**Smart Appointment Booking System**

* **Problem statement**

In today’s world, booking appointments with professionals (doctors, salons, consultants, trainers, etc.) is often inconvenient for both service providers and customers.

**Service providers (clients)** struggle to manage bookings, availability, and reminders. Many rely on phone calls or manual registers, leading to double-booking, missed appointments, and poor customer experience.

**Customers (users)** face difficulty finding trusted service providers, checking their availability, and receiving reliable confirmations/reminders.

Existing systems are often single-business oriented (built for one clinic/salon) and lack flexibility for a platform where multiple clients can onboard their services and customers can book seamlessly.

Thus, there is a need for a **multi-tenant appointment booking platform** where:

1. Service providers can manage their profiles, services, availability, and appointments.
2. Customers can discover service providers by view all option or by their respective IDs, and book appointments with ease.
3. The system handles confirmations, reminders, and notifications asynchronously to improve reliability and scalability.

* **Objectives**

The primary objectives of the Smart Appointment Booking System are:

1. **For Clients (Service Providers)**
   1. Allow clients (doctors, salons, trainers, consultants, etc.) to register their business with a profile photo/logo.
   2. Can perform CRUD operation.
   3. Manage incoming appointments (accept, cancel, mark completed).
2. **For Users (Customers)**
   1. Can register themselves.
   2. Can view all services and decide which one to pick.
   3. Book, cancel, or reschedule appointments.
   4. Receive email confirmations and reminders for appointments.
3. **For the System (Technical Goals):**
4. Secure the APIs with JWT authentication and role-based access (ADMIN, PROVIDER, CUSTOMER).
5. Ensure reliable communication via asynchronous queues (RabbitMQ) for sending confirmations and reminders.
6. Provide structured API documentation using OpenAPI exposed vis Swagger UI for easy integration and testing.
7. Implement global exception handling and API versioning for maintainability.

* **Functional Requirements**

1. **User Management**
   1. Users can register, login, and update their profile (Customer, Provider, Admin).
   2. Role-based access control: Admin, Provider, Customer.
   3. Admin can manage all users (CRUD).
2. **Tenant Management**
   1. Admin or Provider can create tenant.
   2. Provider can create only 1 tenant.
   3. For 1 tenant, multiple services can be created.
   4. Provider can update their own linked tenant
   5. Customers do have the access to look all the available tenants.
3. **Service Management**
   1. Customer do have the access to look all the available services.
   2. All the CRUD operation can be performed by Admin or only that provider which belongs to the tenant for which service is marked.
4. **Appointment Management**
   1. Customers can book appointments.
   2. Appointment Service manages availability and avoids double booking.
   3. Users can view, reschedule, or cancel appointments.
   4. System sends email confirmations upon booking, and a reminder email is sent 4 hours before the appointment scheduled.
5. **Notifications**
   1. Appointment reminders are sent asynchronously (via message queue).
   2. Email notifications include service details, date/time, and staff assigned.
6. **API & Integration**
   1. APIs available for all services with Swagger UI documentation.
   2. API Gateway handles authentication, routing, and rate limiting.
   3. System integrates with email providers via Notification Service.
7. **Security & Authentication**
   1. Spring Security to enforce role-based authorization and authentication.
   2. Login throttling and API rate limiting to prevent abuse.

* **Non-Functional Requirements**

1. **Performance**
   1. API response time should be < 500ms for common requests.
   2. Search queries should return results within 1 second for typical datasets.
2. **Reliability & Availability**
   1. System uptime should be 99.9% (excluding maintenance).
   2. Retry mechanism for failed notifications and queue messages.
3. **Security**
   1. All API calls must use HTTPS.
   2. Passwords must be hashed (BCrypt or similar).
   3. JWT tokens for session management; expiration and refresh policies enforced.
4. **Maintainability**
   1. Swagger UI should document all APIs for easier maintenance.

