

Hotel Booking Application

Distributed Database Management System - CS465



National Institute of Technology Karnataka Surathkal

Date: 28 March, 2024

Submitted To:

Prof. P . Santhi Thilagam

CSE dept. NITK

Group Members:

G. JAYA RAGHA CHARAN - 201CS117

V. SAI SWAROOP - 201CS268

AKSHITHA BADAVATH - 201CS213

ABHINAV SAI PAMPATI- 201CS137

BK ESWAR TEJA - 201CS211

ABSTRACT

This project aims to revolutionize hotel management systems by implementing a modern and scalable solution utilizing microservices architecture, Spring Boot, Java framework, and MongoDB database technology. Traditional hotel management systems often need help with scalability, flexibility, and maintenance, leading to inefficiencies and limitations in managing various hotel operations. By adopting microservices architecture, our system seeks to address these challenges by breaking down the application into independent, modular services, each responsible for a specific aspect of hotel management.

INTRODUCTION

Our system utilizes microservices architecture. Spring Cloud is a crucial component for building reliable and robust microservices. It provides pre-implemented design patterns like service discovery and configuration management. This covers an online application with various architectural patterns like service discovery, centralized configuration, distributed tracing, and event-driven architecture. Different services such as Booking service, Hotel Booking service, inventory service, and notification service are developed to interact synchronously and asynchronously, utilizing message queues like Kafka. This delves into the surrounding services like API Gateway, authentication, and tools for distributed tracing and centralized logging. Zipkin is used for distributed tracing, and Spring Data is used for data storage. The services will follow a logical architecture pattern, with controllers, service layers, repositories, and database interaction. Each microservice possesses its API, enabling independent development, testing, and deployment. This enhances our system's resilience and scalability, as we can update and scale each microservice individually. Furthermore, the flexibility of microservices permits us to utilize diverse technologies for each component, optimizing functionality. Overall, microservices have played a pivotal role in crafting our system, facilitating the creation of a modern, efficient, and scalable solution.

OVERVIEW OF SYSTEM

Hotel Management System aims to revolutionize hospitality services by offering a comprehensive web application tailored to streamline and optimize hotel operations. Employing a microservices architecture, each service will cater to specific functionalities essential for efficient hotel management.

The core services are

- Hotel Service
- Bookings Service
- Inventory Service
- Notification Service
- Discovery Server
- API Gateway

Employing a microservices framework, our system comprises standalone components that interact with each other. These microservices are containerized, enhancing scalability and resilience. The system uses asynchronous and synchronous communication.

PROCESS FLOW

Firstly, let's examine the services we're developing. We have the hotel service, utilizing MongoDB, the booking service connected to a MySQL database, and the inventory service, which also interacts with MySQL to manage inventory information. Additionally, a stateless component, the notification service, sends notifications to users without a database.

In a microservice architecture, services must communicate with each other. For instance, the bookings service communicates with the inventory service to synchronize inventory availability when a booking is placed. An asynchronous communication approach is adopted to notify users of successful bookings, utilizing message queues such as Kafka. Externally, supporting services like the API Gateway facilitate interaction between users and various services, serving as an entry point for user requests.

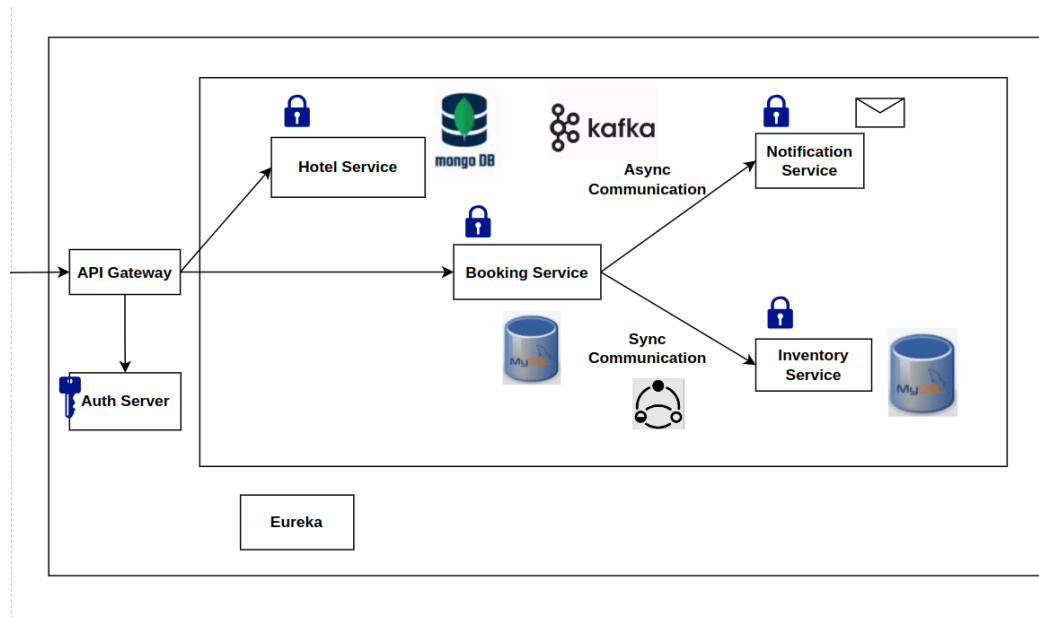


Fig1: Microservice Architecture

TECHNOLOGIES USED:

The technologies used in developing the hotel management system are :

- **MongoDB:** MongoDB acts as the primary database technology for our Hotel Management System, offering a versatile and scalable solution for data management. Utilizing a NoSQL document-oriented database such as MongoDB presents numerous benefits tailored to our needs, including flexible schema handling, scalability, and high performance.
- **SpringCloud:** Spring Cloud is crucial for building reliable and robust microservices. It provides pre-implemented design patterns like service discovery and configuration management
- **Kafka:** Kafka plays a pivotal role as a messaging middleware technology in our microservice-driven hotel management system, ensuring dependable communication among distributed microservices. Kafka's importance lies in its utilization of the Message Queuing Protocol, asynchronous communication support, and its message delivery reliability.

