

MICROSERVICES

What are Microservices?

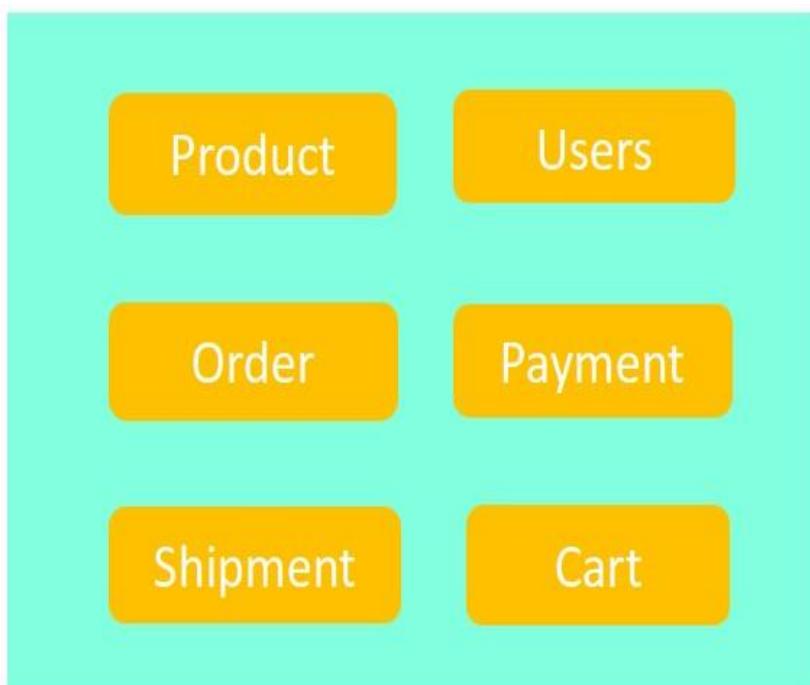
Microservices, also referred to as the microservice architecture, represent an approach to structuring an application as a set of services characterized by:

- Independent deployability
- Loose coupling
- Organization around business capabilities
- Ownership by small teams

This architectural style empowers organizations to swiftly, regularly, reliably, and sustainably deliver extensive and intricate applications—a necessity for competitiveness and success in today's landscape.

Before Microservices

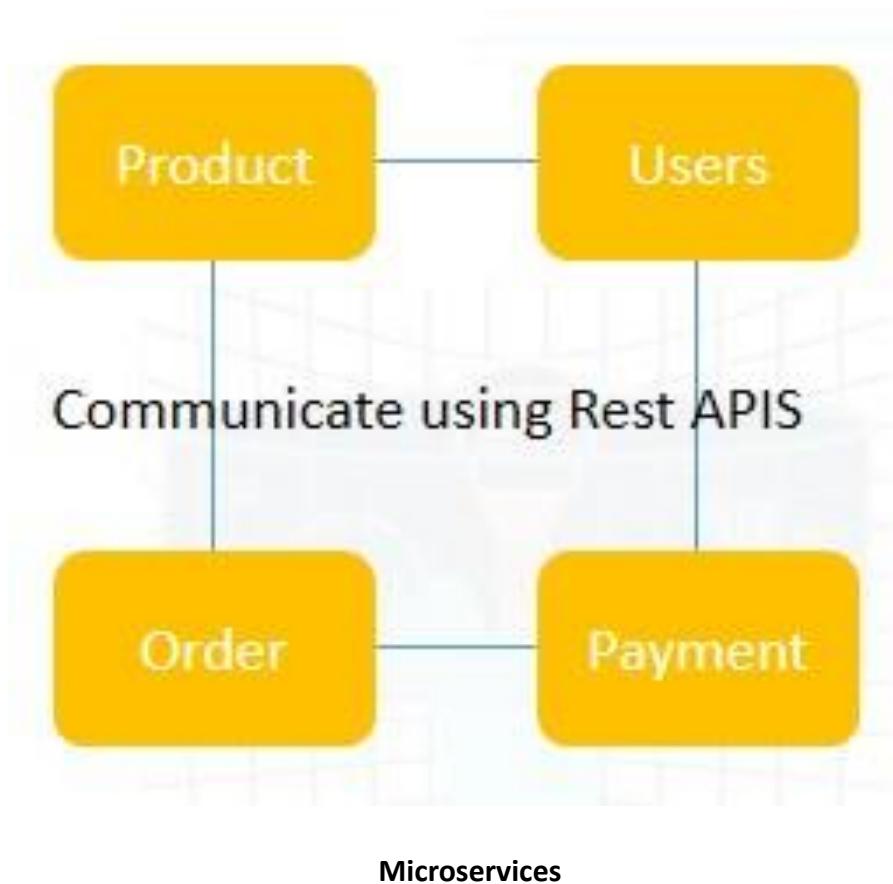
- Monolithic architecture combined multiple components into a single large application. Utilized a single code base.
- Deployed as a single bundle.
- Modifications in one service necessitated redeploying the entire application.
- Encountered challenges in communication among developers.
- Faced scalability issues over time