* **High Level Architecture**

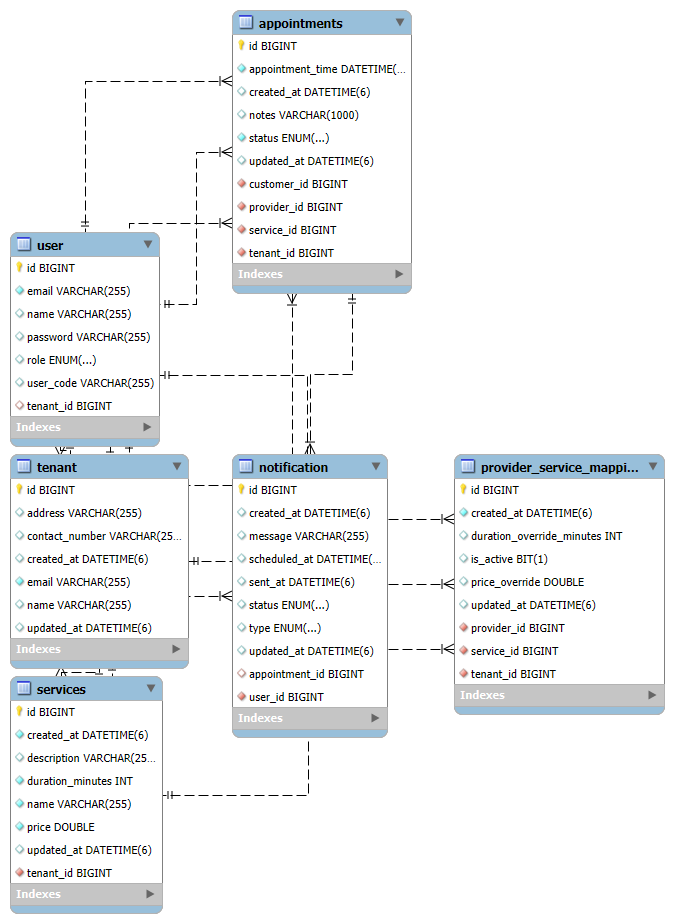
**A diagram of a software application

AI-generated content may be incorrect.**

* **Tech Stack**

1. **Backend**
   1. Spring Boot (Java 21) → Rapid development, production-ready features.
   2. Spring Security + JWT → Robust authentication & role-based access.
   3. Spring Data JPA (Hibernate) → ORM for database operations.
   4. RabbitMQ → Async queues for email notifications & reminders.
2. **Database**
   1. MySQL → Reliable relational DB for structured data (users, services, appointments).
3. **API Documentation**
   1. OpenAPI 3 (Swagger UI) → Interactive API docs.
4. **Deployment & Tools**
   1. Docker → Containerization & scaling.
   2. Git + Maven → Version control & build tool.

**ER Diagram**Generated from MySQL



**OpenAPI(docs)  
  
**

**Deployment Guide**

**1. Deployment Approach**

The project is packaged as a Spring Boot executable JAR and containerized with Docker.  
This approach is chosen because:

* Portability: Docker ensures the app runs the same on any environment.
* Isolation: Each service (app, MySQL, RabbitMQ) runs in its own container to avoid dependency conflicts.
* Simplicity: No need to install Java, Maven, or RabbitMQ manually on the host machine.
* **Key services used:**

| **Service** | **Purpose** | **Reason for Choice** |
| --- | --- | --- |
| Spring Boot App | Main application (APIs, business logic) | Lightweight, easy to containerize |
| MySQL | Relational database for users, appointments, services | Well-supported, reliable, easy integration |
| RabbitMQ | Messaging for async notifications and reminders | Reliable message broker, integrates with Spring AMQP |

**2. Deployment Steps (Docker-based)**

1. **Prepare the Environment**

* Ensure Docker Desktop or Docker Engine is installed.
* Clone the repository:

git clone <https://github.com/toxic-30/appointment-booking-system.git>  
cd SmartAppointmentBookingSystem

1. **Start Dependencies**

* Use Docker Compose to start MySQL and RabbitMQ containers:

docker-compose up -d

1. **Build & Run the App Container**

* The steps are –

docker build -t smartappointmentbookingsystem-image .

docker run -d --name smartappointmentbookingsystem-app \

--network=smartappointmentbookingsystem\_default \

-p 8085:8085 smartappointmentbookingsystem-image

1. **Verify Deployment**

* REST APIs: <http://localhost:8085>
* Swagger UI: <http://localhost:8085/swagger-ui/index.html>
* RabbitMQ Management: <http://localhost:15672> (user: guest, password: guest)

**3. Stopping / Cleaning Up**

* Steps are -   
  docker stop smartappointmentbookingsystem-app

docker rm smartappointmentbookingsystem-app

docker-compose down

**4. Notes / Considerations**

* The Docker-based deployment avoids cloud infrastructure for simplicity and local testing.
* If deploying to a cloud in the future, this same Docker setup can be used with services like AWS ECS, Azure Container Instances, or Kubernetes, providing easy scaling and isolation.
* Ports (8085 for the app, 3306 for MySQL, 5672/15672 for RabbitMQ) must be open on the host or cloud firewall.