### Team Group Giga chad

### Synopsis

The project objective is to develop a scalable and user-friendly hotel booking system that enables customers to book rooms across multiple hotels and it supports these hotel branches in managing room availability, bookings, and cancellations.   
The system uses a microservices architecture to ensure it remains modular, scalable, and well-synchronized across distributed systems.  
  
Built with Spring Boot and REST APIs, the project’s main features include checking room availability, handling bookings, and processing cancellations. A centralized broker service oversees these operations to ensure smooth functionality.  
  
The project is designed for the hospitality sector, focusing on hotel reservations and management.

### Technology Stack

The system leverages a robust set of technologies to ensure scalability, reliability, and efficiency in managing hotel bookings across distributed branches:

* **REST APIs**: These facilitate seamless communication between system components, such as the API Gateway Broker, Hotel MicroServices, and Database. Their simplicity, scalability, and platform-independent nature make them ideal for connecting microservices.
* **Java (version 8/11)**: This is the primary programming language for building microservices. It has been chosen for its robustness, scalability, and extensive ecosystem.
* **Spring Boot**: Simplifies the development and deployment of microservices with its modular and lightweight framework.
* **Maven**: Manages project dependencies and builds the application efficiently.
* **MySQL**: Provides reliable, scalable data storage with easy integration into our architecture.
* **DBeaver and MySQL Workbench**: These database management tools are used for designing, querying, and maintaining the relational database structure, making data management efficient and user-friendly.
* **Relational Database Management System (RDBMS)**: The system relies on an RDBMS approach to ensure data integrity, consistency, and robust query capabilities, critical for managing complex hotel booking operations.

This combination of technologies ensures a scalable, maintainable, and efficient hotel booking system capable of handling distributed operations.

### System Overview

#### Main Components:

1. **Central API Gateway (Broker)**: Manages incoming requests and routes them to the appropriate hotel branch or service.
2. **Hotel Microservices (Distributed Nodes)**: Each hotel branch operates as an independent service with its database, managing local room availability, bookings, and cancellations.
3. **Database Systems**: Each hotel branch maintains a local database for managing its data. A central database could aggregate important data for reporting, analytics, and management purposes.

#### Endpoints

The application provides the following endpoints for accessing various services:

**Quotation Services**:

* + [http://localhost:8080/quotations](http://localhost:8080/quotations" \t "_new)
  + [http://localhost:8081/quotations](http://localhost:8081/quotations" \t "_new)
  + [http://localhost:8082/quotations](http://localhost:8082/quotations" \t "_new)

**Broker Service**:

* + [http://localhost:8083/applications](http://localhost:8083/applications" \t "_new)

#### System Architecture:

A diagram of a business

Description automatically generated

#### How does the system work?

The lifecycle of a client request in the hotel reservation system begins when the Client searches for available hotels on specific dates. The request is sent to the Broker (API Gateway) via the POST /applications endpoint at 8083/applications, which routes it to the appropriate service. To check hotel availability, the Broker forwards the request to the CheckingService via the POST /quotations endpoint at 8080/quotations. This service queries the Database (DB) for available hotels based on the client’s requested dates and preferences and returns a list to the client through the Broker. After reviewing the options, the Client sends a booking request to the BookingService via the POST /quotations endpoint at 8082/quotations. The BookingService checks the DB for availability and confirms the booking, sending the confirmation back to the Client. Suppose the Client decides to cancel the reservation later. In that case, they can send a cancellation request (including the hotel name and booking dates) to the CancelService using the POST /quotations endpoint at 8081/quotations. The Broker routes this request to the CancelService, which processes the cancellation by updating the DB and confirming the cancellation with the Client.

Throughout this process, the Broker ensures each request reaches the correct service, allowing the services to remain independent. The independence of each service enhances the system’s flexibility and resilience, enabling individual components to be updated, tested, or scaled independently without impacting the others. This separation also improves failure isolation—if one service fails, it doesn't affect the entire system, ensuring more reliable operation and easier maintenance. This architecture simplifies the request lifecycle and improves maintainability, as each service is modular and can be managed with minimal disruption to the whole system. By maintaining the independence of each service, the system can continue functioning smoothly even as components evolve or encounter issues, ensuring that clients experience minimal disruption.

#### Scalability and Fault Tolerance:

Our system is designed with the following features to ensure scalability and fault tolerance:

1. **API Gateway**: We use an API Gateway to dynamically route traffic and manage distributed service scaling, ensuring efficient request handling across the system.
2. **Independent Hotel Branch Microservices**: Each hotel branch is deployed as an independent microservice, allowing it to operate autonomously. This design ensures that the failure of one branch does not impact the functionality of others.
3. **Distributed Databases**: We employ distributed databases to store booking data, enabling consistent data access and replication across nodes. This approach ensures reliability and data integrity even in the case of individual node failures.
4. **Dockerization**: Each microservice is containerized using Docker, enabling easy deployment, scalability, and isolation. This ensures consistent environments across development, testing, and production, while also facilitating rapid scaling of individual services to handle increased traffic.

These strategies collectively enhance the system’s ability to scale with demand and recover gracefully from potential failures.

### Contributions

* **Muhammad and Giacomo**: Responsible for developing the microservice architecture and broker functionalities, as well as implementation and testing.
* **Sagar**: Responsible for database design, implementation, and dockerization resolution.

### Reflections

**Key Challenges and Solutions**:

Synchronizing bookings across distributed branches was a significant challenge, which we addressed by implementing distributed databases and asynchronous message queues. We ensured service reliability through redundancy mechanisms, health checks, and load balancers and streamlined inter-service communication using an API Gateway.

**Potential Improvements**:

We plan to implement advanced caching mechanisms to reduce database load, integrate AI-driven demand prediction for optimizing room availability, and enhance the client interface to improve user experience.

**Learnings**:

We found that REST APIs facilitated smooth communication but required careful design to avoid bottlenecks. Distributed databases improved scalability but underscored the need for robust replication strategies. Spring Boot significantly accelerated our development process while highlighting the importance of efficient modularity. Additionally, we learned that monitoring tools are essential for maintaining system performance and reliability.