1. Title Page

Title: POC for Basic Microservice Architecture

Name: Abhishek Anand

Date: 02-April-2025

Project Name: Microservice Hotel Management

2. Introduction

Overview:

This project is a baseline implementation for a hotel management system using microservice architecture. Here, I am trying to implement a hotel booking system using Agentic AI in future iterations, but in the first iteration, I am just familiarizing myself with microservice architecture by building hotel and rating services.

Objective:

Learning the base microservice architecture and implementation of Agentic AI using Spring AI and MCP

3. Technologies Used

I. Spring Web
II. Spring Reactive
III. Spring Cloud
IV. PostgreSQL
V. MongoDB
VI. Docker
VII. Eureka Server
VIII. OpenFeign
IX. Resilience4j

4. Architecture Overview

Diagram:

I am trying to create multiple services named: hotel-service, rating-service, api-gateway, config-service, service-registry, and user-service. They will manage data for their respective domains as their names suggest. Each service will communicate with the required services to retrieve needed details.

Explanation:

Each service will have its own implementation:

I. **Hotel-service**: Stores hotel data like hotel name, location, and description.
II. **Rating-service**: Stores feedback/ratings, along with corresponding `user-id` and `hotel-id`.
III. **User-service**: Manages user details such as name, email, and role.
IV. **Service-registry**: This is a Eureka Server. It keeps track of all running services. Apart from registration, it enables service discovery, load balancing, and health checks, which support smooth scalability.
V. **API-Gateway**: This service acts as the single entry point for all requests. Since we are breaking the monolithic architecture into multiple services, each would otherwise expose its own endpoints, causing a poor user experience. The API Gateway routes requests to the appropriate services. We only need to hit the DNS mapped to the API Gateway.
VI. **Config-Server**: This service allows storing configuration files in a centralized external repository. This makes configuration management easier and more maintainable.

5. Implementation Details

Step-by-Step Process:

We need to have a class diagram and flow structure of the project to start with. In my iteration, I am focusing on achieving a microservice architecture. In the second iteration, I will implement GraphQL and Agentic AI.

Create each base service from the class diagram. In this version, I have created a hotel service, a rating service related to them, and a user service to log which user is searching for what.

Once the base is created, we need to establish communication between them for data transfer. There are a few ways to achieve this:

I. **RestTemplate** (deprecated but still in use)
II. **RestClient** (new and preferred for modern Spring apps)
III. **WebClient** (non-blocking, reactive)
IV. **OpenFeign** (preferred for simplicity)

In my case, I chose **OpenFeign** for most of the services and used **RestClient** once for learning purposes. If you use option 1 or 2, you must handle load-balancing using either @LoadBalanced while creating the @Bean, or by using DiscoveryClient and ServiceInstance in the service implementation class.

Service Registry:

To enable smooth communication and allow services to load-balance, we need a centralized location where services can discover each other.

Here, the **service-registry / Eureka-server / discovery-server** comes into the picture.

It registers all the services using either their IP addresses and port numbers or their instance names (best practice).

By setting up proper configuration (e.g., eureka.client.service-url.defaultZone=http://localhost:8761/eureka), each service registers itself and can be addressed properly.

API-Gateway:

With many services, calling individual ports/IPs can be confusing and inconsistent.

To solve this, we use an **API-Gateway**.

It routes incoming requests to the correct service based on path matching.

We can also apply custom filters, modify headers, and implement security (authentication/authorization) at this layer.

Config Server:

To avoid repeating the same configurations across multiple services, we use a **Config Server**.

This helps us store configurations externally (e.g., GitHub) and makes the system loosely coupled.

Upon restarting, services fetch updated configuration automatically.

Resilience and Fault Tolerance:

As services are dependent on each other, we need to manage failures and resources carefully.

For this, we use **Resilience4j** modules like:

- **Circuit-Breaker**:

  Prevents cascading failures.

  After a certain number of failures, it stops calling the service and moves to **open state**.

  In the **open state**, a fallback method responds.

  Then it goes to **half-open state** and checks if the issue is resolved before going back to **closed state**.

- **Rate-Limiter**:

  Prevents server overload, better for managing db operations especially write operations.

  Controls the number of requests allowed within a specific time frame.

  If exceeded, it can either queue, reject, or return fallback.

- **Retry**:

  Automatically retries failed requests up to a configured number of times.

  If still failing, it calls the fallback method.

Challenges:

✓ **Problems Overcome:**

I. Creating data flow between services.
II. Implementing basic security.

✧ **Problems Still Facing:**

I. Docker Compose implementation for microservices (structure and dependencies).
II. Circuit-breaker and retry working together.
III. Circuit-breaker blocking fallback exception handling properly.

6. SDLC Model

✓ Agile and Iterative

This project is following Agile and Iterative model. As I am building this project in steps, I will be adding features in iterations, each iteration focusing on one part of the system. For example, first iteration was about basic microservice understanding, next will be about GraphQL and Agentic AI integration.
Each feature is planned, implemented, tested, and improved in small sprints. This gives better control, flexibility and learning in each step.

7. Results

✓ What was achieved:

I. Understood and implemented base microservice structure.
II. Successfully created communication between services using OpenFeign.
III. Implemented API Gateway and Eureka server.
IV. Added Resilience4J features like Circuit Breaker and Rate Limiter.
V. Managed centralized configurations using Config Server with GitHub.
VI. Built understanding of distributed architecture and its working.

✧ Limitations:

I. Docker Compose integration with microservices is not fully working.
II. Circuit Breaker and Retry are clashing when used together.
III. Exception handling in fallback methods of Circuit Breaker is not consistent.
IV. UI and GraphQL part is not yet integrated, planned for next iteration.

8. Conclusion

This project gave me a good hands-on understanding of how microservices work in real-time applications. I was able to create multiple independent services and connect them properly using service registry and gateway. Also explored how to build fault-tolerant services using resilience patterns.

It is still in progress and will be extended in future iterations.

In future iteration, I will try to imply what is the industry standard and is there any specific reason for that approach.

9. Future Work

I. Add GraphQL layer for better querying and fetching.
II. Integrate Agentic AI using Spring AI and MCP.
III. Solve the Docker Compose and containerization issues.
IV. Make services production ready with full security.
V. Add proper UI and make complete working prototype.
VI. Optimize communication and performance further.

10. References

I. Micro-service intro : https://www.baeldung.com/cs/microservices
II. Microservice: https://spring.io/blog/2015/07/14/microservices-with-spring
III. Eureka Server and client: https://docs.spring.io/spring-cloud-netflix/docs/current/reference/html/#service-discovery-eureka-clients
IV. Cloud-Config : https://docs.spring.io/spring-cloud-config/reference/index.html
V. API-GATEWAY: https://www.redhat.com/en/topics/api/what-does-an-api-gateway-do
VI. GATEWAY-Configuration : https://spring.io/projects/spring-cloud-gateway
VII. Spring cloud: https://spring.io/projects/spring-cloud