# Microservices Interview Questions

## 1) What are microservices?

➔ Microservices is an architectural style.

➔ In Microservice an application is built as a collection of small, independent services that communicate over lightweight protocols like HTTP or messaging queues.

➔ Each service focuses on a specific business capability, ensuring loose coupling and high cohesion.

## 2) What are the key features of microservices architecture?

➔ **Decentralized governance:** Independent teams for development.

➔ **Componentization:** Each service is a component.

➔ **Flexibility in technology:** Services can use different tech stacks.

➔ **Scalability:** Services can scale independently.

➔ **Resilience:** Faults are isolated.

## 3) How do microservices communicate with each other?

**Microservices communicate through:**

➔ **Synchronous communication** : HTTP/REST APIs, gRPC.

➔ **Asynchronous communication** : Message brokers like RabbitMQ, Kafka, or JMS.

## 4) What is Service Discovery in Microservice ?

➔ Service Discovery is the mechanism by which a microservice locates other services it needs to interact with.
➔ Since services often scale dynamically, their network locations (like IP addresses or hostnames) are not fixed.
➔ Instead of hardcoding the location of other services, service discovery enables services to register themselves and discover others **automatically**.

### 🧩 Why is it Needed?

**In a microservice architecture:**

➔ Services are deployed dynamically, often as containers.
➔ They can scale up/down based on load.
➔ They may crash and restart, getting a new IP or hostname.
➔ Hardcoding addresses leads to fragile systems.

**Hence, we need a dynamic and fault-tolerant mechanism for service lookup.**

### 🧱 Key Components of Service Discovery

**1. Service Registry**

- A central database where services register themselves.
- Keeps track of available services and their locations (host:port/IP).
- **Examples:**
  - Eureka (Netflix OSS)
  - Consul

- Zookeeper
- etcd

**2. Service Provider (Registering services)**

- A microservice instance that registers itself with the service registry on startup.
- **It includes:**
  - Service name
  - Instance ID
  - Health check URL
  - Host and Port

  ✅ **Example: A UserService running on 10.0.0.4:8080 registers itself as UserService.**

**3. Service Consumer (Discovering services)**

- A microservice that queries the service registry to find the location of another service it wants to call.
- **It can:**
  - Query the registry directly
  - Use a client library that abstracts service discovery
  - Rely on a service mesh or load balancer

**4. Health Checks**

- The registry performs periodic checks to ensure registered services are alive and healthy.
- If a service fails health checks, it gets deregistered.

## 🔀 Two Types of Service Discovery

### 1. Client-side Discovery

- The **client** is responsible for:
  - Querying the registry
  - Choosing a service instance (load balancing)
  - Making the request directly to that instance

✅ Example: Netflix Eureka with Ribbon or Spring Cloud LoadBalancer

📌 **Pros**: Simple architecture

📌 **Cons**: Tightly couples client with registry logic

### 2. Server-side Discovery

- The **client sends request to a load balancer**, which queries the registry and forwards the request.

✅ Example: Kubernetes with kube-proxy or Envoy as ingress/load balancer

📌 **Pros**: Clients stay clean

📌 **Cons**: Requires external load balancing infrastructure

## 🛠 Common Tools for Service Discovery

| Tool | Type | Notes |
|---|---|---|
| **Eureka** | Client-side | Spring Cloud ecosystem, Netflix OSS |
| **Consul** | Both | Key-value store + health checks |
| **Zookeeper** | Both | Hierarchical store, often used with Kafka |
| **etcd** | Both | Used in Kubernetes (backing store) |
| **Kubernetes DNS** | Server-side | Services are discoverable via DNS |

**Ques) What is the role of API-Gateway ?**

➔ An **API Gateway** is a **server-side application** that acts as an **entry point** into a system built with microservices.
➔ It manages all client request in a centralized way.
➔ Some API gateway are **Spring Cloud Gateway, NGINX**

📌 **Why Do We Need an API Gateway in Microservices?**

**In a microservices architecture:**

- Each microservice is independently deployable and owns its own data.
- Services often expose REST or gRPC APIs.
- A frontend app might need to talk to multiple services to render a single view.

**Without an API Gateway:**

- Clients need to know the location of each microservice.
- Clients have to make multiple network calls to get data.
- Every service must handle authentication, logging, rate-limiting, etc., which leads to code duplication.

An **API Gateway solves these problems** by acting as a **single entry point** that simplifies the interactions between clients and microservices.

## 🧱 Detailed Responsibilities of an API Gateway

### 1. Request Routing

- Routes requests to the appropriate microservice.
- Example:

  **GET /users/123 -> User Service**

  **POST /orders  -> Order Service**

### 2. Request Aggregation / Response Composition

- When a client needs data from multiple services, the gateway can **combine the responses** and return a single unified response.
- ✅ Benefit: Client doesn't make multiple API calls.

**Example:**

- UI needs user profile + order history.
- API Gateway:
  - Calls UserService /users/123
  - Calls OrderService /users/123/orders
  - Merges the data and returns combined JSON.

### 3. Authentication and Authorization

- Handles:

    - Token validation (e.g., JWT)
    - OAuth2 flows
- Prevents unauthorized access to services.

**4. Rate Limiting and Throttling**

- Prevents abuse by limiting how many requests a client can make.
- Example: Max 100 requests per minute per IP.

**5. Load Balancing**

- Distributes traffic to multiple instances of a service.
- Helps in scaling and fault tolerance.

**6. Caching**

- Caches frequent API responses to reduce load and improve latency.
- Example: Product catalog data or config settings.

**7. Logging and Monitoring**

- Logs each request, response time, and errors.
- Integrated with tools like:
    - ELK Stack (Elasticsearch, Logstash, Kibana)
    - Prometheus + Grafana

### 🔁 Real-World Scenario

- **Without API Gateway** : Client → UserService, ProductService, OrderService (3 separate calls)
- **With API Gateway** : Client → API Gateway → internally fans out to services and returns combined response.

## ✅ Benefits Recap

| Benefit | Description |
|---|---|
| 🔐 Centralized Security | Manage auth in one place |
| 🚦 Simplified Client Logic | Only one endpoint to call |
| ⚡ Improved Performance | Caching, load balancing |
| 📊 Better Observability | Unified logging and metrics |

**Ques) How do we handle distributed transaction in Microservice ?**

➔ **Distributed transactions** are one of the more **challenging aspects** of microservices architecture, especially when each service manages its **own database.**

➔ In a **monolith**, you often have a single **ACID transaction** managed by a single DB.

➔ But in **microservices**, different services have **separate databases .**

### ❌ Problem:

- Traditional **ACID transactions** (Atomicity, Consistency, Isolation, Durability) don't work across microservices easily.
- If one service succeeds and another fails, you need to **roll back** or **compensate** — not trivial.
- Also all the services which are involved in distributed transaction must be available to complete the client request.

## ✅ Solutions for Handling Distributed Transactions

### 1. Avoid Distributed Transactions If Possible

- Design services around business boundaries to keep transactions local.
- Use event-driven architecture to decouple services.

### 2. Saga Pattern (Recommended) 🌀

- The **Saga Pattern** breaks a distributed transaction into a **series of local transactions** coordinated via events or commands.