- **Postman:** Postman plays a critical role in our development workflow, offering an extensive platform for testing and troubleshooting our microservice-driven hotel management system. Postman's functionalities include API testing and validation, facilitating collaboration and team workflows, and streamlining automated testing processes.
- **Zipkin:** Zipkin is an open-source distributed tracing system designed to help developers troubleshoot latency issues in service-oriented architectures (SOA) and microservices-based applications.
- **KeyCloak:** It provides features for authentication and authorization to secure their applications and APIs. Keycloak supports various authentication mechanisms
- **Docker:** Docker provides tools and workflows to streamline the containerization process, making it easier to develop, deploy, and manage applications across different environments.
- **MySQL:** MySQL is used to store and manage structured data. Booking service and inventory services are used in this project using MySQL.
- **Eureka:** It is designed to facilitate the deployment and management of microservices in a cloud-based environment. Eureka allows services to register themselves with the registry upon startup and provides a mechanism for other services to locate and communicate with them dynamically. Inventory service, handling service discovery using Netflix Eureka, and creating a discovery server module using Spring Cloud Netflix Eureka server.

These technologies offer a resilient and scalable foundation for building microservices-oriented applications equipped with features for monitoring, testing, and facilitating communication among microservices.

DEVELOPMENT PROCESS:

The development process of the microservices-based hotel management system involved several key steps:

Project Setup:

- SpringCloud, Eureka for backend development
- Keycloak for authentication and authorization
- MongoDB and MySql for data storage
- Resilience4J for fault tolerance
- Kafka for event-driven architecture
- Postman for testing

Dependencies used:

- ❖ Lombok
- ❖ spring Boot
- ❖ spring web
- ❖ spring web flux
- ❖ spring data MongoDB
- ❖ spring security
- ❖ MySQL Driver
- ❖ Resilience4J
- ❖ Eureka Server
- ❖ Keycloak
- ❖ Apache Kafka
- ❖ Zipkin

MICROSERVICES BUILDING:

For each service, our logical architecture comprises a controller layer and a repository layer. Client HTTP requests are received in the controller layer, and subsequent business logic is executed in the service layer. Some services also involve communication with a message queue, particularly the booking and notification services. Following message queue communication, data is stored in a MySQL or MongoDB database, facilitated by the repository layer. This structure is consistent across all services, ensuring uniformity and efficiency throughout the system.

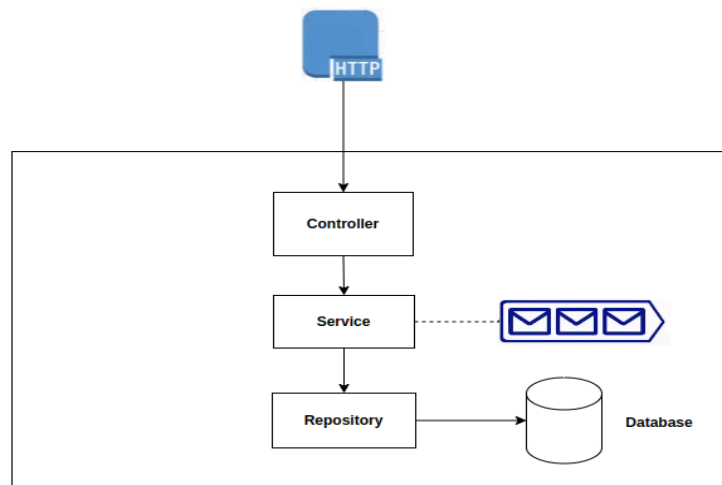


Fig2: Request flow

METHODOLOGY:

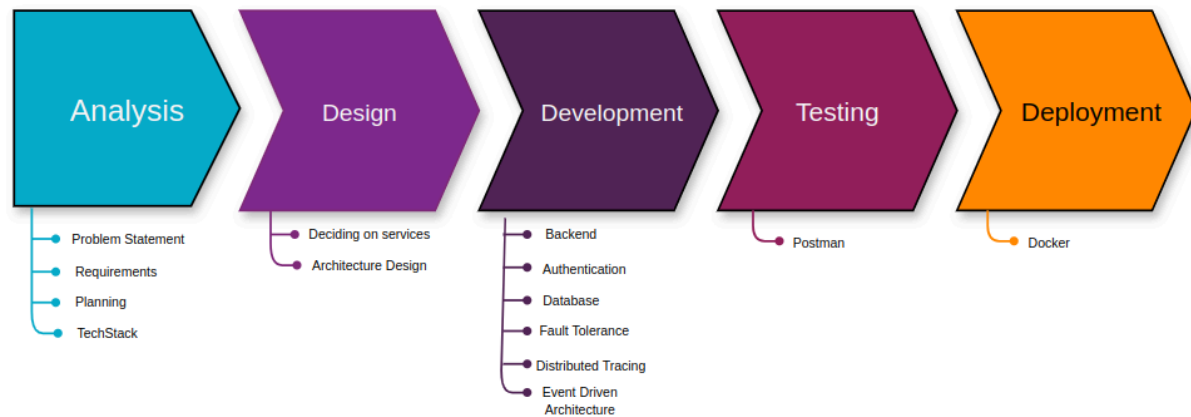


Fig3: Project Methodology

The project methodology consists of five stages, starting with Analysis (Stage-1), where we carefully selected the problem statement, identified requirements, determined the necessary technology stack, and planned the subsequent steps.

Moving to Stage 2, the Design phase, we decided on the required services, parameters, and overall architecture design.

In Stage 3, the Development phase, we created various services, including backend, Authentication, Database, Fault Tolerance, Tracing, and Event-Driven Architecture.

Once the implementation was completed, we transitioned to Stage 5, where we thoroughly tested the project's functionality using Postman. Finally, we utilized Docker to containerize some services for streamlined deployment.

SERVICES:

Hotel Service:

The Hotel Booking Microservice is a crucial component of our system that handles the addition of hotels to our database, pricing, and description.

Creates a hotel-id after adding the hotel.

POST - Adds the hotels to the MongoDB database.

GET - Lists out all the hotels in the database

Bookings Service:

The Bookings service Microservice in our system helps us make the booking and the number of rooms required. This facilitates synchronous communication between bookings-service and inventory service to check the availability of rooms in a particular hotel.

Inventory Service:

Inventory management systems are integral to hotel management, aiding resource optimization by consistently maintaining optimal inventory levels. This microservice is intricately crafted to monitor room availability and associated details concerning hotel accommodations.

The primary objectives of the Inventory Management microservice include:

- Tracking the bookings.
- Managing bookings and replenishment of supplies.
- Ensuring accurate availability of information for users.

InterProcess Communication:

Implementing synchronous communication between microservices using Spring's Web Client, emphasizing efficient communication strategies and demonstrating the steps involved in configuring and utilizing Web Client for making HTTP requests within microservices.



