Smart Healthcare Appointment System

Project Overview

The **Smart Healthcare Appointment System** is a **Spring Boot 3 application** that helps a hospital manage **patients, doctors, appointments, prescriptions, and medical records**.

It integrates **Spring concepts** (Bean Lifecycle, Scopes, IoC, DI, AOP, Security, Testing) and uses **both relational databases (MySQL/Postgres with JPA/Hibernate)** and **NoSQL (MongoDB for prescriptions & records)**.

Requirements

Exceptions

Functional Requirements

1. **Authentication & Authorization**
    a. Users must log in (Basic Auth or JWT).
    b. Roles: **Admin**, **Doctor**, **Patient**.
    c. Role-based access:
        i. **Admin**: Manage doctors & patients.
        ii. **Doctor**: Manage appointments & prescriptions.
        iii. **Patient**: Book/cancel appointments, view records.
2. **Doctor Management**
    a. Admin can add/update/remove doctors.
    b. Patients can search doctors by specialty.
3. **Patient Management**
    a. Admin registers new patients.
    b. Patients can update their personal details.
4. **Appointment Management**
    a. Patients can book/cancel appointments.
    b. Prevent double-booking for the same doctor & time.
    c. Doctors can mark appointments as completed.
5. **Prescription & Medical Records**
    a. Doctors add prescriptions (stored in MongoDB).
    b. Patients can view their prescription history.
    c. Records may include notes, medicines, lab results.

6. **Logging**
    a. Use **Spring AOP** to log:
        i. Appointment booking/cancellation.
        ii. Prescription updates.
7. **Testing**
    a. Use **JUnit + Mockito** to test:
        i. Appointment booking logic (no double booking).
        ii. CRUD APIs for doctors & patients.

Non-Functional Requirements

- **Caching**: Use Hibernate first/second-level caching for frequently accessed doctor data.
- **Security**: Role-based authorization, secure endpoints.

Use Cases

Use Case 1: Patient Books Appointment

Actors: Patient, System
Steps:

1. Patient logs in.
2. Patient searches for doctor by specialty.
3. Patient selects available slot.
4. System checks availability (no double-booking).
5. Appointment is saved → log event via AOP.

Use Case 2: Doctor Adds Prescription

Actors: Doctor, System
Steps:

1. Doctor logs in.
2. Doctor opens today's appointments.
3. Selects patient → enters notes, medicines.
4. Prescription saved in **MongoDB**.

5. Patient can view prescription in portal.

Use Case 3: Admin Manages Doctors

Actors: Admin, System
Steps:

1. Admin logs in.
2. Admin adds doctor with name, specialty, availability.
3. Doctor record saved in relational DB.

Use Case 4: Patient Views Medical Records

Actors: Patient, System
Steps:

1. Patient logs in.
2. Navigates to “My Records”.
3. Records fetched from **MongoDB**.
4. Display prescriptions, lab reports.

Example Scenarios

Scenario 1: Double Booking Attempt

- Patient A books Dr. Smith at 10:00 AM.
- Patient B tries the same time slot.
- System rejects Patient B → "Time slot already taken".
- Logged via AOP: *"Double booking attempt prevented for Dr. Smith at 10:00 AM"*.

Scenario 2: Prescription Storage

- Doctor John adds prescription:
  - Patient ID: 101
  - Medicines: [Paracetamol, Vitamin D]
- Saved in MongoDB as document.

Scenario 3: Admin Adds Doctor

- Admin adds Dr. Emily (Cardiologist).
- Record saved in relational DB.
- Cached doctor list is refreshed.

Architecture

- **Frontend**: Use Postman for API testing.
- **Backend**: Spring Boot 3
- **Database Layer**:
    - MySQL/Postgres (Doctors, Patients, Appointments)
    - MongoDB (Prescriptions & Medical Records)
- **Security Layer**: Spring Security with role-based access.
- **AOP Layer**: Logging for booking & prescriptions.
- **Testing**: JUnit + Mockito.

Deliverables

1. **Working Spring Boot Project** (Maven-based).
2. **GitHub Repository** (public, with readme file). Include project description, setup instructions, screenshots (if any).
3. **Demo Presentation**