Application using Monolithic Architecture

Microservices

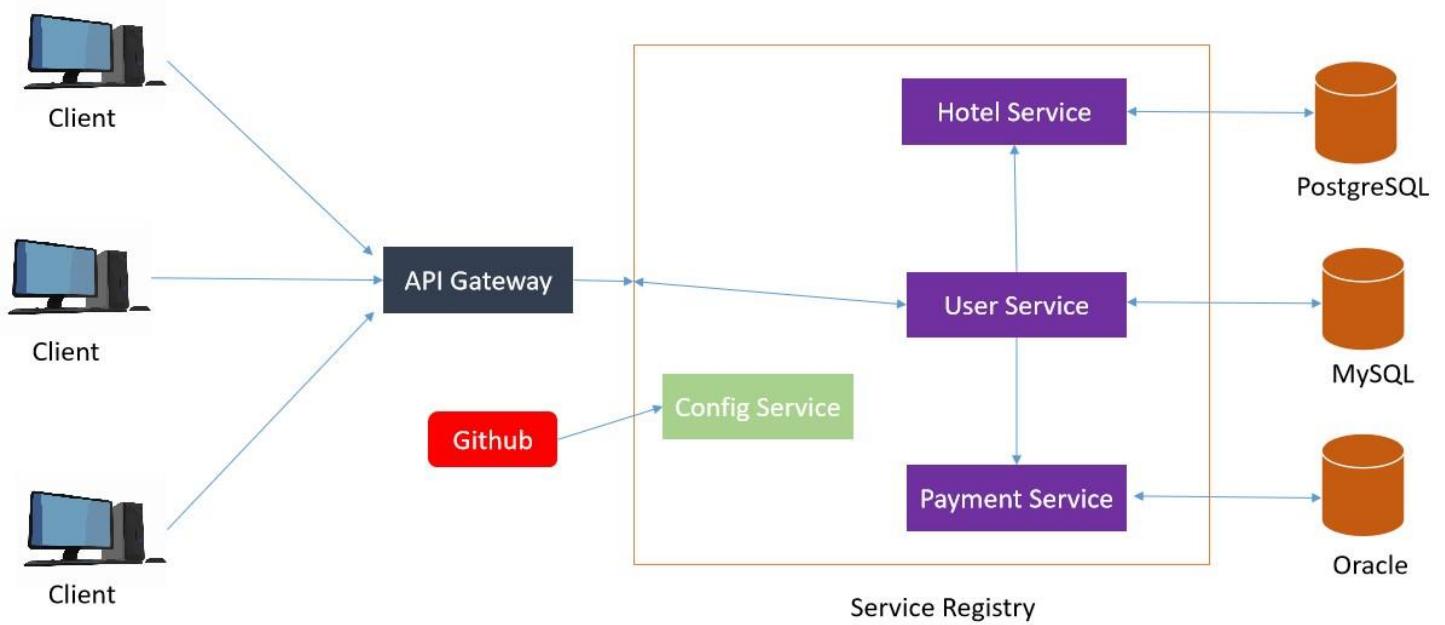
- Breaking down extensive applications into smaller segments.
- Utilizing distinct codebases.
- Managing each module autonomously.
- Employing varying technology stacks.
- Managing microservices poses complexity.



Microservice Architecture

It represents an architectural design that structures an application as a set of loosely interconnected, nely detailed services that communicate through lightweight protocols. One of its primary objectives is to enable teams to independently develop and deploy their services without reliance on others. This goal is achieved by minimizing dependencies within the code base, enabling developers to evolve their services with minimal constraints imposed by users, and shielding additional complexity from users.

Consequently, organizations can rapidly expand and scale software, as well as integrate o the-shelf services more seamlessly. This approach reduces communication requirements. However, these advantages entail the challenge of maintaining decoupling. Careful design of interfaces and their treatment as public APIs is necessary. One approach involves utilizing multiple interfaces within the same service or creating multiple versions of the service to ensure smooth integration without disrupting existing users of the code.



Service Registry:

A Service Registry serves as a centralized system responsible for managing and structuring information regarding services in a distributed computing environment. In the realm of software development and system architecture, services denote autonomous, modular elements that execute specific functions or duties. These services encompass various types such as microservices, web services, or other software components.

The primary objective of a Service Registry lies in its role as a directory or repository housing details about the accessible services within a network. It aids in streamlining communication and synchronization among different services by enabling them to dynamically discover each other. Whenever a service is deployed or becomes accessible, it registers its whereabouts and associated metadata within the Service Registry. Subsequently, other services can interrogate the registry to identify and pinpoint the services they require.

Crucial concepts of Service Registry

Service Registration: When a service initiates or becomes accessible, it autonomously enrolls itself within the Service Registry. This enrollment typically entails providing details such as the service's location (including host and port), endpoint URLs, and pertinent metadata.