Synchronous Communication

Fig4: Interprocess Communication

Implementing event-driven architecture using Kafka for asynchronous communication in microservices to process events like bookings and notifications efficiently

Discovery Server:

In this scenario, the order service faces the challenge of determining which instance of the inventory service to call, as hardcoded IP addresses are unreliable due to potential instances going down unpredictably. To address this issue, we employ a pattern known as the service discovery pattern.

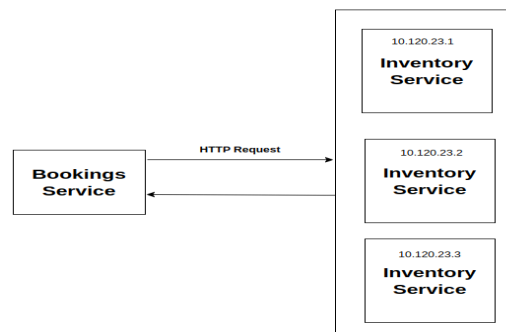


Fig5: Discovery Server

A discovery server is a repository storing information about various services, including their names and IP addresses. When microservices start up, they register with the discovery server, which maintains this information in its local registry. When one service needs to communicate with another, it queries the discovery server for the location of the desired service. Upon receiving a request, the discovery server responds with the corresponding IP address. This enables services to locate and communicate with each other dynamically without hardcoded URLs.

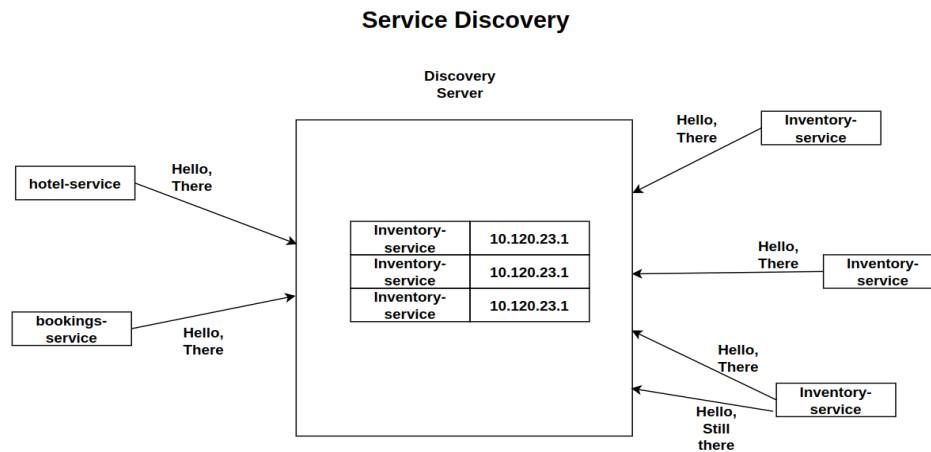


Fig6: Service Discovery

Furthermore, upon the initial request to the discovery server, it sends its registry as a response to the client. The client stores this registry locally. If the discovery server is unavailable, the client consults its local copy to find the service. It iterates through the entries until an available instance is found. If all instances are down, communication fails, indicating the service's unavailability.

Client's local copy of service Registry

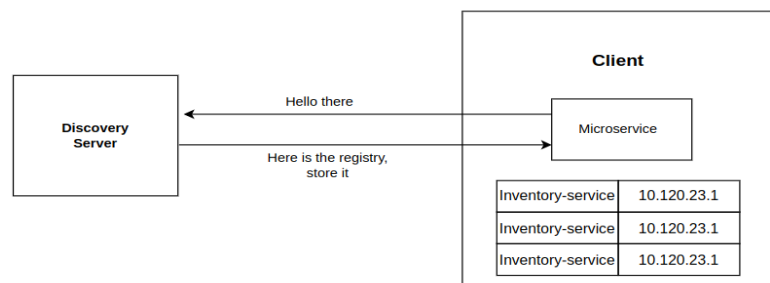


Fig7: Client's legal copy

Enhancing a method to return a list of inventory responses, changing parameter types in the inventory controller, updating the order service to communicate with the inventory service, handling service discovery using Netflix Eureka, and creating a discovery server module using Spring Cloud Netflix Eureka server.

API Gateway:

Implementing service discovery using the Eureka server and client to balance loads effectively, performing disruptive tests by taking down the discovery server to observe system behavior, and introducing an API Gateway to handle requests and efficiently route them to different services in a microservices architecture. It also delves into configuring and implementing the API Gateway using Spring Cloud Gateway and integrating it as a discovery client.

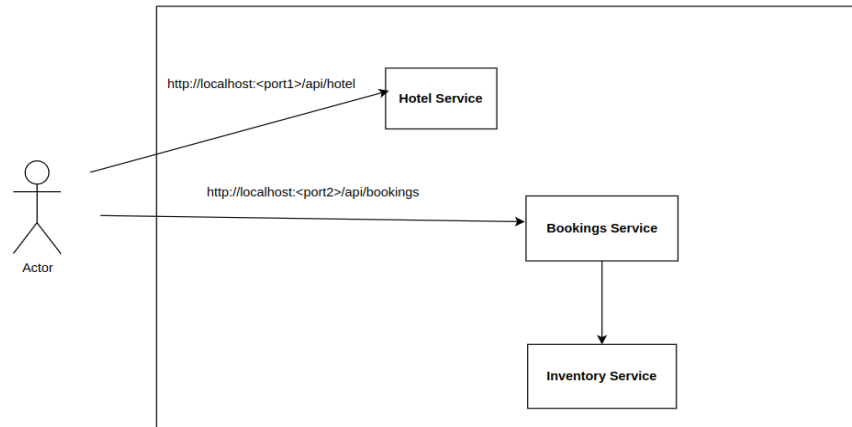


Fig8: Monolithic implementation

Defining routes within an API Gateway by setting up routes for different services, modifying requests and responses using predicates and filters, and configuring load balancing. It also delves into troubleshooting issues like port conflicts, accessing the Eureka server through the API Gateway, and securing microservices with Keycloak for authentication and authorization.