**Types of Sagas:**

#### a) Choreography (Event-based Saga)

- Services **publish and listen to events**.
- No central coordinator.

**Example:**

**OrderService** → creates order → emits OrderCreated

**Payment Service** → listens to OrderCreated → processes payment → emits PaymentCompleted

**Shipping Service** → listens to PaymentCompleted → arranges shipping

If one step fails, the service emits a **compensating event** (e.g., PaymentFailed → triggers CancelOrder)

✅ Pros:

- Decoupled
- Scales well

❌ Cons:

- Harder to track/debug
- Complex flow control

### b) Orchestration (Command-based Saga)

- A **central orchestrator** service manages the flow and decisions.

  **Example:**

  1. OrderService calls SagaOrchestrator → "start order"
  2. SagaOrchestrator calls PaymentService
  3. Based on result, it calls next step or compensates.

  ✅ Pros:

  - Easier to manage flow
  - Centralized logic

  ❌ Cons:

  - Orchestrator becomes a logic bottleneck

Tools: Camunda, Axon Framework, Temporal.io, Netflix Conductor

## Orchestration using Rest

### 🧩 Real-World Example – Order Placement (Saga)

**Scenario**: Placing an order involves:

- Reserving inventory
- Charging payment
- Creating shipping info

**Saga flow:**

1. OrderService creates order → emits OrderCreated
2. InventoryService reserves items → emits InventoryReserved
3. PaymentService charges card → emits PaymentCompleted
4. ShippingService schedules delivery → emits ShippingScheduled

**If PaymentService fails:**

- It emits PaymentFailed
- Other services receive that and run **compensating transactions** (e.g., InventoryService rolls back reservation)

## Saga Flow

text

```
Client -> OrderService -> SagaOrchestrator
SagaOrchestrator -> InventoryService
SagaOrchestrator -> PaymentService
```

Absolutely! Here's a basic example of the **Orchestration Saga Pattern** using **Spring Boot**, with a focus on a simple **Ordering Flow** involving:

1. **Order Service** – Starts the saga and interacts with orchestrator.
2. **Payment Service** – Processes payment.
3. **Inventory Service** – Reserves inventory.
4. **Saga Orchestrator** – Central service that coordinates the steps.

We'll use **REST communication** (simpler to follow) — later, you can replace it with Kafka for an event-driven version.

## 1. Common Models

java Copy Edit

```java
// OrderRequest.java
public class OrderRequest {
    private String orderId;
    private String productId;
    private int quantity;
    private double amount;
}
```

java Copy Edit

```java
// SagaStepResponse.java
public class SagaStepResponse {
    private boolean success;
    private String message;
}
```

## ✅ 2. Order Service

java Copy Edit

```java
@RestController
@RequestMapping("/orders")
public class OrderController {

    @Autowired
    private RestTemplate restTemplate;

    @PostMapping
    public ResponseEntity<String> placeOrder(@RequestBody OrderRequest request) {
        ResponseEntity<String> response = restTemplate.postForEntity(
            "http://localhost:8081/saga/start", request, String.class);
        return response;
    }
}
```

## ✅ 3. Saga Orchestrator

java Copy Edit

```java
@RestController
@RequestMapping("/saga")
public class SagaOrchestratorController {

    @Autowired
    private RestTemplate restTemplate;

    @PostMapping("/start")
    public ResponseEntity<String> startSaga(@RequestBody OrderRequest request) {
        // Step 1: Reserve Inventory
        ResponseEntity<SagaStepResponse> inventoryRes = restTemplate.postForEntity(
            "http://localhost:8082/inventory/reserve", request, SagaStepResponse.class);

        if (!inventoryRes.getBody().isSuccess()) {
            return ResponseEntity.status(400).body("Inventory reservation failed");
        }

        // Step 2: Process Payment
        ResponseEntity<SagaStepResponse> paymentRes = restTemplate.postForEntity(
            "http://localhost:8083/payment/charge", request, SagaStepResponse.class);

        if (!paymentRes.getBody().isSuccess()) {
            // Compensate inventory
            restTemplate.postForEntity(
                "http://localhost:8082/inventory/cancel", request, String.class);
            return ResponseEntity.status(400).body("Payment failed, inventory released");
        }

        return ResponseEntity.ok("Order processed successfully");
    }
}
```

## ✅ 4. Inventory Service

java Copy Edit

```java
@RestController
@RequestMapping("/inventory")
public class InventoryController {

    @PostMapping("/reserve")
    public SagaStepResponse reserve(@RequestBody OrderRequest request) {
        // Dummy logic
        return new SagaStepResponse(true, "Inventory reserved");
    }

    @PostMapping("/cancel")
    public String cancel(@RequestBody OrderRequest request) {
        return "Inventory reservation cancelled";
    }
}
```

## ✅ 5. Payment Service

java Copy Edit

```java
@RestController
@RequestMapping("/payment")
public class PaymentController {

    @PostMapping("/charge")
    public SagaStepResponse charge(@RequestBody OrderRequest request) {
        if (request.getAmount() > 1000) {
            return new SagaStepResponse(false, "Insufficient funds");
        }
        return new SagaStepResponse(true, "Payment successful");
    }
}
```

# Event Driven Saga Orchestration

We'll still use:

- Order Service → initiates the order and sends `OrderCreated` event.
- Saga Orchestrator → listens for `OrderCreated`, coordinates the saga by sending commands and listening for events.
- Inventory Service
- Payment Service

We'll use Apache Kafka topics for communication:

- `order-events`
- `inventory-events`
- `payment-events`
- `saga-events` (optional, for general coordination)

## 🗂️ Project Layout

Each service has its own Spring Boot app:

```
saga-order-service
saga-orchestrator
saga-inventory-service
saga-payment-service
```

## 🧩 Step-by-Step Code Overview

### ✅ 1. Define Common Events (shared across services)

java Copy Edit

```java
public class OrderEvent {
    private String orderId;
    private String productId;
    private int quantity;
    private double amount;
    private String status; // CREATED, INVENTORY_RESERVED, PAYMENT_COMPLETED, etc.
}
```

### ✅ 2. Order Service – Send `OrderCreated` Event

java Copy Edit

```java
@RestController
@RequestMapping("/orders")
public class OrderController {

    @Autowired
    private KafkaTemplate<String, OrderEvent> kafkaTemplate;

    @PostMapping
    public ResponseEntity<String> placeOrder(@RequestBody OrderEvent event) {
        event.setStatus("ORDER_CREATED");
        kafkaTemplate.send("order-events", event);
        return ResponseEntity.ok("Order Created and sent to orchestrator");
    }
}
```