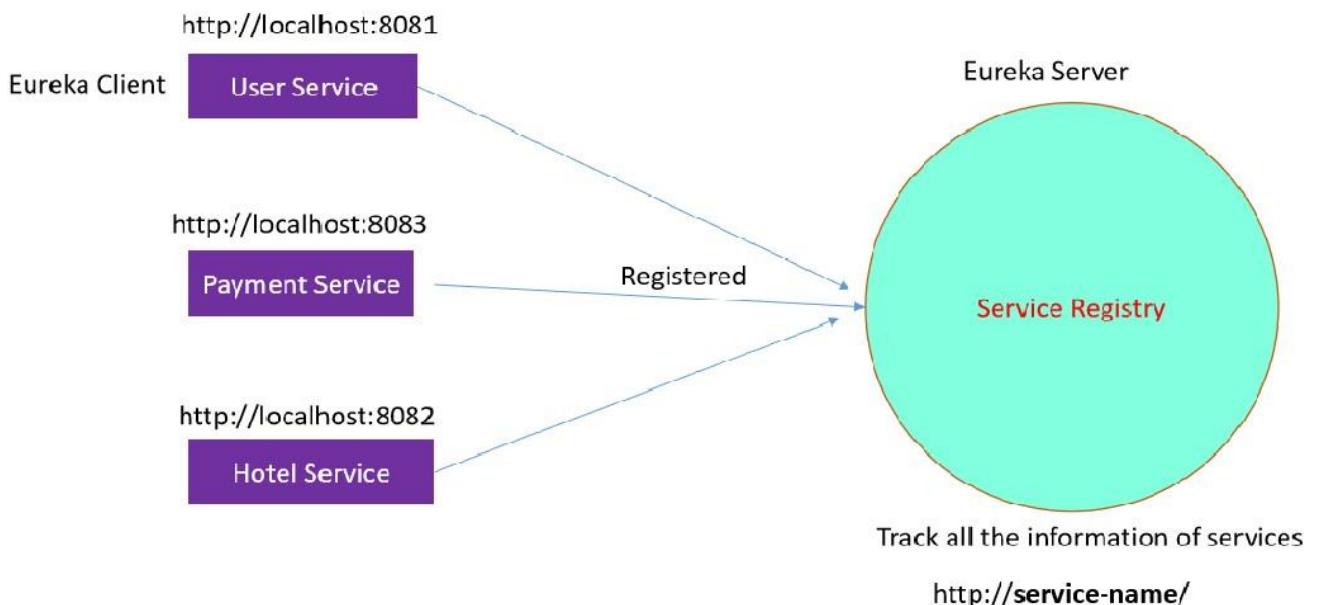
Service Discovery: Other services have the capability to retrieve information from the Service Registry, enabling them to pinpoint the location and specifics of the services they rely on. This fosters adaptable and dynamic communication among services within a distributed system.

Health Checking: Certain Service Registries integrate health checks to verify the operational status of registered services. This serves to prevent directing requests to services that might be currently unavailable or encountering operational issues.

Load Balancing: In conjunction with aiding service discovery, a Service Registry might also support load balancing by equitably distributing requests across multiple instances of a service. This ensures optimal utilization of resources.

Distributed Systems: Service Registries prove particularly valuable in distributed systems where services are deployed across diverse servers, containers, or cloud instances.

Service Registry



Building a Eureka Server:

Server dependency:

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId> </dependency>
```

Server Annotation:

The `@EnableEurekaServer` annotation, part of the Spring Framework, serves the purpose of activating Eureka Server functionality within a Spring Boot application. Eureka stands as a service registry and discovery tool that facilitates communication among microservices within a distributed system. Incorporating `@EnableEurekaServer` in your Spring Boot application configures it to operate as a Eureka Server.

Actions that occur upon utilizing @EnableEurekaServer

Service Registration and Discovery: The Eureka Server manages the registration and discovery of services. Services, including microservices, register themselves with the Eureka Server during startup. This server maintains a repository containing information about available services and their respective locations.

Annotation in Spring Boot: @EnableEurekaServer, a specialized Spring Boot annotation, designates a configuration class as a Eureka Server. Upon implementation, this annotation triggers the necessary configuration and setup to enable your Spring Boot application to function as a Eureka Server.

Configuration Class: Typically, you apply the @EnableEurekaServer annotation to a class annotated with either @SpringBootApplication or @Configuration. This designated class acts as the entry point for your Eureka Server application.

```
package com.registry.service.registry;
import org.springframework.boot.SpringApplication; import
org.springframework.boot.autoconfigure.SpringBootApplication; import
org.springframework.cloud.netflix.eureka.server.EnableEurekaServer; @SpringBootApplication
@EnableEurekaServer
public class ServiceRegistryApplication { public static void
main(String[] args) {
    SpringApplication.run(ServiceRegistryApplication.class, args);
}
}
```

In this example:

@SpringBootApplication signifies the classification of this class as a Spring Boot application. @EnableEurekaServer activates the capabilities of the Eureka Server. After the successful launch of your Eureka Server, additional microservices can enlist themselves with it, while clients gain the ability to explore and interact with these registered services via the Eureka Server.

Ensure the inclusion of essential dependencies in your project, such as spring-cloud-starter-netflix-eureka-server, to incorporate the necessary Eureka Server dependencies within your classpath. Additionally, configure the application properties or YAML file to define Eureka Server settings if required.

Building a Eureka Client

Client dependency

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId> </dependency>
```

Client Annotation:

@EnableDiscoveryClient is an additional annotation provided within the Spring Cloud framework, utilized to activate service discovery functionalities within a Spring Boot application. Unlike @EnableEurekaServer, which is specifically employed for establishing a Eureka Server, @EnableDiscoveryClient is utilized in microservices aiming to enlist themselves within a service registry (such as Eureka, Consul, or others) for discovery by other services.

Here's the functionality of @EnableDiscoveryClient:

Service Registration: Upon incorporating the @EnableDiscoveryClient annotation into a Spring Boot application, it gains the ability to register itself within a service registry. This registration typically contains essential information about the service, such as its network location, endpoints, and associated metadata.

Service Discovery: As the application now operates as a service, it can also explore and identify other services enlisted within the same service registry. This functionality proves beneficial in a microservices architecture, enabling dynamic communication among services.

Versatile Service Registries: Spring Cloud offers support for various service registries, including Eureka, Consul, ZooKeeper, and others. The @EnableDiscoveryClient annotation abstracts the intricacies of the underlying service registry implementation, facilitating the seamless transition between different registries without necessitating changes to your application code.