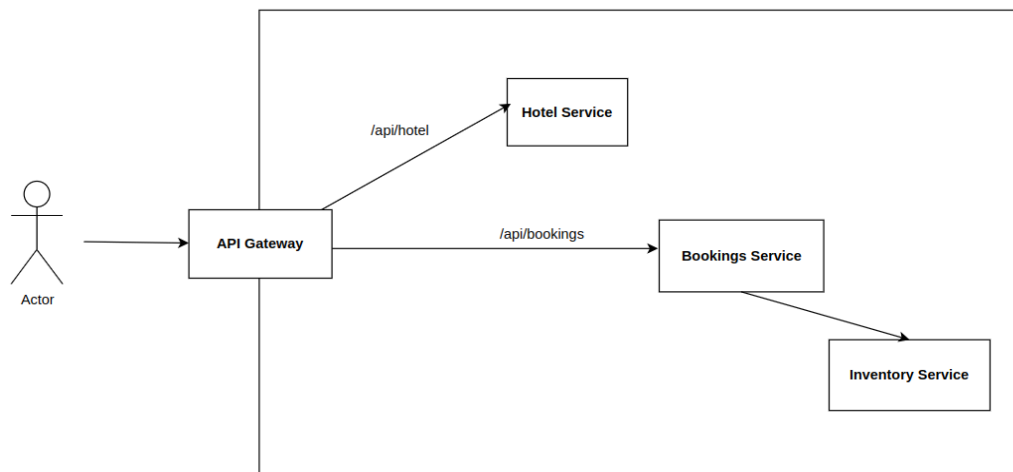


Fig9: Microservice implementation

Installing Keycloak by running a Docker image on port 8181 instead of the default 8080 to avoid conflict with an API Gateway. After setting up Keycloak, configuring the API Gateway to communicate with Keycloak involves adding dependencies, specifying the issuer URI, and creating a security configuration class. Testing the secured endpoints with Postman involves requesting a JWT token from Keycloak, providing credentials, and utilizing the token for authentication. Securing microservices using the Keycloak authorization server and then delves into implementing a circuit breaker pattern for resilient communication between services. The process involves configuring basic authentication to access the Eureka Discovery server using Spring Boot Starter Security.

Notification Service:

To configure the notification service, add the application.properties file and integrate dependencies for the Eureka client, Zipkin, via start.spring.io. This process encompasses configuring essential parameters such as the Eureka client service URL and server port, enabling Spring Sleuth integration, specifying the Zipkin URL, setting up Kafka configurations, and managing serialization/deserialization. Furthermore, it includes troubleshooting issues associated with configuring the Kafka consumer group ID and explores Dockerizing the services through Docker compose.

CIRCUIT BREAKER:

Spring Cloud Circuit Breaker is utilized to implement fault tolerance using Resilience4J. The project involves adding dependencies like Spring Cloud Starter Circuit Breaker Resilience 4J and Spring Boot Starter Actuator, configuring Resilience 4J properties, integrating circuit breaker logic in the application, implementing fallback logic, testing the circuit breaker functionality, and handling slow network connections using timeout in Resilience4J.

Circuit Breaker

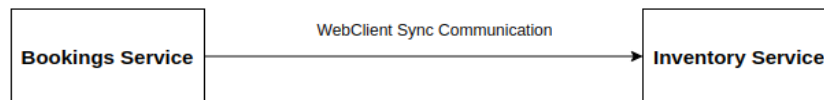


Fig10: Circuit Breaker

Here's a brief overview of states in circuit breaker and how it works:

Closed State: Initially, the circuit breaker is in a closed state. In this state, requests can be passed on to the external service. The circuit breaker monitors the responses from the service.

Open State: If the circuit breaker detects that the external service is failing or responding slowly beyond a certain threshold, it transitions to the open state. In this state, requests to the external service are blocked, and a fallback mechanism (if implemented) may be triggered to handle the failure gracefully.

Half-Open State: After a certain period of time or based on predefined conditions, the circuit breaker may transition to a half-open state. In this state, a limited number of requests can be passed through to the external service to check if they have recovered. If these requests are successful, the circuit breaker transitions back to the closed state. Otherwise, it remains in the open state.

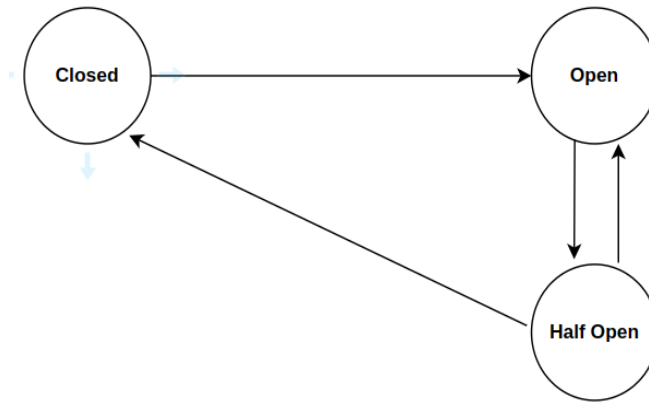


Fig11: Circuit Breaker States

DISTRIBUTED TRACING:

Distributed tracing is a methodology used in software development and system monitoring to track and analyze requests' flow through a distributed system or microservices architecture. It involves instrumenting applications to generate trace data, which contains information about the path of a request as it passes through various services and components. Distributed tracing in handling timeout exceptions and visualizing request paths in Zipkin. Additionally, it delves into implementing event-driven architecture using Kafka for asynchronous communication in microservices to process events like order placements and notifications efficiently.