### ✅ 3. Saga Orchestrator – Listens and Sends Commands

java Copy Edit

```java
@Component
public class SagaOrchestrator {

    @Autowired
    private KafkaTemplate<String, OrderEvent> kafkaTemplate;

    @KafkaListener(topics = "order-events", groupId = "saga")
    public void handleOrderCreated(OrderEvent event) {
        if ("ORDER_CREATED".equals(event.getStatus())) {
            // Send to inventory
            event.setStatus("RESERVE_INVENTORY");
            kafkaTemplate.send("inventory-events", event);
        }
    }

    @KafkaListener(topics = "inventory-events", groupId = "saga")
    public void handleInventoryResponse(OrderEvent event) {
        if ("INVENTORY_RESERVED".equals(event.getStatus())) {
            event.setStatus("PROCESS_PAYMENT");
            kafkaTemplate.send("payment-events", event);
        } else if ("INVENTORY_FAILED".equals(event.getStatus())) {
            event.setStatus("ORDER_CANCELLED");
            kafkaTemplate.send("order-events", event);
        }
    }

    @KafkaListener(topics = "payment-events", groupId = "saga")
    public void handlePaymentResponse(OrderEvent event) {
        if ("PAYMENT_COMPLETED".equals(event.getStatus())) {
            event.setStatus("ORDER_COMPLETED");
        } else {
            event.setStatus("ORDER_FAILED");
            // Compensation: Cancel Inventory
            event.setStatus("CANCEL_INVENTORY");
            kafkaTemplate.send("inventory-events", event);
        }
        kafkaTemplate.send("order-events", event);
    }
}
```

## ✅ 4. Inventory Service

java Copy Edit

```java
@KafkaListener(topics = "inventory-events", groupId = "inventory")
public void handleInventory(OrderEvent event) {
    if ("RESERVE_INVENTORY".equals(event.getStatus())) {
        // Business logic (assume success for now)
        event.setStatus("INVENTORY_RESERVED");
    } else if ("CANCEL_INVENTORY".equals(event.getStatus())) {
        // Compensation logic
        System.out.println("Inventory cancelled for " + event.getOrderId());
    }
    kafkaTemplate.send("inventory-events", event);
}
```

## ✅ 5. Payment Service

java Copy Edit

```java
@KafkaListener(topics = "payment-events", groupId = "payment")
public void handlePayment(OrderEvent event) {
    if ("PROCESS_PAYMENT".equals(event.getStatus())) {
        if (event.getAmount() > 1000) {
            event.setStatus("PAYMENT_FAILED");
        } else {
            event.setStatus("PAYMENT_COMPLETED");
        }
        kafkaTemplate.send("payment-events", event);
    }
}
```

## ✅ 6. Kafka Configuration

java Copy Edit

```java
@Configuration
public class KafkaConfig {
    @Bean
    public KafkaTemplate<String, OrderEvent> kafkaTemplate(ProducerFactory<String, OrderEvent
        return new KafkaTemplate<>(pf);
    }

    @Bean
    public ConsumerFactory<String, OrderEvent> consumerFactory() {
        return new DefaultKafkaConsumerFactory<>(Map.of(
            ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092",
            ConsumerConfig.GROUP_ID_CONFIG, "saga",
            ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class,
            ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, JsonDeserializer.class,
            JsonDeserializer.TRUSTED_PACKAGES, "*"
        ));
    }
```

```java
    @Bean
    public ProducerFactory<String, OrderEvent> producerFactory() {
        return new DefaultKafkaProducerFactory<>(Map.of(
            ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092",
            ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class,
            ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, JsonSerializer.class
        ));
    }
}
```

### 🧪 Example Flow

1. Client calls `/orders` in Order Service → sends `ORDER_CREATED`.
2. Orchestrator listens, sends `RESERVE_INVENTORY`.
3. Inventory responds with `INVENTORY_RESERVED`.
4. Orchestrator sends `PROCESS_PAYMENT`.
5. Payment returns either:
   - `PAYMENT_COMPLETED` → `ORDER_COMPLETED`
   - `PAYMENT_FAILED` → `CANCEL_INVENTORY` → `ORDER_FAILED`

**Ques) What is Fault Tolerance ? How do you ensure fault tolerance in microservices ?**

➔ Fault tolerance is the ability of a system to continue functioning even when some of its components fail.

**Instead of crashing entirely, a fault-tolerant microservice system:**

- Handles errors gracefully
- Degrades functionality (gracefully)
- Recovers automatically where possible

## 🧰 How to Ensure Fault Tolerance in Microservices

Here's a breakdown of the key strategies and tools:

### ✅ 1. Retry Mechanism

Automatically retry failed requests a limited number of times.

```java
@Retryable(value = {RemoteServiceException.class}, maxAttempts = 3)
public String callRemoteService() {
   return restTemplate.getForObject("http://inventory-service/check", String.class);
}
```

Use libraries like:

- **Spring Retry**
- **Resilience4j Retry**

## ✅ 2. Circuit Breaker Pattern

➔ Prevents a failing service from being called repeatedly.
➔ If failures cross a threshold, the circuit "opens" to avoid overwhelming the failing service.

🛠 Tools:

- **Resilience4j**
- **Netflix Hystrix (deprecated)**
- **Spring Cloud CircuitBreaker**

java Copy Edit

```java
@CircuitBreaker(name = "inventoryService", fallbackMethod = "fallbackInventory")
public String checkInventory() {
    return restTemplate.getForObject("http://inventory-service/check", String.class);
}
```

## ✅ 3. Timeouts

Always set **timeouts** on all network calls — don't let calls hang indefinitely.

Always set **timeouts** on all network calls — don't let calls hang indefinitely.

properties Copy Edit

```properties
# For RestTemplate
rest.template.read.timeout=2000
rest.template.connect.timeout=2000
```

## ✅ 4. Health Checks + Auto-Healing

➔ Use **readiness/liveness probes** with orchestration tools like **Kubernetes** to detect and recover from failures.

yaml Copy Edit

```yaml
livenessProbe:
  httpGet:
    path: /actuator/health
    port: 8080
  initialDelaySeconds: 15
  periodSeconds: 10
```

## 5. Distributed Tracing & Monitoring

**You can't fix what you can't see. Use observability tools:**

- **Distributed Tracing :** Zipkin, Jaeger, OpenTelemetry
- **Metrics** **:** Prometheus, Micrometer
- **Logging** **:** ELK stack (Elasticsearch, Logstash, Kibana), Fluentd

## 6. Service Discovery and Load Balancing

- Use service discovery (like **Eureka** or **Consul**) to avoid hardcoding endpoints.
- Combine with load balancers (e.g., Spring Cloud LoadBalancer) to avoid routing traffic to unhealthy instances.

**Ques) Explain about Circuit Breaker Design Pattern ?**

- ➔ The Circuit Breaker design pattern is a crucial tool in fault-tolerant microservices.
- ➔ **Circuit Breaker** helps **avoid calling a failing service** repeatedly.
- ➔ It helps prevent cascading failures and allows your system to recover gracefully when a service is failing repeatedly.
- ➔ The **Circuit Breaker** pattern is inspired by electrical circuits.
- ➔ It **monitors calls to a remote service** and **"breaks" the circuit** (i.e., stops calls) if too many failures occur within a time window.

## ⚙️ 2. What is the **Circuit Breaker** Pattern?

**Circuit Breaker** is one of the most popular **design patterns** used to achieve fault tolerance, especially in **distributed systems** or **microservices**.

It prevents repeated calls to a **failing service**, allowing it time to recover, and avoids overloading the system.