Here's a simple example:

```
package com.user;
import org.springframework.boot.SpringApplication; import
org.springframework.boot.autoconfigure.SpringBootApplication; import
org.springframework.cloud.client.discovery.EnableDiscoveryClient; import
org.springframework.cloud.openfeign.EnableFeignClients;
@SpringBootApplication
@EnableDiscoveryClient @EnableFeignClients
public class UserServiceApplication { public static
void main(String[] args) {
    SpringApplication.run(UserServiceApplication.class, args);
}
```

In this example:

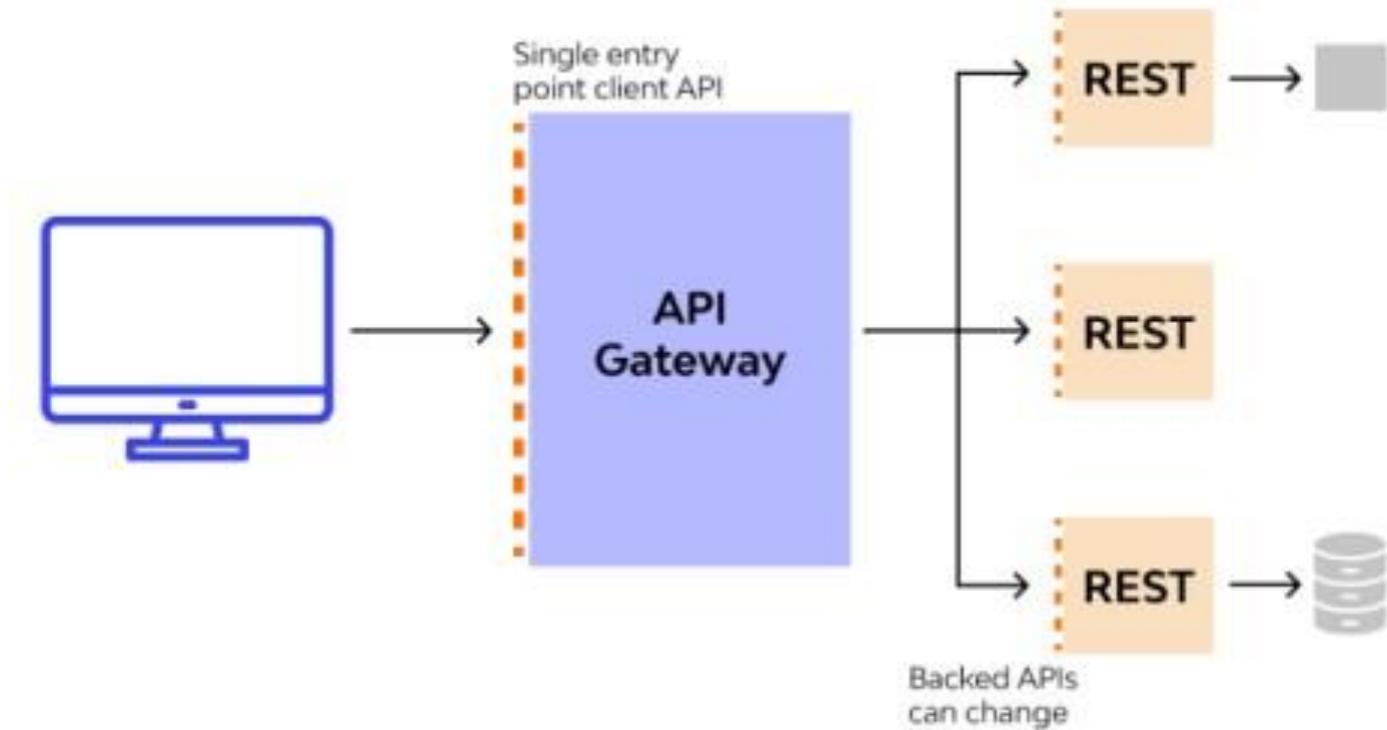
@SpringBootApplication signifies the classification of this class as a Spring Boot application.

@EnableDiscoveryClient activates the service discovery capability. Once a microservice is tagged with @EnableDiscoveryClient, it gains the ability to enlist itself within a configured service registry, while also acquiring the capability to explore other services enlisted within the same registry.

It's essential to note that when employing Eureka as the service registry, Spring Cloud offers a more specific annotation named @EnableEurekaClient. This annotation, a subset of @EnableDiscoveryClient, is specifically designed for integration with the Eureka service registry. In most scenarios, utilizing @EnableDiscoveryClient ensures flexibility for potential transitions to different service registries in the future.

API Gateway:

Within a microservices architecture, an API Gateway functions as a central access point responsible for managing and directing external client requests toward the appropriate microservices. It acts as a communication interface that streamlines interactions between clients and the diverse microservices.



Below are some key roles and objectives fulfilled by an API Gateway within the realm of microservices:

Integration of Microservices:

Microservices typically represent distinct functionalities or elements within a system. An API Gateway consolidates client interactions with multiple microservices through a singular entry point, streamlining the client-side user experience.

Request Routing

The API Gateway effectively channels incoming requests to the appropriate microservice, considering factors such as the request URL, HTTP method, or other pertinent parameters. This routing functionality plays a pivotal role in managing the intricacies of interactions within a microservices ecosystem.

Load Balancing

Through the allocation of incoming requests across various microservice instances, an API Gateway aids in distributing the workload, ensuring efficient resource utilization and sustained high availability.