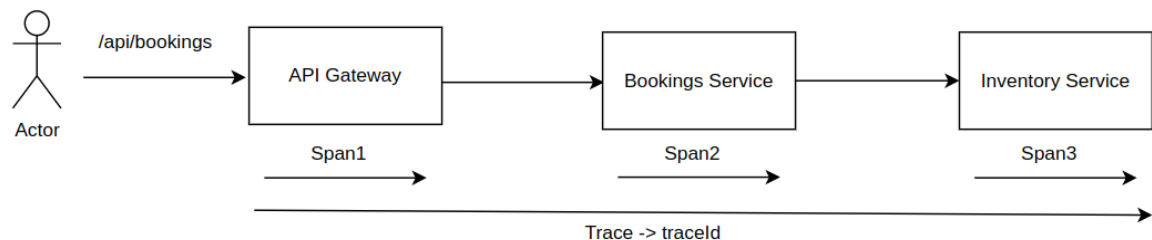


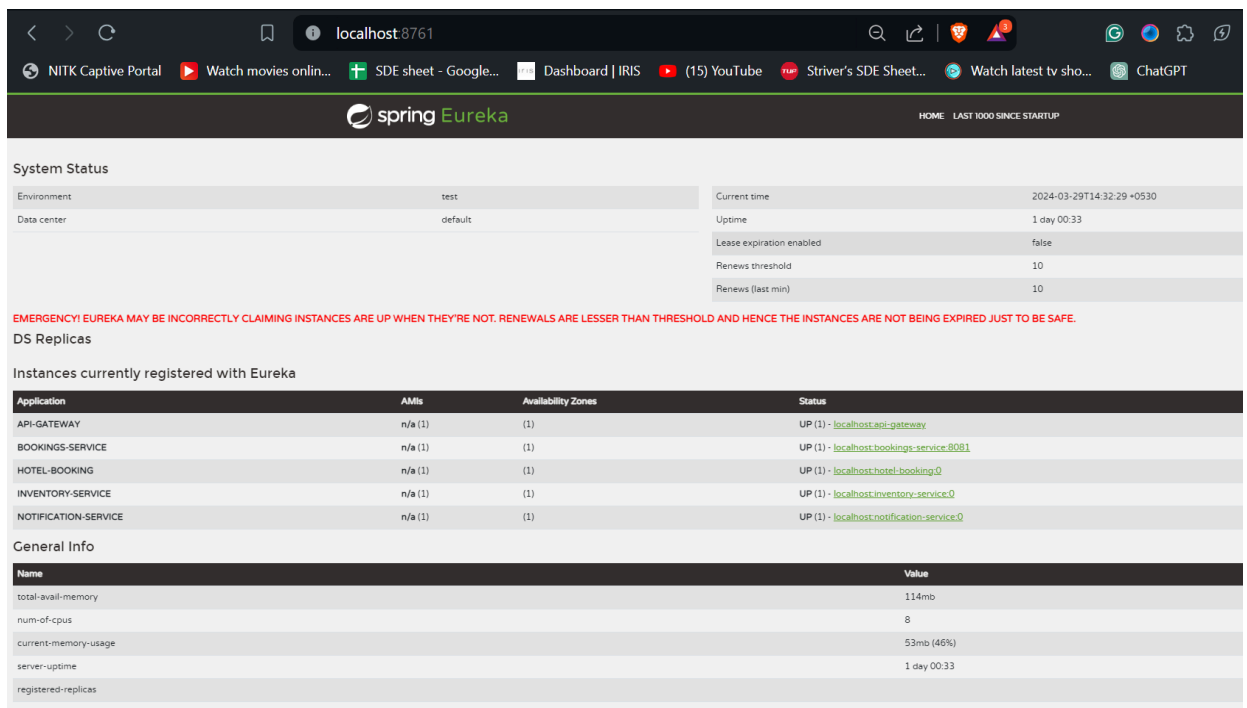
Fig12: Distributed Tracing

SETTING UP KAFKA:

Configuring Kafka within a Spring Boot project involves leveraging the Spring for Apache Kafka library. This entails configuring Kafka templates for message transmission, defining Kafka listeners, setting up serializers, creating relevant Java classes, and establishing communication between services using Kafka topics. The process includes downloading necessary dependencies, organizing modules, specifying Kafka listeners, managing events, and validating the notification service's operational effectiveness through testing.

TESTING:

Eureka Server:



The screenshot shows the Spring Eureka discovery server interface in a web browser at localhost:8761. The browser's address bar and tabs are visible at the top. The Eureka logo and 'HOME LAST 1000 SINCE STARTUP' are in the header. The main content area is divided into several sections:

- System Status:** A table showing environment details.

Environment	test	Current time	2024-03-29T14:32:29 +0530
Data center	default	Uptime	1 day 00:33
		Lease expiration enabled	false
		Renews threshold	10
		Renews (last min)	10
- Emergency Warning:** A red text message: "EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE."
- DS Replicas:** A section titled "Instances currently registered with Eureka".
- Registered Instances Table:**

Application	ADMs	Availability Zones	Status
API-GATEWAY	n/a (1)	(1)	UP (1) - localhost:api-gateway
BOOKINGS-SERVICE	n/a (1)	(1)	UP (1) - localhost:bookings-service:8081
HOTEL-BOOKING	n/a (1)	(1)	UP (1) - localhost:hotel-booking:0
INVENTORY-SERVICE	n/a (1)	(1)	UP (1) - localhost:inventory-service:0
NOTIFICATION-SERVICE	n/a (1)	(1)	UP (1) - localhost:notification-service:0
- General Info:** A table showing system metrics.

Name	Value
total-avail-memory	114mb
num-of-cpus	8
current-memory-usage	53mb (46%)
server-up-time	1 day 00:33
registered-replicas	
unavailable-replicas	

Fig13: Eureka discovery server

Fig13 is the Eureka discovery server working in port 8761. Here we can find all the services we are using and other details about the work environment etc.

Postman:

Postman is used to test the working condition of the application. Postman is a widely used API development tool that simplifies the process of testing, documenting, and sharing APIs. It provides an intuitive user interface for sending HTTP requests, inspecting responses, and organizing API workflows.

Access Token:

Fig14 shows the access token the postman generated after giving it the OAuth2.0 keycloak credentials. This will be included in the header while sending the HTTP request to the system.

