-facing functionality, the application could also include features for doctors, such as calendar management and patient history tracking. Doctors could use the system to view their schedules, update their availability, and manage their patient lists.

Overall, a doctor appointment system built using Spring Boot can provide a powerful and flexible solution for managing patient appointments. By leveraging the strengths of Spring Boot, developers can create a scalable and reliable application that meets the needs of both patients and doctors.

**CHAPTER -8**

**REFERENCES**

* <https://github.com/Bhavanikumar1509/CareAppoint>
* <https://spring.io/>
* <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/>
* <https://legacy.reactjs.org/docs/getting-started.html>
* <https://react.dev/>
* <https://staruml.com/>