Dependency:

<dependency>

```
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-gateway</artifactId> </dependency>
```

Routing management

```
spring: cloud:
  gateway:
    routes:
      - id: USER-SERVICE uri:
        lb://USER-SERVICE predicates: -
          Path=/users/** - id: HOTEL-
            SERVICE uri: lb://HOTEL-SERVICE
            predicates: - Path=/hotels/**
      - id: PAYMENT-SERVICE uri:
        lb://PAYMENT-SERVICE predicates:
      - Path=/payments/**
```

Con guration Server

A con guration server stands as a dedicated element entrusted with the task of managing con guration se ings for microservices. Its primary role involves centralizing and externalizing con guration information, streamlining the management and update processes for multiple services. The utilization of a con guration server assists in maintaining uniformity across microservices, facilitates real-time con guration alterations sans the need for service redeployment, and bolsters the overall adaptability of the system.

This is accomplished through the utilization of a Git Hub Repository.

Dependency:

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

Annotation:

The @EnableCon gServer annotation serves as a pivotal annotation within the Spring Cloud Con g Server module. Its purpose lies in enabling the functionality of a con guration server within a Spring Boot application. By annotating a class with @EnableCon gServer, you instruct Spring to con gure the application as a centralized con guration server.

```
@SpringBootApplication @EnableConfigServer
public class ConfigServerApplication { public static
void main(String[] args) {
    SpringApplication.run(ConfigServerApplication.class, args);
}
}
```

Application.yml Configuration

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/Sachin7414/microservices-configuration
          clone-on-start: true
```

Communication Between Microservices By Using Feign Client:

Feign, developed by Netflix, represents a declarative web service client aimed at simplifying the process of executing HTTP requests and engaging with RESTful APIs. It forms part of the Spring Cloud Netflix project, fostering integration with diverse Netflix OSS components for constructing microservices-oriented applications.

Feign proves especially advantageous in microservices architectures wherein one microservice necessitates communication with another via HTTP. Instead of crafting intricate HTTP client code for request handling and response management, Feign empowers developers to articulate the desired interaction with the remote service through a straightforward and intuitive declarative interface.

Dependency:

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-openfeign</artifactId> </dependency>
```

Annotation:

The `@FeignClient` annotation is an integral component of the Spring Cloud OpenFeign library, utilized to designate a client for a microservice. It facilitates the Feign declarative programming paradigm, enabling developers to outline a Java interface detailing the HTTP requests directed towards a particular microservice.

Below is an overview of the `@FeignClient` annotation along with its frequently employed attributes:

```
@FeignClient(name = "HOTEL-SERVICE")
@Service
public interface HotelService { @GetMapping("/hotels/{hotelId}")
    Hotel getHotel(@PathVariable Long hotelId);
    @PostMapping("/hotels")
    Hotel saveHotel(Hotel hotel);
```

}

What is Exception Handling:

Exception handling within microservices entails the oversight and management of errors and exceptions occurring in a distributed system composed of numerous services. In the microservices paradigm, where each service functions autonomously and inter-service communication typically occurs through APIs (e.g., RESTful APIs, messaging systems), robust exception handling plays a pivotal role in ensuring the reliability and durability of the entire system.

Centralized Exception Handling:

Centralized exception handling involves consolidating exception-handling logic at a central point within an application. This consolidation assists in uniformly managing and addressing exceptions across the entirety of the codebase. Often, this approach incorporates a dedicated component or aspect to globally handle exceptions, providing a consistent and systematic approach to handling errors.

@RestControllerAdvice and @ExceptionHandler:

For instance, Spring Boot offers the `@RestControllerAdvice` annotation to establish a universal exception handler. By annotating a class with `@RestControllerAdvice` and defining methods with `@ExceptionHandler` annotations, you can streamline exception handling logic for all controllers within your microservices.

Within a class marked with `@RestControllerAdvice`, you can define methods annotated with `@ExceptionHandler` to manage specific types of exceptions. These methods are invoked upon encountering an exception of the specified type thrown from any controller method within your application.

```
@RestControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(ResourceNotFoundException.class) public
    ResponseEntity<ApiResponse>
    handlerResourceNotFoundException(ResourceNotFoundException ex){
        String message = ex.getMessage();
        ApiResponse response =
        ApiResponse.builder().message(message).success(true).status(HttpStatus.NOT_FOUND).build();
        return new ResponseEntity<ApiResponse>(response, HttpStatus.NOT_FOUND);
    }
}
```

Custom Exception:

A specialized exception class proves beneficial in situations where a segment of your application confronts a scenario where it fails to locate a particular resource. For instance, within a web service, if a client requests a resource that is absent, you could opt to throw a `ResourceNotFoundException` to signify this circumstance. By

extending the `RuntimeException` class, catching this exception explicitly is not obligatory, although you retain the ability to handle it if necessary.

```
public class ResourceNotFoundException extends RuntimeException { public  
ResourceNotFoundException() {  
super("Resource not found on server !!");  
}  
public ResourceNotFoundException(String message) { super(message);  
}  
}  
  
public class ResourceNotFoundException extends RuntimeException { public  
ResourceNotFoundException() {  
super("Resource not found on server !!");  
}  
public ResourceNotFoundException(String message) { super(message);  
}  
}
```

Exception Throw:

Throwing an exception denotes the action of indicating that an exceptional or uncommon condition or error has occurred during program execution. When an exception is thrown, it disrupts the regular flow of execution and transfers control to a designated exception handling block.