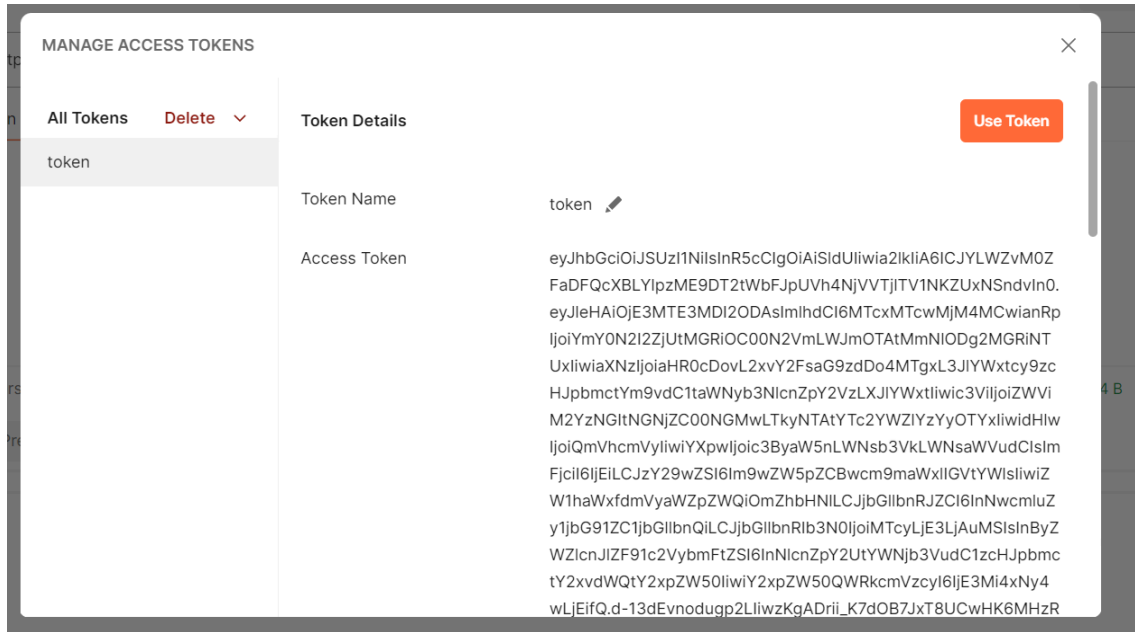


Fig14: New access token generation

Hotel Service:

Fig15 and Fig16 test GET and POST requests from hotel service, respectively. Get request will return all the hotels present in the MongoDB database. Here, our request will be redirected to hotel service after API Gateway does authentication.

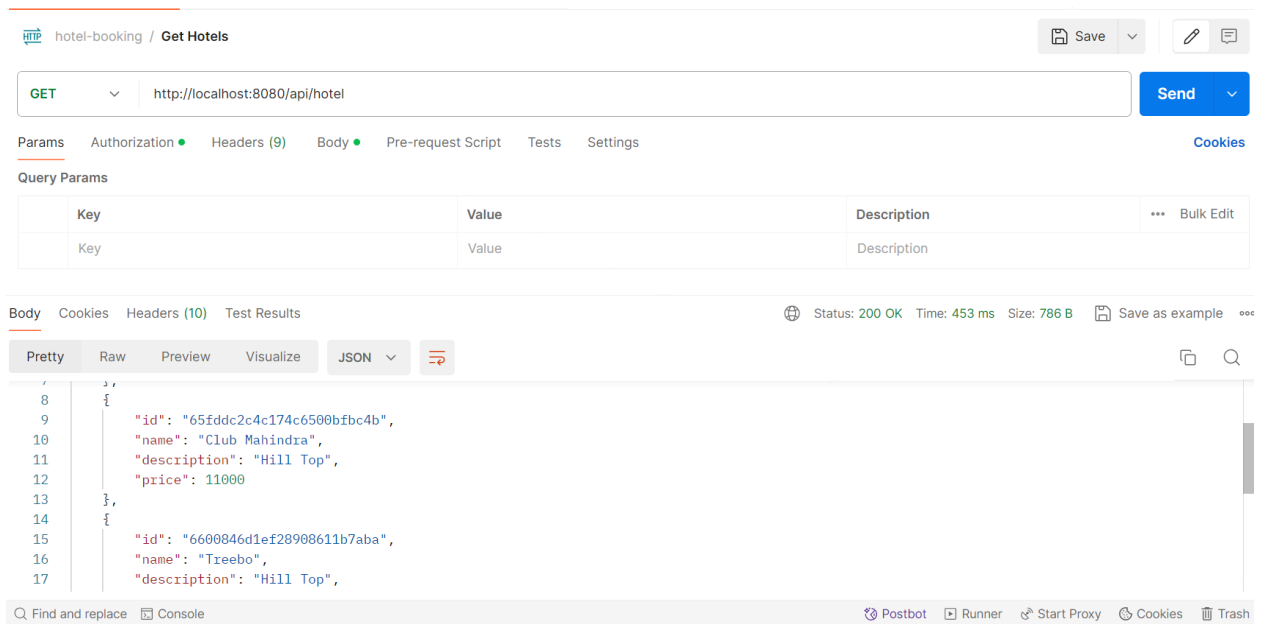


Fig15: Get request to hotel service

Post Request will update the hotel database by adding new hotel details that we have sent along with the request attached in the body.

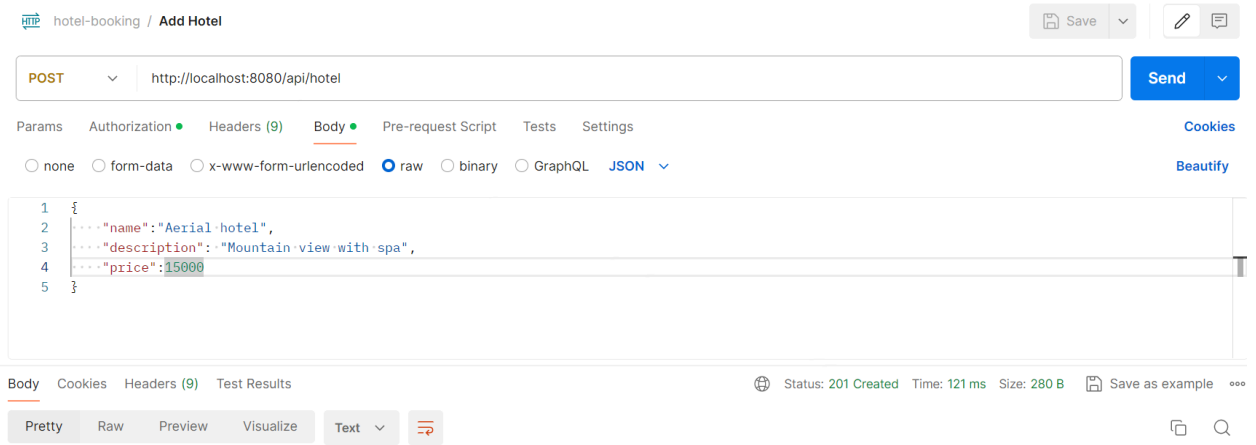


Fig16: Post Request to hotel service

Booking Service:

Fig17 is the post request for the booking service. It will create a new booking for the request after the authentication with the auth server and verification of room availability with inventory service. When the booking is made, we can observe a response saying, "Booking made successfully."