> So:
> 🔌 **Circuit Breaker = a tool to help achieve fault tolerance.**

## 🛠️ 3. What is **Resilience4j**?

**Resilience4j** is a **Java library** that provides several fault tolerance patterns — including:

| Feature | Purpose |
|---|---|
| `CircuitBreaker` | Prevent calls to failing services |
| `Retry` | Retry failed calls with backoff |
| `RateLimiter` | Throttle request rate |
| `Bulkhead` | Isolate failures using separate thread pools |
| `TimeLimiter` | Timeout long-running calls |
| `Fallback` | Provide backup behavior on failure |

## 🚦 States of a Circuit Breaker

There are 3 main states:

### 1. **Closed**

- Everything is normal.
- Requests pass through to the service.
- Failures are monitored.

### 2. **Open**

- Too many failures occurred.
- The circuit "breaks".
- No requests are sent to the service.
- A fallback is usually triggered.

### 3. **Half-Open**

- After a cooldown time, the circuit lets a limited number of requests through.
- If they succeed → back to **Closed**.
- If they fail → back to **Open**.

## 🧰 When to Use

Use a circuit breaker when calling remote services (like APIs, databases, or microservices) that might:

- Be temporarily down.
- Be overloaded.
- Have high latency or intermittent failures.

## Real-world Example (Spring Boot with Resilience4j)

java Copy Edit

```java
@CircuitBreaker(name = "productService", fallbackMethod = "fallbackGetProduct")
public Product getProduct(String productId) {
    return restTemplate.getForObject("http://product-service/api/products/" + productId, Produ
}

public Product fallbackGetProduct(String productId, Throwable ex) {
    return new Product(productId, "Fallback Product", 0.0);
}
```

Config (application.yml):

yaml Copy Edit

```yaml
resilience4j.circuitbreaker:
  instances:
    productService:
      failureRateThreshold: 50
      waitDurationInOpenState: 10s
      permittedNumberOfCallsInHalfOpenState: 3
```

**Circuit Breaker Commonly Used With**

- **Retry**
- **Timeouts**
- **Fallbacks**
- **Rate limiters**

## Circuit Breaker with Rate Limiter

### 🧩 Why Combine Circuit Breaker + Rate Limiter?

- **Circuit Breaker** helps **avoid calling a failing service** repeatedly.
- **Rate Limiter** helps **throttle traffic** to prevent overloading your service or downstream services.

Together, they protect your system from:

- Unstable or crashing services (**Circuit Breaker**).
- Traffic spikes or abuse (**Rate Limiter**).

### 💡 Use Case

Imagine you have a microservice that:

- Calls a **payment gateway** (which might go down).
- Is exposed to **high traffic** (maybe bots or bursty clients).

You'd want to:

- Use **Circuit Breaker** to avoid calling the payment service if it's failing.
- Use **Rate Limiter** to cap incoming calls per second to protect the service from overload.

## Code Example

java

Copy Edit

```java
@RateLimiter(name = "paymentRateLimiter")
@CircuitBreaker(name = "paymentService", fallbackMethod = "fallbackPayment")
public String processPayment(String userId) {
    return restTemplate.getForObject("http://payment-service/api/pay/" + userId, String.class
}

public String fallbackPayment(String userId, Throwable ex) {
    return "Payment temporarily unavailable. Please try later.";
}
```

## application.yml Config

yaml

Cop

```yaml
resilience4j:
  circuitbreaker:
    instances:
      paymentService:
        failureRateThreshold: 50
        waitDurationInOpenState: 10s
        permittedNumberOfCallsInHalfOpenState: 3

  ratelimiter:
    instances:
      paymentRateLimiter:
        limitForPeriod: 5       # 5 requests
        limitRefreshPeriod: 1s  # per second
        timeoutDuration: 0      # don't wait if limit exceeded
```

## 🥇 1. Combine Patterns (Layered Defense is Best)

The best approach is usually a **combination of these patterns** for end-to-end resilience:

| Pattern | Purpose |
|---|---|
| 🕯️ **Circuit Breaker** | Avoid repeated calls to failing services |
| 🔁 **Retry (with Backoff)** | Recover from transient failures |
| 🔧 **Fallbacks** | Provide degraded functionality when needed |
| 🧱 **Bulkheads** | Isolate failures so one service doesn't crash others |
| ⏱️ **Timeouts** | Prevent hanging calls |
| 🚦 **Rate Limiter** | Throttle traffic, avoid overload |
| 📨 **Async Messaging (Queue/Kafka)** | Decouple services, handle retries later |
| 🩺 **Health Checks + Auto Healing** | Remove bad instances (via Kubernetes, service mesh) |

## 🧠 Best Practices by Use Case

### 🔄 Synchronous REST Calls

- ✅ Circuit Breaker
- ✅ Timeouts
- ✅ Retry (limited, exponential backoff)
- ✅ Fallback
- ✅ Rate Limiter

### 📨 Asynchronous Communication (e.g., Kafka, RabbitMQ)

- ✅ Retry + Dead Letter Queue (DLQ)
- ✅ Idempotency checks
- ✅ Timeout for response (if waiting)
- ✅ Monitoring

### 🔗 Critical Services (e.g., Payments, Orders)

- ✅ Circuit Breaker + Fallback
- ✅ Isolation (bulkheads)
- ✅ Queue-based buffering (event-driven)

## Ques) What is Docker and Kubernetes and how it is related to microservice ?

### 🐳 What is Docker?

➔ **Docker** helps you **package your app** so it can run **anywhere** without saying "it works on my machine".

**Think of it like:**

A **lunchbox** — you put your food (your app) and everything it needs (like rice, spoon, napkin = libraries/configs) into a single box.
Wherever you take the lunchbox (your friend's home, office, picnic), it works the same.

### ☸️ What is Kubernetes?

**Kubernetes** helps you **run lots of lunchboxes (apps)** across many people (computers). It makes sure:

- They're **running** properly.
- If one **breaks**, it **replaces** it.
- If more people are hungry, it **makes more lunchboxes**.
- It **balances** who gets what (load balancing).
- It **manages updates** so you can roll out new versions smoothly.

### 🎯 In Microservices:

**Let's say you built 3 services:**

- 🛒 OrderService
- 💳 PaymentService
- 📦 InventoryService

Here's what Docker and Kubernetes do:

| Tool | What It Does |
|---|---|
| 🐳 **Docker** | Packs each service with its tools so it runs the same everywhere |
| ☸️ **Kubernetes** | Runs and manages all your services on servers, handles traffic, restarts failures, scales up or down |

### 🔁 Together:

- Docker is **how** you package a service.
- Kubernetes is **where and how** you run and manage those packages.

**Ques) Explain Microservice with JWT token and OAUTH2 Security .**

### 🎯 What is OAuth2 (in simple terms)?

➔ OAuth2 is a way to authorize access.
➔ It doesn't ask for a **username/password every time** — instead, it uses access tokens to prove who you are.
➔ 🔒 In microservices, OAuth2 is often used to protect APIs and make sure only trusted users or services can call them.

### 🔐 OAuth2 Flow in Microservices (Step-by-Step)

**Let's say we have:**

- User using a frontend or client app
- API Gateway (entry point)
- Auth Server (token provider)
- OrderService, PaymentService, etc. (secured microservices)

#### ✅ Step 1: User Logs In

- The user logs in via the frontend (or a mobile app).
- User enters their **username + password**.
- The frontend sends the login details to the Auth Server (like Keycloak, Auth0, or your custom server).