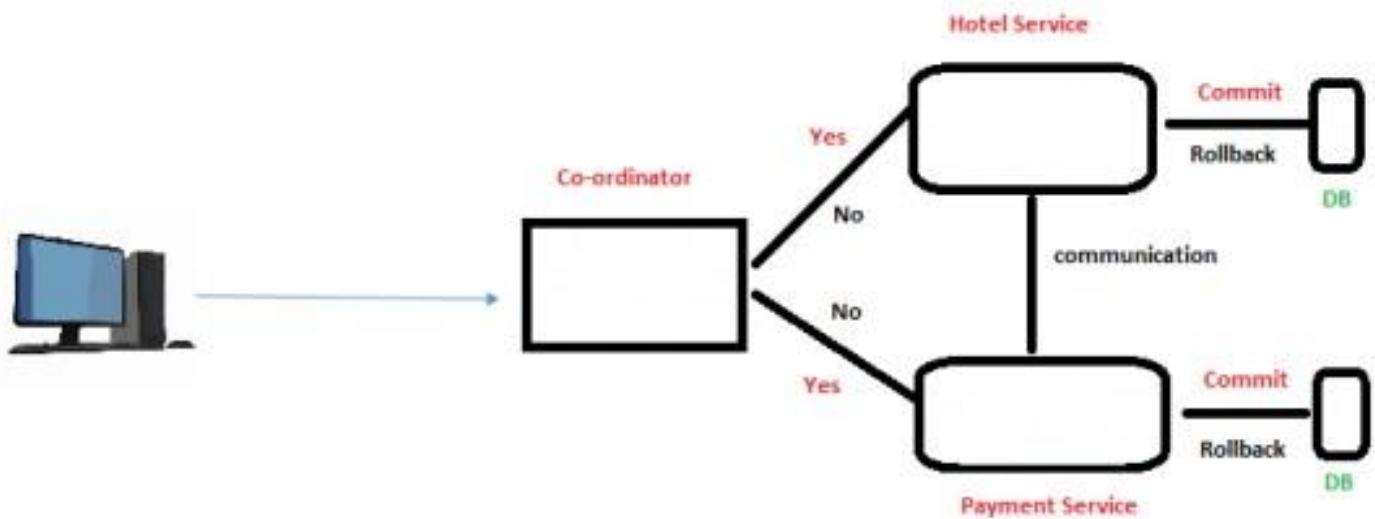
```
@Override  
public User getuser(Long id) {  
User user = userRepository.findById(id).orElseThrow(() -> new  
ResourceNotFoundException("User with given id not found on server!! " + id)); List<Payment> payments =  
paymentService.getAllPaymentByUserId(user.getUserId()); payments.forEach(payment -> {  
Hotel hotel = hotelService.getHotel(payment.getHotelId());  
List<Booking> userBookings = hotel.getBookings().stream()  
.filter(booking -> booking.getUserId().equals(payment.getUserId()))  
.collect(Collectors.toList()); hotel.setBookings(userBookings);  
payment.setHotels(hotel);  
});  
user.setPayments(payments);  
return user;  
}
```

Designing distributed transactions presents unique challenges due to involving multiple participants situated in different locations, necessitating coordination to preserve overall transaction consistency and integrity. Challenges include network failures, latency issues, and the essential need for coordination among dispersed components.

Two-Phase Commit (2PC) stands as a distributed transaction protocol specifically engineered to guarantee transactional atomicity across multiple transactional resources like databases or services. In the realm of

microservices, where autonomous services engage in transactions through communication, 2PC serves as a mechanism to uphold consistency throughout distributed systems.

The primary objective is to ensure either the successful completion or failure of a transaction without leaving it in an inconsistent state. Two-Phase Commit serves to render distributed transactions atomic, thereby maintaining their cohesive nature.



Hotel Service

```
@Override  
public Hotel saveHotel(Hotel hotel) {  
    hotel.setStatus(HotelStatus.CREATED.name());  
    return  
    hotelRepository.save(hotel);  
}  
  
@Override public Booking rollbackHotel(Booking booking) {  
    booking.setStatus(HotelStatus.ROLLBACK.name());  
    return  
    bookingRepository.save(booking);  
}  
  
@Override public Booking commitHotel(Booking booking) {  
    booking.setStatus(HotelStatus.BOOKED.name());  
    return  
    bookingRepository.save(booking);  
}
```

Payment Service:

```

@Override
public Payment savePayment(Payment payment) {
    Booking booking=new Booking();
    Hotel hotel = hotelService.getHotel(payment.getHotelId()); if(Double.compare(payment.getPrice(),
    hotel.getPrice())==0){
        if(hotel.getStatus().equalsIgnoreCase(" ")){
            Payment payment1 = commitPayment(payment);
            booking.setUserId(payment1.getUserId());
            booking.setHotelId(hotel.getId());
            hotelService.commitHotel(booking); return payment1;
            // Transaction committed
        }
        Payment payment1 = rollbackPayment(payment);
        booking.setUserId(payment1.getUserId());
        booking.setHotelId(hotel.getId());
        hotelService.rollbackHotel(booking); return payment1; //
        Transaction Rollback
    }
    Payment payment1 = rollbackPayment(payment);
    booking.setUserId(payment1.getUserId());
    booking.setHotelId(hotel.getId());
    hotelService.rollbackHotel(booking); return payment1; //
    Transaction Rollback
}
@Override public Payment commitPayment(Payment payment) {
    payment.setStatus(PaymentStatus.APPROVED.name()); return
    paymentRepository.save(payment);
}
@Override public Payment rollbackPayment(Payment payment) {
    payment.setStatus(PaymentStatus.ROLLBACK.name()); return
    paymentRepository.save(payment);
}

```

Message Queue

Message queuing facilitates communication between applications by transmitting messages to each other. Acting as temporary storage, the message queue stores messages when the receiving program is occupied or not linked. Comprising a producer, a broker (representing the message queue software), and a consumer, a message queue establishes the framework for communication. This setup offers asynchronous communication capabilities between applications.