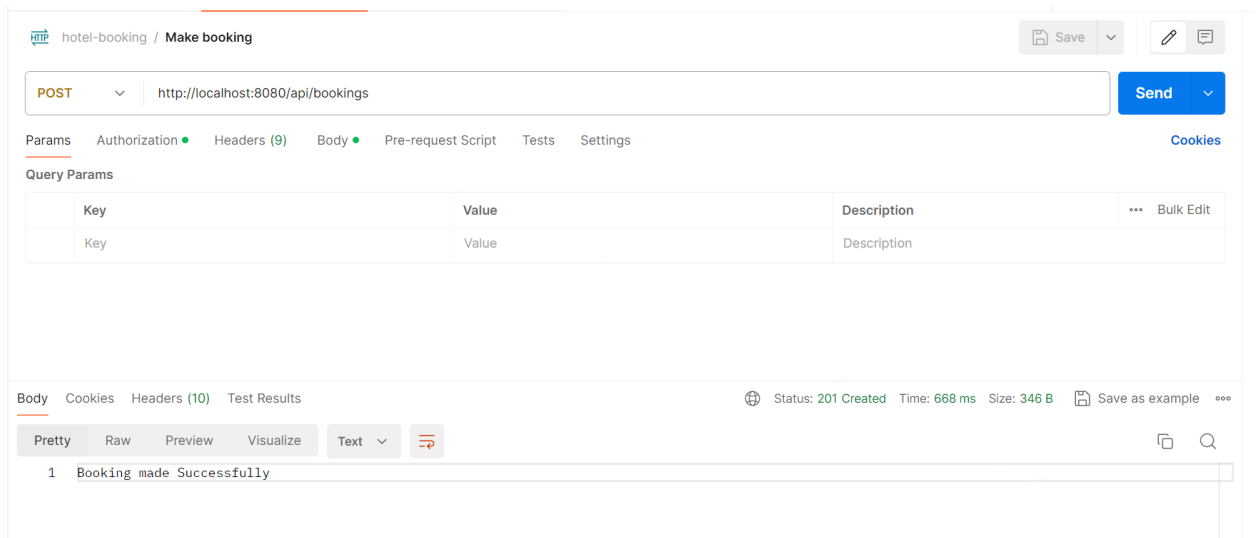
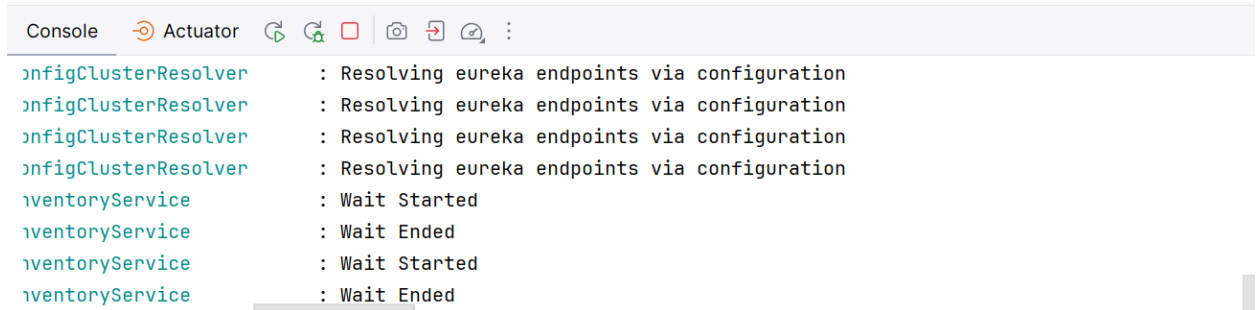


Fig17: Post Request to Bookings service

Circuit Breaker:

The circuit breaker is working in the inventory service log, as shown in Fig18. Here, we can observe that timeout and retry properties have been implemented.

The timeout property will end the program if a specific time limit is reached, and the retry property will keep trying to send requests until the request is successful or the number of tries limit is reached.



```
Console  Actuator  [Icons]
nfigClusterResolver : Resolving eureka endpoints via configuration
nfigClusterResolver : Resolving eureka endpoints via configuration
nfigClusterResolver : Resolving eureka endpoints via configuration
nfigClusterResolver : Resolving eureka endpoints via configuration
ventoryService      : Wait Started
ventoryService      : Wait Ended
ventoryService      : Wait Started
ventoryService      : Wait Ended
```

Fig18: Timeout and retry properties

Fig19 is the actuator. Here, we can see the condition of the service, i.e., in which state it is, the number of requests received, etc. The actuator runs in port 8081 in this example.

```
{
  "status": "UP",
  "components": {
    "circuitBreakers": {
      "status": "UP",
      "details": {
        "inventory": {
          "status": "UP",
          "details": {
            "failureRate": "0.0%",
            "failureRateThreshold": "50.0%",
            "slowCallRate": "0.0%",
            "slowCallRateThreshold": "100.0%",
            "bufferedCalls": 5,
            "slowCalls": 0,
            "slowFailedCalls": 0,
            "failedCalls": 0,
            "notPermittedCalls": 0,
            "state": "CLOSED"
          }
        }
      }
    },
    "db": {
      "status": "UP",
```

Fig19: Actuator

Zipkin:

Zipkin is an open-source distributed tracing system that helps developers gather, visualize, and troubleshoot requests as they travel through complex distributed systems. It provides insights into latency issues and dependencies between services, which helps diagnose.

Fig20 shows the Zipkin server, which captured the requests sent.