### 🔑 Step 2: Get Access Token

- The **Auth Server** verifies the credentials.
- If successful, it returns an **Access Token** (usually a JWT – JSON Web Token).
- This token is like a **secure key** that proves the user is allowed to access services.
- This token contains:
  - Who the user is
  - What they can access (scopes/roles)
  - Expiry time

### 📦 Step 3: Client Sends Token with Requests

- The client (browser/app) sends this **token** in the **Authorization header**:

```
Authorization: Bearer <access_token>
```

### 🛡️ Step 4: API Gateway or Microservice Validates the Token

- The **API Gateway** or each **microservice** checks if the token:
  - Is valid (not expired, not tampered)
  - Has the right permissions
- They usually do this by verifying the token **signature** (no need to call Auth Server every time if using JWT).

### 🔁 Step 5: Forward the Request

- **If the token is valid:**
  - The Gateway forwards the request to the right microservice (e.g. OrderService)
  - Or the service processes it directly if there's no gateway

### ❌ Step 6: If Token is Invalid

- If token is missing, expired, or wrong:
  - Return **401 Unauthorized** or **403 Forbidden**

# Code To Understand Spring Security with Oauth2

## 1. Set Up Keycloak as Authorization Server

You can either run Keycloak locally via Docker:

bash Copy Edit

```
docker run -p 8080:8080 \
  -e KEYCLOAK_ADMIN=admin \
  -e KEYCLOAK_ADMIN_PASSWORD=admin \
  quay.io/keycloak/keycloak:24.0.2 \
  start-dev
```

Then:

1. Go to `http://localhost:8080`
2. Login with `admin/admin`
3. Create a **Realm** (e.g. `microservice-demo`)
4. Create a **Client** (e.g. `order-service`) with:
   - Access Type: `confidential`
   - Authorization Enabled: ✅
   - Valid redirect URIs: `*`
5. Create a **User** (e.g. `testuser`) with a password
6. Assign a role like `user` and map it to the token

## 2. Spring Boot Microservice (OrderService) Code

`pom.xml`

xml Copy Edit

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

`application.yml`

yaml Copy Edit

```
server:
  port: 8081

spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8080/realms/microservice-demo
```

**issuer-uri:** This is the URL of your OAuth2 provider's realm (in Keycloak).

### 🔐 SecurityConfig.java

java Copy Edit

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
          .authorizeHttpRequests(auth -> auth
              .requestMatchers("/orders/public").permitAll()
              .anyRequest().authenticated()
          )
          .oauth2ResourceServer(oauth2 -> oauth2.jwt());

        return http.build();
    }
}
```

### 🎯 OrderController.java

java Copy Edit

```java
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import java.security.Principal;

@RestController
public class OrderController {

    @GetMapping("/orders")
    public String getOrders(Principal principal) {
        return "Orders for user: " + principal.getName();
    }

    @GetMapping("/orders/public")
    public String publicEndpoint() {
        return "This is a public endpoint";
    }
}
```

## 🧪 Testing the Flow

1. Get a token using Keycloak:

```bash
curl -X POST http://localhost:8080/realms/microservice-demo/protocol/openid-connect/token \
  -d "client_id=order-service" \
  -d "username=testuser" \
  -d "password=123456" \
  -d "grant_type=password" \
  -d "client_secret=<client-secret>" \
  -H "Content-Type: application/x-www-form-urlencoded"
```

2. Use the token to call the protected endpoint:

```bash
curl http://localhost:8081/orders \
  -H "Authorization: Bearer <access_token>"
```

## 🔍 1. **Roles vs Scopes (Quick Clarification)**

| Concept | Description |
|---|---|
| 🔐 **Role** | Tied to the user (e.g. `ADMIN`, `USER`) — controls access to resources |
| 🎯 **Scope** | Defines what the token can do (e.g. `read`, `write`, `order.create`) — more fine-grained |

Both can be embedded inside the **JWT token** issued by the OAuth2 server.

## ✅ 2. **JWT Token Contains Roles / Scopes**

A decoded JWT might look like:

```json
{
  "preferred_username": "john",
  "realm_access": {
    "roles": ["USER", "ADMIN"]
  },
  "scope": "read write order.create"
}
```

## 🛡️ 3. **Spring Boot: Validating Roles**

### 🔐 Using `@PreAuthorize` with `hasRole` or `hasAuthority`

In Spring, roles are mapped as `ROLE_<NAME>`, so `ADMIN` becomes `ROLE_ADMIN`.

### Example:

java Copy Edit

```java
import org.springframework.security.access.prepost.PreAuthorize;

@RestController
public class OrderController {

    @GetMapping("/admin/orders")
    @PreAuthorize("hasRole('ADMIN')")
    public String adminOnly() {
        return "Only admins can see this";
    }

    @GetMapping("/user/orders")
    @PreAuthorize("hasAnyRole('USER', 'ADMIN')")
    public String userAndAdmin() {
        return "User or Admin can see this";
    }
}
```

## 🔧 5. **How Does Spring Know Roles or Scopes?**

Spring Security **reads JWT claims** like `realm_access.roles` or `scope`, and maps them to **granted authorities**.

If you're using Keycloak, Spring may need a custom converter because Keycloak puts roles inside `realm_access`.

### Add Custom JWT Role Converter (for Keycloak):

java Copy Edit

```java
public class KeycloakRoleConverter implements Converter<Jwt, Collection<GrantedAuthority>> {

    @Override
    public Collection<GrantedAuthority> convert(Jwt jwt) {
        Collection<GrantedAuthority> authorities = new ArrayList<>();

        Map<String, Object> realmAccess = jwt.getClaim("realm_access");
        if (realmAccess != null && realmAccess.containsKey("roles")) {
            List<String> roles = (List<String>) realmAccess.get("roles");
            roles.forEach(role -> authorities.add(new SimpleGrantedAuthority("ROLE_" + role));
        }

        return authorities;
    }
```

java

Copy Edit

```java
@Configuration
@EnableMethodSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
          .authorizeHttpRequests(auth -> auth
              .requestMatchers("/public/**").permitAll()
              .anyRequest().authenticated()
          )
          .oauth2ResourceServer(oauth2 -> oauth2
              .jwt(jwt -> jwt.jwtAuthenticationConverter(jwtAuthenticationConverter()))
          );

        return http.build();
    }

    @Bean
    public JwtAuthenticationConverter jwtAuthenticationConverter() {
        JwtAuthenticationConverter converter = new JwtAuthenticationConverter();
        converter.setJwtGrantedAuthoritiesConverter(new KeycloakRoleConverter());
        return converter;
    }
```

## Ques) Spring Security With Single Sign On ( SSO)

### 🧠 What is SSO?

**Single Sign-On means:**

**Login once** → Access multiple apps/services without logging in again.

**Think : Google login** → Access Gmail, Drive, YouTube… all without re-entering your password.

### ✅ Common SSO Setup with Spring Security

You usually use Spring Security **with OAuth2 / OpenID Connect** and an **Identity Provider (IdP)** like:

- **Keycloak**
- **Okta**
- **Auth0**
- **Azure AD**
- **Google**

### 🔐 Key Components in SSO

| Component | Description |
|---|---|
| **Client App** | Your Spring Boot app (or multiple apps) |
| **Authorization Server (IdP)** | Keycloak/Auth0 that handles login + tokens |
| **Resource Servers** | Other microservices that validate access tokens |

## 👣 Step-by-Step: Spring Boot SSO (Login via OAuth2)

### 🧩 1. Add Dependencies in `pom.xml`

xml — Copy — Edit

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

### ⚙️ 2. Configure OAuth2 Client in `application.yml`

yaml — Copy — Edit