Dependency

```
<dependency>
<groupId>org.springframework.amqp</groupId>
<artifactId>spring-rabbit-test</artifactId>
<scope>test</scope>
</dependency>
```

Configuration (Application.yml)

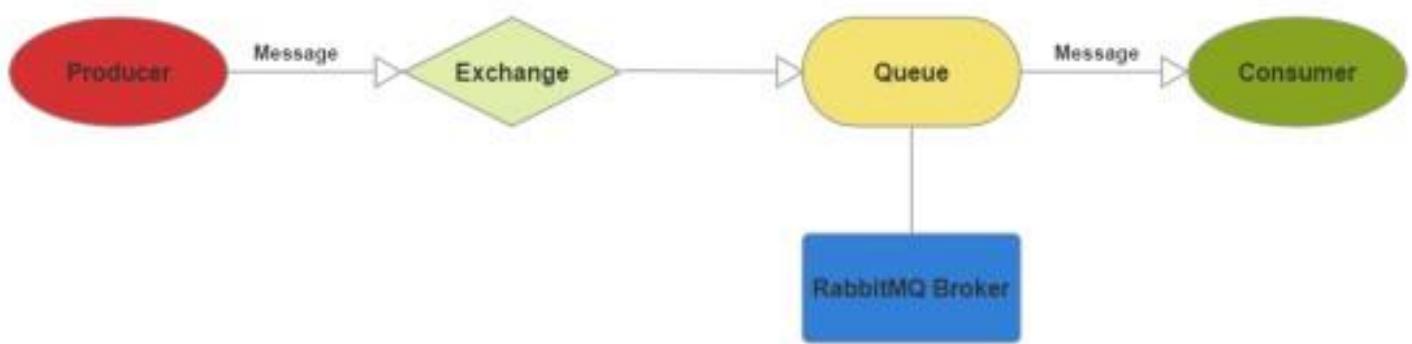
```
spring: rabbitmq: host:
localhost port: 5672
username: guest
password: guest
rabbitmq: rating: queue:
name: rating_queue
exchange:
name: rating_exchange routing:
key: name: rating_routing_key
```

RabbitMQ:

RabbitMQ functions as a message queue software, serving as an intermediary platform where various applications can exchange messages.

Initially designed to adhere to the Advanced Message Queuing Protocol (AMQP), RabbitMQ now extends its support to several other API protocols, including STOMP, MQTT, and HTTP.

RabbitMQ stands out as one of the most straightforward and freely accessible choices for deploying messaging queues.



Producer:

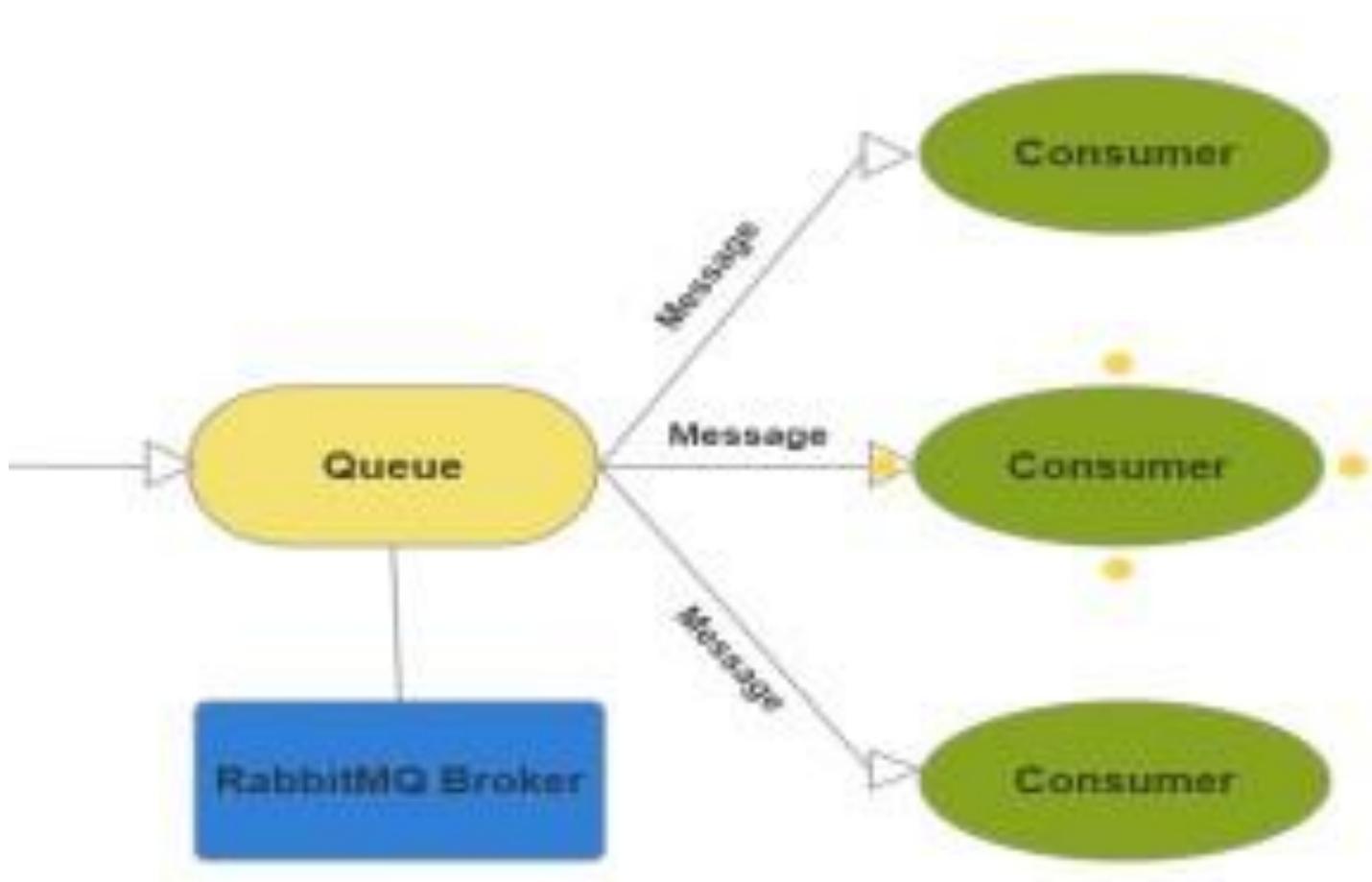
The producer refers to an application responsible for transmitting messages to the RabbitMQ broker, while the consumer denotes an application tasked with retrieving messages from the RabbitMQ broker.

