**Title:**

Healthcare Appointment and Patient Management System with Microservices Architecture

**Objective:**

The goal of this project is to design and implement a scalable, efficient, and secure healthcare platform where patients can book appointments, doctors can manage their schedules, and administrative personnel can oversee the entire system. The platform will be built using a microservices architecture to ensure modularity, independence, and scalability. The system will use multiple databases (MySQL, MongoDB, and H2), event-driven communication via Apache Kafka, and will be containerized using Docker, with orchestration handled by Kubernetes.

**Project Requirements**

**1. Microservices Architecture:**

The application will be broken down into independent microservices, each handling a specific aspect of the system. This modular approach allows easy scaling, updating, and maintenance of each component without affecting others. The following microservices will be implemented:

* **Eureka Server** :

This project help use to create Eureka server running on port number 8761. Which server project keep the track about micro services.

* **Patient Service: 8181 port number** 
  + **Description**: Manages patient registration, updates to patient details, and retrieval of patient information.
  + **Database**: MySQL (for structured, relational data).
  + **Endpoints**:
    - POST /patients: Register a new patient.
    - GET /patients/{id}: Retrieve a patient by ID.
    - PUT /patients/{id}: Update a patient’s information.
* **Doctor Service: 8282** 
  + **Description**: Manages doctor profiles and schedules, ensuring doctors are available for appointments.
  + **Database**: H2 (an in-memory database for fast access to real-time doctor schedules).
  + **Endpoints**:
    - POST /doctors: Register a new doctor.
    - GET /doctors/{id}: Retrieve a doctor’s profile.
    - GET /doctors/schedules: Retrieve all doctors’ availability.
* **Appointment Service: 8383** 
  + **Description**: Manages appointment scheduling, updating, and cancellation for patients and doctors.
  + **Database**: MongoDB (for flexible and scalable appointment management).
  + **Endpoints**:
    - POST /appointments: Schedule a new appointment.
    - GET /appointments/{id}: Retrieve appointment details.
    - PUT /appointments/{id}: Update or cancel an appointment.
* **Billing Service:**
  + **Description**: Processes billing for appointments and generates invoices.
  + **Database**: Could use a relational DB (MySQL) or NoSQL DB (MongoDB), depending on billing requirements.
  + **Endpoints**:
    - POST /billing: Generate a new bill after an appointment.
    - GET /billing/{id}: Retrieve bill details.

**2. Database Integration:**

* **MySQL**: Used for the **Patient Service** to manage structured, relational data (e.g., personal details, medical history, etc.). The strong relational model helps in maintaining data integrity and running complex queries.
* **MongoDB**: Used for the **Appointment Service** to store appointment information, which may involve complex data types (e.g., patient info, appointment status, etc.). MongoDB’s schema flexibility allows for scaling as more types of data (like notifications, reminders) are added in the future.
* **H2**: An in-memory database used for the **Doctor Service** to store real-time data on doctor schedules and availability. This allows fast access to frequently changing data, such as appointment slots and schedules.

**3. Message-Driven Communication with Apache Kafka:**

For asynchronous communication between microservices, **Apache Kafka** will be used to publish and consume real-time events. The following critical events will be handled via Kafka:

* **Appointment Created Event**: Published by the **Appointment Service** and consumed by the **Doctor Service** to notify doctors of new bookings and update their availability.
* **Appointment Cancelled Event**: Triggered when an appointment is cancelled and broadcast to both the **Doctor Service** (to free up schedule slots) and the **Billing Service** (to void or refund charges).
* **Payment Completed Event**: Sent from the **Billing Service** to the **Patient Service** to notify patients of successful payments.
* **Doctor Availability Event**: The **Doctor Service** will notify the **Appointment Service** when there’s a change in a doctor’s availability.

These Kafka events will allow microservices to communicate in real-time, keeping the system responsive without direct dependency between services.

**4. Containerization and Deployment with Docker and Kubernetes:**

To ensure the system is portable and scalable, each microservice will be packaged into a **Docker container**. This ensures a consistent environment across development, testing, and production.

* **Docker**: Each service (Patient, Appointment, Doctor, Billing) will be deployed as a Docker container. This containerized setup will streamline the deployment process and simplify testing and integration.
* **Docker – compose or Kubernetes**: Kubernetes will be used to orchestrate the deployment of these containers, providing:
  + **Scaling**: Automatically scale services up or down based on demand (e.g., during peak hours, the Appointment Service might need more instances).
  + **Load Balancing**: Distribute incoming requests evenly across service instances.
  + **Self-Healing**: Kubernetes can automatically restart failed services or containers, ensuring higher availability and reliability.

**5. API Gateway:**

An **API Gateway** (e.g., Spring Cloud Gateway or Zuul) will be used to route requests from clients (patients, doctors, administrators) to the appropriate microservice. It will also provide security features such as rate-limiting and request logging.

* The gateway will handle all external requests and forward them to the relevant microservice, ensuring that each service can focus purely on business logic.

**6. Security and Authentication:**

The system will employ **OAuth2/JWT** for secure authentication and authorization:

* **Patient Login**: Patients will authenticate using username/password and will be issued a **JWT token** that must be presented in subsequent requests.
* **Doctor Login**: Similarly, doctors will authenticate and manage their schedules securely using their token.
* **Role-Based Access**: Different roles (patient, doctor, admin) will have access to different endpoints and data, ensuring privacy and security.

**7. Challenges Addressed:**

* **Scalability**: The microservices architecture ensures that the system can scale as demand increases, with individual services scaled independently based on load.
* **Fault Isolation**: Failure in one microservice (e.g., the Billing Service) won’t bring down other services (e.g., Patient or Appointment Service), ensuring higher system availability.
* **Real-Time Communication**: Kafka’s event-driven architecture will keep the system highly responsive, ensuring updates (e.g., appointment status, billing info) propagate instantly across services.
* **Portability and Ease of Deployment**: Docker containers ensure consistency across environments, while Kubernetes automates scaling and self-healing, reducing operational overhead.
* **Security**: Role-based authentication using OAuth2/JWT ensures that sensitive data is securely accessed only by authorized users.

**Note :** You need to create all required end point and test these endpoint using Post man client and take the screen short. These end point validated using JWT token and allow to access respective User.