```yaml
spring:
  security:
    oauth2:
      client:
        registration:
          keycloak:
            client-id: sso-client
            client-secret: secret123
            scope: openid, profile, email
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code/{registrationId}"
        provider:
          keycloak:
            issuer-uri: http://localhost:8080/realms/myrealm
```

- Replace with your **Keycloak realm**, **client ID**, and **secret**.
- Spring Boot automatically creates the login page and handles the OAuth2 login flow.

### 🪄 3. Secure Controllers with `@AuthenticationPrincipal`

java — Copy — Edit

```java
@Controller
public class HomeController {

    @GetMapping("/")
    public String home(@AuthenticationPrincipal OidcUser user, Model model) {
        model.addAttribute("name", user.getFullName());
        return "home"; // Show logged-in info
    }
}
```

### 4. Customize Security (Optional)

java Copy Edit

```java
@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
          .authorizeHttpRequests(auth -> auth
              .requestMatchers("/public/**").permitAll()
              .anyRequest().authenticated()
          )
          .oauth2Login(); // Enables SSO login via OAuth2

        return http.build();
    }
}
```

## 5. Login Flow with SSO

1. User visits your app
2. Spring redirects to Keycloak login
3. User logs in once
4. Gets redirected back with token
5. All other services using the same IdP accept the same login

**Ques) Set the microservice using Keycloak and Spring cloud gateway platform**

### 🔐 Objective

- ✅ Login once via Keycloak (SSO)
- ✅ Token gets stored (session or cookie or frontend)
- ✅ Every request goes through Gateway
- ✅ Gateway forwards the JWT to downstream microservices
- ✅ Microservices validate the token, no login needed again

### 🧩 Components Setup

We'll configure:

1. Keycloak (Identity Provider)
2. Spring Cloud Gateway (OAuth2 Client + Proxy)
3. Microservices (Resource Servers using JWT validation)

## 🔧 1. Keycloak Setup

1. **Start Keycloak (dev mode)**

bash Copy Edit

```
docker run -p 8081:8080 \
-e KEYCLOAK_ADMIN=admin \
-e KEYCLOAK_ADMIN_PASSWORD=admin \
quay.io/keycloak/keycloak:24.0.1 start-dev
```

2. Open Keycloak UI: http://localhost:8081
3. Create a realm: `sso-realm`
4. Create client: `gateway-client`
   - Access type: `confidential`
   - Enable: `Standard Flow`, `Authorization`
   - Redirect URI: `http://localhost:8080/login/oauth2/code/*`
   - Add `http://localhost:8080` to Web Origins
5. Create user, assign roles like `ROLE_USER`, `ROLE_ADMIN`

## 2. Spring Cloud Gateway Configuration

`pom.xml`

xml Copy Edit

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
```

`application.yml`

yaml Copy Edit

```yaml
server:
  port: 8080

spring:
  security:
    oauth2:
      client:
        registration:
          keycloak:
            client-id: gateway-client
            client-secret: secret123
            scope: openid, profile
            provider: keycloak
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code/{registrationId}"

        provider:
          keycloak:
            issuer-uri: http://localhost:8081/realms/sso-realm

cloud:
  gateway:
    routes:
      - id: service-a
        uri: http://localhost:8082/
        predicates:
          - Path=/service-a/**
        filters:
          - StripPrefix=1
      - id: service-b
        uri: http://localhost:8083/
        predicates:
          - Path=/service-b/**
        filters:
          - StripPrefix=1
```

## 🔒 3. Microservice A/B Setup

`pom.xml`

xml Copy Edit

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```

`application.yml`

yaml Copy Edit

```yaml
server:
  port: 8082

spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8081/realms/sso-realm
```

## 🧪 4. Test the Flow

1. Hit: `http://localhost:8080/service-a/hello`
2. You'll be redirected to Keycloak
3. Login once ✅
4. You'll be redirected back
5. Gateway forwards request with JWT
6. Service A/B validates the JWT automatically

`SecurityConfig.java`

java Copy Edit

```java
@Configuration
public class SecurityConfig {
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
          .authorizeHttpRequests(auth -> auth
              .anyRequest().authenticated()
          )
          .oauth2ResourceServer(oauth2 -> oauth2.jwt());
        return http.build();
    }
}
```

`RestController.java`

java Copy Edit

```java
@RestController
@RequestMapping("/hello")
public class HelloController {

    @GetMapping
    public String hello(@AuthenticationPrincipal Jwt jwt) {
        return "Hello from Service A - user: " + jwt.getClaim("preferred_username");
    }
}
```

**Ques) Explain Multi-Factor Authentication in microservice ?**

- ➔ MFA is when a system requires users to prove their identity with more than just a username and password .
- ➔ This helps make sure that even if someone steals a password, they can't easily get into the system.
- ➔ It's a way to add an extra layer of protection to sensitive services.

**1. User Authentication (First Step)**

- The user logs in by entering their **username and password**.
- This login request goes to a **central authentication service** (a microservice that handles authentication).

**2. MFA Process (Second Step)**

- Once the user's credentials are validated, the authentication service checks if **MFA is enabled** for the user.
  - If **MFA is enabled**, the service asks the user for the **second factor** (for example, a code from an app like Google Authenticator or a code sent via SMS).
  - If **MFA is not enabled**, the system just lets the user in after password verification.

**3. Verification of Second Factor**

- After the user submits the second factor (e.g., the code from their phone), the authentication service verifies that the code is correct.
- If the code is valid, the authentication service sends a token (a secure identifier) back to the user. This token proves that the user has successfully logged in.

**4. Accessing Other Microservices**

- Now, when the user tries to access other parts of the application (other microservices), the system checks that the **token** is valid. The token is usually passed in an HTTP header to each service.
- Microservices verify that the token has come from a trusted authentication service, ensuring that the user has been authenticated and passed MFA.

**Key Points in a Microservice Architecture:**

- ➔ **Authentication service** : A microservice dedicated to handling login and MFA.
- ➔ **Token-based access** : Once MFA is successful, the system issues a token (like a JWT - JSON Web Token) to the user. This token is used to access other microservices securely.
- ➔ **Security across services** : Each microservice checks the token to ensure the user has been authenticated and authorized to perform actions.