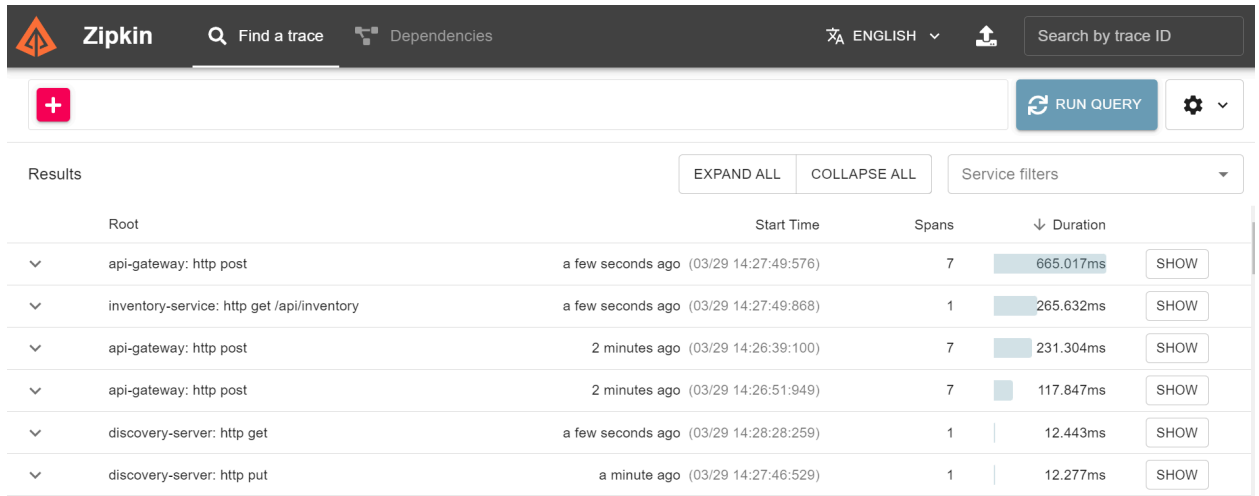


Fig20: Zipkin to trace all the requests

Fig21 is the complete diagnosis of an HTTP request sent to the booking service. It also shows the request from API Gateway to the authentication server. We can also find the span ID and other details of the request.

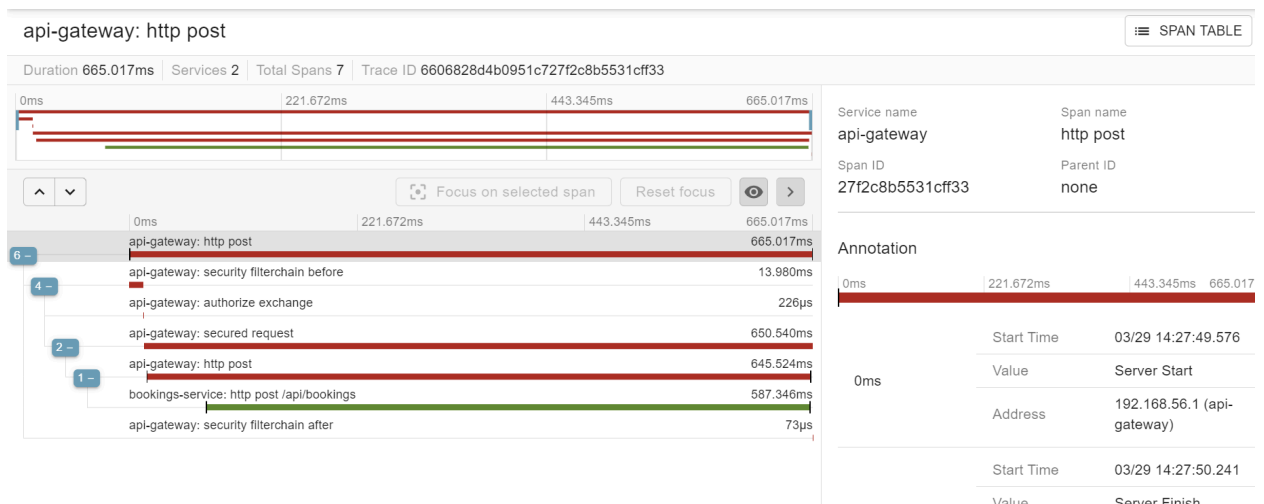
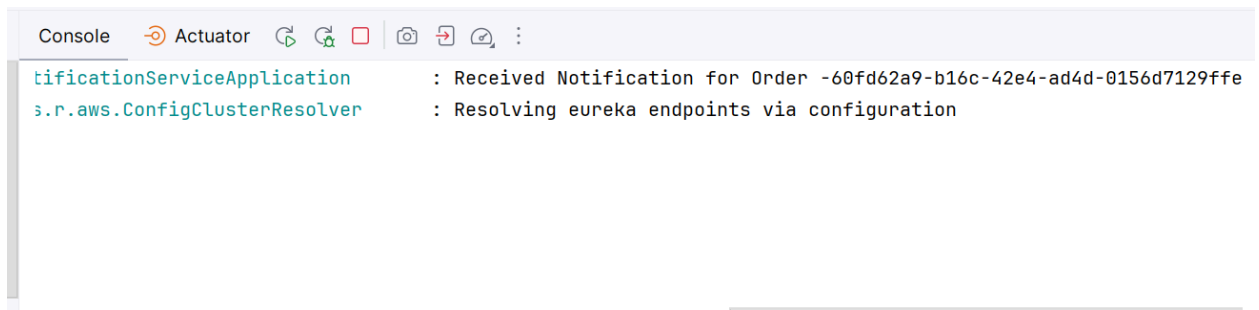


Fig21: Zipkin to trace specific request

Notification Service:

Fig22 is the notification service output in the console log. This can prove that the notification service can identify the order ID of the booking, which can be used for further steps.



```
Console  Actuator  [Icons]  :
NotificationServiceApplication      : Received Notification for Order -60fd62a9-b16c-42e4-ad4d-0156d7129ffe
s.r.aws.ConfigClusterResolver       : Resolving eureka endpoints via configuration
```

Fig22: Notification Server console

CONCLUSION:

This project successfully developed a software application utilizing an event-driven architecture. The project followed a structured Software Development Life Cycle (SDLC) methodology, ensuring a clear definition of requirements, robust architecture design, and comprehensive service implementation. The application leverages backend services, authentication mechanisms, a database, fault tolerance strategies, tracing capabilities, and the core event-driven architecture. Rigorous testing using Postman verified application functionality, while containerization with Docker paves the way for streamlined deployment. This project demonstrates the successful development of a complex software solution with the potential for high functionality, adaptability, and scalability.

FUTURE WORK:

1. Feature Enhancements:

Providing more new features related to hotel management will be a significant improvement to the project. This can be turned into a full-scale production application with a few more services in place

2. Scalability Enhancements:

Conduct a load testing analysis to identify performance limitations under high event volume. Implement horizontal scaling for backend services to distribute the processing load across multiple instances and improve scalability. Explore event-sharding techniques to partition event streams based on specific criteria, enhancing processing efficiency and scalability.

3. Frontend Integration:

Develop a user-friendly front-end application to interact with the event-driven architecture. This could involve event visualization, which creates dashboards or interfaces to display real-time and historical event data, providing users with insights into event flows and system activity. Implement functionalities for users to subscribe to specific events, filter and search events, and trigger event generation based on user actions.

4. Security Enhancements:

Develop a role-based access control (RBAC) system for event producers and consumers, ensuring proper authorization for event creation and consumption.

Implement message signing or encryption for sensitive event data to ensure data integrity and confidentiality during transmission.