Example:

```
public void sendMessage(String message){  
    LOGGER.info(String.format("Message sent -> %s", message));  
    rabbitTemplate.convertAndSend(ratingExchange, ratingRoutingKey, message);  
}  
  
@Override  
public Rating saveRating(Rating rating) { Rating rating1 =  
ratingRepository.save(rating);  
sendMessage(rating1 == null ? " " : String.valueOf(rating1.getHotelId())); return rating1;  
}
```

Consumer:

The consumer represents an application that retrieves messages from the RabbitMQ broker (Queue). There is a possibility of having multiple consumers within this context.



Example

```

@RabbitListener(queues = {"${rabbitmq.rating.queue.name}"})
public void rabbitMQListener(String message) { Long hotelId =
Long.parseLong(message.trim());
Optional<Hotel> byId = hotelRepository.findById(hotelId); if
(byId.isPresent()) { Hotel hotel = byId.get();
if (hotel.getTotalRating() == null) hotel.setTotalRating(0);
hotel.setTotalRating(hotel.getTotalRating() + 1); hotelRepository.save(hotel);
}
LOGGER.info("Received message from RabbitMQ -> " + message);
}

```

Queue:

A queue within a RabbitMQ broker functions as a buffer or storage space to retain messages. These messages are deposited into the queue by a producer and subsequently retrieved by a consumer. When a message is retrieved, it gets consumed and is consequently removed from the queue. Therefore, each message undergoes processing only once.

Example

```

//spring bean for rabbitmq queue
@Bean public Queue ratingQueue(){
    return new Queue(ratingQueue);
}

```

Message:

Information that is sent from the producer to a consumer through RabbitMQ. Message can be a String, JSON, etc...

Exchange:

Essentially, it operates as an intermediary linking the producer and a queue. Rather than transmitting messages directly to a queue, a producer can send them to an exchange instead.

Subsequently, the exchange forwards these messages to one or multiple queues.

Example:

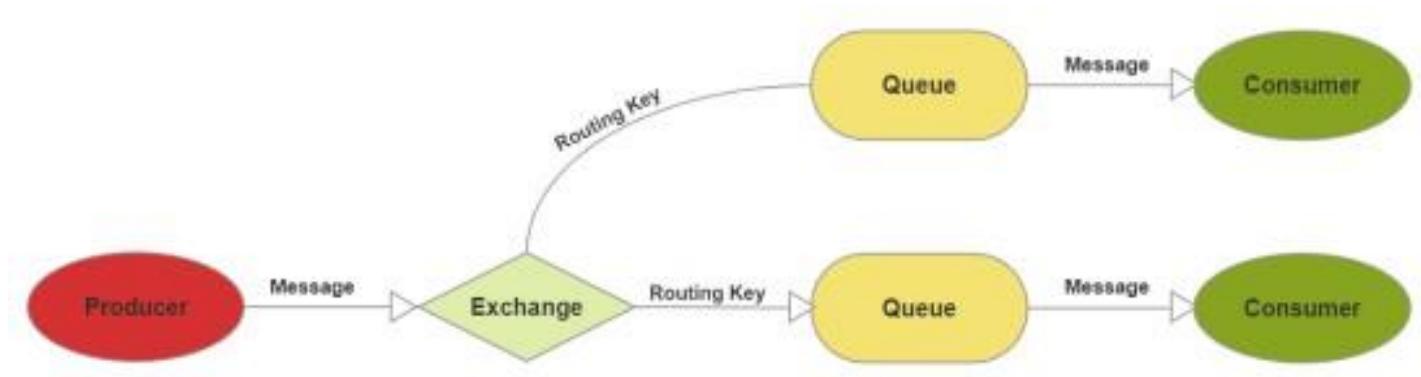
```

//spring bean for rabbitmq exchange
@Bean public TopicExchange ratingExchange(){ return new
TopicExchange(ratingExchange);
}

```

Routing Key:

The routing key serves as a crucial factor that the exchange examines to determine the appropriate routing of messages to respective queues. Comparable to an address, the routing key specifies the destination for the message within the exchange.



Binding:

A binding is a link between a queue and exchange.

RabbitMQ Configuration:

Example

```
// binding between queue and exchange using routing key
@Bean
public Binding ratingBinding(){ return BindingBuilder.bind(ratingQueue())
    .to(ratingExchange())
    .with(ratingRoutingKey);
}
```