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.11.5</version>
</dependency>
<dependency>
    <groupId>com.google.zxing</groupId>
    <artifactId>core</artifactId>
    <version>2.0</version>
</dependency>
<dependency>
    <groupId>com.google.zxing</groupId>
    <artifactId>javase</artifactId>
    <version>2.0</version>
</dependency>
```

### Step 2: Authentication Service with JWT and MFA

**User Model** (`User.java`)

This class represents the user, with fields for MFA secret and token storage.

java Copy

```java
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class User {
    @Id
    private String username;
    private String password;
    private boolean mfaEnabled;
    private String mfaSecret;

    // Getters and Setters
}
```

**JWT Utility Class** ( `JwtUtil.java` )

This utility class will be used to generate and validate JWT tokens.

java Copy

```java
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import java.util.Date;

public class JwtUtil {
    private static final String SECRET_KEY = "supersecretkey";  // Use environment variables in re

    public static String generateToken(String username) {
        return Jwts.builder()
                .setSubject(username)
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis() + 3600000)) // 1 hour expiratio
                .signWith(SignatureAlgorithm.HS256, SECRET_KEY)
                .compact();
    }

    public static boolean validateToken(String token) {
        try {
            Jwts.parser().setSigningKey(SECRET_KEY).parseClaimsJws(token);
            return true;
        } catch (Exception e) {
            return false;
        }
    }

    public static String extractUsername(String token) {
        return Jwts.parser().setSigningKey(SECRET_KEY).parseClaimsJws(token).getBody().getSubject(
    }
}
```

**Authentication Service** ( `AuthService.java` )

This service handles login, MFA verification, and JWT token generation.

java Copy

```java
import com.google.common.io.BaseEncoding;
import io.jsonwebtoken.Jwts;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.security.SecureRandom;

@Service
public class AuthService {
    @Autowired
    private UserRepository userRepository;

    public String login(String username, String password) {
        User user = userRepository.findByUsername(username);

        if (user == null || !user.getPassword().equals(password)) {
            throw new RuntimeException("Invalid credentials");
        }

        if (user.isMfaEnabled()) {
            return user.getMfaSecret();  // Send the MFA secret to the user for scanni
        }

        // No MFA needed, generate JWT token.
        return JwtUtil.generateToken(username);
    }

    public String verifyMfa(String username, String mfaCode) {
        User user = userRepository.findByUsername(username);

        if (user == null) {
            throw new RuntimeException("User not found");
        }

        // Verify the TOTP code using the user's secret
        if (!verifyMfaCode(user.getMfaSecret(), mfaCode)) {
            throw new RuntimeException("Invalid MFA code");
        }

        // Generate JWT if MFA is verified
        return JwtUtil.generateToken(username);
    }
```

```java
@RestController
@RequestMapping("/auth")
public class AuthController {
    @Autowired
    private AuthService authService;

    @PostMapping("/login")
    public String login(@RequestBody LoginRequest request) {
        return authService.login(request.getUsername(), request.getPassword());
    }

    @PostMapping("/verify-mfa")
    public String verifyMfa(@RequestBody MfaRequest request) {
        return authService.verifyMfa(request.getUsername(), request.getMfaCode());
    }
}
```

**DTOs for Login and MFA requests:**

java

```java
public class LoginRequest {
    private String username;
    private String password;

    // Getters and setters
}

public class MfaRequest {
    private String username;
    private String mfaCode;

    // Getters and setters
}
```

**Ques) How do you monitor microservices?**

➔ **Tools:** Prometheus, Grafana, ELK stack, Jaeger, Zipkin.

➔ **Metrics:** CPU, memory, request latency, error rates.

**Ques) What are idempotent operations in microservices?**

➔ Idempotency is a super important concept in microservices—especially when dealing with **APIs**, **network calls**, and **distributed systems**.

➔ Idempotency means that no matter how many times you perform the same action, the result will be the same.

💡 **Why is it important in Microservices?**

In a microservice architecture:

- Services often **communicate over a network**, which is **unreliable**.
- Network calls may **fail**, **timeout**, or **get retried** automatically.
- If a request gets **repeated**, you **don't want it to cause duplicate actions**.

Imagine this:

🛒 **You place an order for a phone.**

The network is slow, so your system retries the order request.

⚠️ Without idempotency → you might get **charged twice**, or **get two phones** delivered.

✅ With idempotency → the system knows it's the **same order**, and only processes it **once**.

## 📦 Where do we use it?

➔ Order placement (E-commerce)

➔ Payment processing (Banking)

➔ Email sending (Avoid duplicate emails)

➔ User creation/update (Avoid duplicate users)

## 🔧 How do we implement it?

**Idempotency key:**

The client (frontend or another service) sends a unique key with the request.
The server checks if it's seen that key before.

➔ If **yes** → ignore or return the stored response.
➔ If **No** → process and store the response.

🧠 **In short:**

| Reason to Use | Explanation |
|---|---|
| Prevent duplicates | Avoid charging or processing something more than once |
| Ensure safety | Especially when retrying failed requests |
| Improve reliability | Clients can safely retry operations |
| Maintain consistency | Especially across services in distributed systems |

## Step By Step Changes

### 💡 Why Should the UI Send the Key?

**The backend doesn't know if a request is a retry or a duplicate—only the frontend (UI) or API consumer knows that. So it's the client's responsibility to:**

1. **Generate a unique key for each user-initiated action (like clicking "Place Order").**
2. **Resend the same key if the action is retried (e.g., after a network timeout or error).**

## ✅ How the UI Should Handle It

### 1. When to Generate the Key?

➔ When the user clicks a button (e.g., "Place Order"), before making the API call.

### 2. How to Generate the Key?

➔ Use a UUID (Universally Unique Identifier) or something similar.

javascript Copy Edit

```javascript
import { v4 as uuidv4 } from 'uuid';

const idempotencyKey = uuidv4(); // Example: "6f25e944-29ab-4f2c-a601-ffbfe8f4c234"
```

Install with:

bash Copy Edit

```bash
npm install uuid
```

### 3. How to Store It Temporarily?

Store it **in memory**, or in local/session storage if retries happen on reload.

javascript Copy Edit

```javascript
// Optionally store in sessionStorage to retry across page reloads
sessionStorage.setItem("order-key", idempotencyKey);
```

### 4. How to Send It with the API Request?

Use the `Idempotency-Key` header in the request.

Example with `fetch`:

javascript Copy

```javascript
fetch("http://localhost:8080/orders", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
    "Idempotency-Key": idempotencyKey,
  },
  body: JSON.stringify({
    product: "Phone",
    quantity: 1
  }),
});
```

Or using `axios`:

javascript Copy Edit

```javascript
import axios from 'axios';

axios.post("http://localhost:8080/orders", {
  product: "Phone",
  quantity: 1
}, {
  headers: {
    "Idempotency-Key": idempotencyKey
  }
});
```

### 💬 How to Handle Retries?

If the UI gets a **network error or timeout**, it should **retry the same request** using the **same idempotency key**:

- If it uses a **new key**, the backend will think it's a new order.
- If it uses the **same key**, the backend will return the previously processed response.

### 🛡️ Bonus: Prevent Duplicate Submissions in UI

Also make the button temporarily disabled to avoid double clicks:

```javascript
<button disabled={isSubmitting}>Place Order</button>
```

## Backend Implementation

➔ A client sends a POST request to place an order.

➔ The request includes an Idempotency-Key in the header.

➔ The server stores processed keys to avoid handling the same request multiple times.

### Flow:

1. Client sends `POST /orders` with header `Idempotency-Key: abc123`.
2. Server checks if the key `abc123` is already used.
   - If yes → return the saved response.
   - If no → process the order, save the key and return the response.

### Sample Code:

**1. Order Entity**

```java
@Entity
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String product;
    private int quantity;

    // Constructors, getters, setters
}
```

**2. IdempotencyKey Entity**

```java
@Entity
public class IdempotencyKey {
    @Id
    private String key; // The client-provided idempotency key

    private Long orderId; // Link to the order created

    // Constructors, getters, setters
}
```

4. **Order Controller**

java

Copy Edit

```java
@RestController
@RequestMapping("/orders")
public class OrderController {

    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private IdempotencyKeyRepository keyRepository;

    @PostMapping
    public ResponseEntity<?> placeOrder(
        @RequestBody Order order,
        @RequestHeader(value = "Idempotency-Key", required = true) String idempotencyKey
    ) {
        // Check if the key already exists
        Optional<IdempotencyKey> existingKey = keyRepository.findById(idempotencyKey);
        if (existingKey.isPresent()) {
            Long orderId = existingKey.get().getOrderId();
            Order existingOrder = orderRepository.findById(orderId).orElse(null);
            return ResponseEntity.ok(existingOrder);
        }

        // Save the order
        Order savedOrder = orderRepository.save(order);

        // Save the idempotency key with reference to the order
        IdempotencyKey key = new IdempotencyKey();
        key.setKey(idempotencyKey);
        key.setOrderId(savedOrder.getId());
        keyRepository.save(key);

        return ResponseEntity.ok(savedOrder);
    }
}
```

**Ques) How do you manage configuration in microservices?**

- **Externalized configurations** : Using tools like Spring Cloud Config or Consul.
- **Environment-specific settings** : Separate configurations per environment.

**Ques) What is a distributed log aggregator, and why is it used?**

➔ A distributed log aggregator collects logs from all microservices.

➔ Tools like the **ELK stack** and **Fluentd** are used for centralized logging and troubleshooting.

# What is difference between SAGA and 2 Phase Commit? Which one will you prefer and why?

| **2 Phase Commit** | **Saga Pattern** |
|---|---|
| Strong Consistency | Eventual Consistency |
| Synchronous | Asynchronous |
| Blocking | Non Blocking |
| Complexity | Simplicity |
| Performance Overhead | Better Performance |
| Strong Data Consistency | Temporarily Inconsistency |

**Ques) How many to ways to communicate b/w microservice ?**

## Ways to communicate between Microservices

- We have seen Synchronous communications through -
  - Rest APIs
  - GraphQl
  - Feign using Eureka discoveries
  - GRPC ( 10 times faster than REST APIs ) - developed by Google as substitute of REST with many more features.

- A synchronous call means that a service waits for the response after performing a request.

- Today we will look at ways to do asynchronous communication in java. This communication usually involves some kind of messaging system like
  - Active Mqs
  - Rabbit MQs
  - Kafka

**Ques) What If message broker is down ?**

## What if the message broker is down?

- A message broker is a vital part of the asynchronous architecture and hence must be fault tolerant
- This can be achieved by setting up additional standby replicas that can do failover. Still, even with auxiliary replicas, failures of the messaging system might happen from time to time.
- If it's essential to ensure the message arrives at its destination, a broker might be configured to work in at-least-once mode. After the message reaches the consumer, it needs to send back ACK to the broker. If no acknowledgement gets to the broker, it will retry the delivery after some time.

## Ques) What is Point to Point Async Communication vs Publisher Subscriber Model ?

### What is PTP Async communication

- PTP - A queue will be used for this type of messaging-based communication.
- The service that produces the message, which is called as producer (sender), will send the message to a queue in one message broker and the service that has an interest in that message, which is called a consumer (receiver), will consume the message from that queue and carry out further processes for that message
- One message sent by a producer can be consumed by only one receiver and the message will be deleted after consumed.
- If the receiver or an interested service is down, the message will remain persistent in that queue until the receiver is up and consumes the message.
- For this reason, messaging-based communication is one of the best choices to make our microservices resilient.

A popular choice for the queueing system is RabbitMQ, ActiveMQ

### What is Publisher-Subscriber Async communication

- In publisher-subscriber messaging-based communication, the topic in the message broker will be used to store the message sent by the publisher and then subscribers that subscribe to that topic will consume that message
- Unlike point to point pattern, the message will be ready to consume for all subscribers and the topic can have one or more subscribers. The message remains persistent in a topic until we delete it.
- In messaging-based communication, the services that consume messages, either from queue or topic, must know the common message structure that is produced or published by producer or publisher.
- examples are Kafka, Amazon SNS etc

# How SAGA DP handles failure of any individual SAGA?

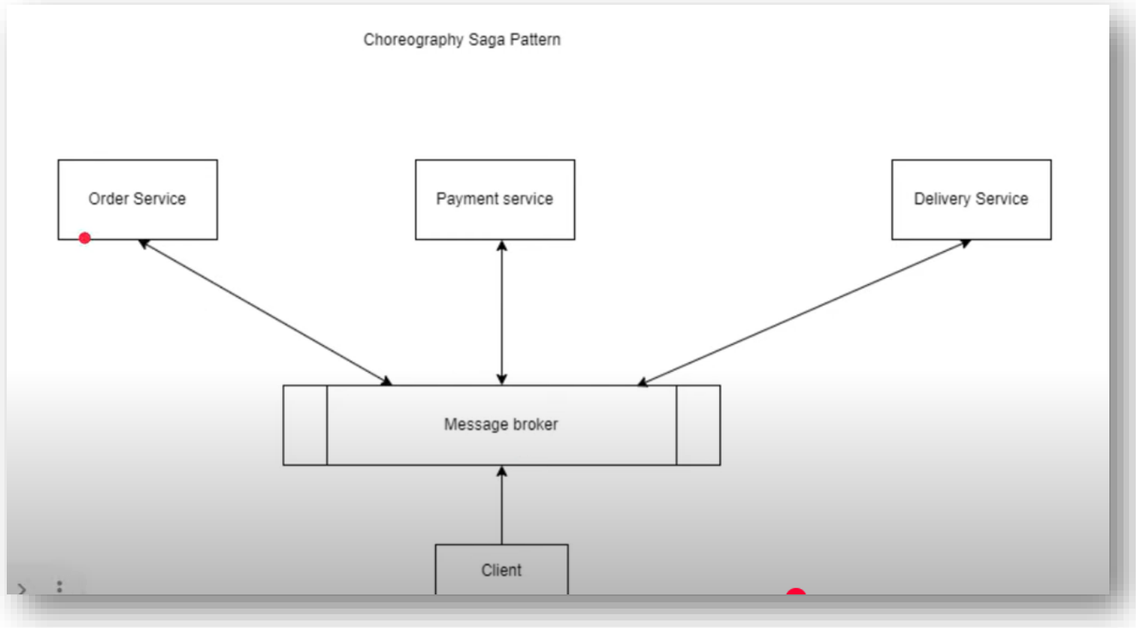
- The saga pattern provides **transaction management** with using a sequence of local transactions of microservices. **Every microservices** has its own database and it can able to **manage local transaction in atomic way** with strict consistency.

- So saga pattern **grouping these local transactions and sequentially i**nvoking one by one. Each local transaction updates the database and **publishes an event to trigger the next local transaction.**

- If one of the step is failed, than saga patterns trigger to **rollback transactions** that are a set of **compensating transactions that rollback the changes on previous microservices** and restore data consistency.

## Ques) Ways to implement Saga ?

- ➔ **Choreography**
- ➔ **Orchestration**




### Choreography Design Pattern

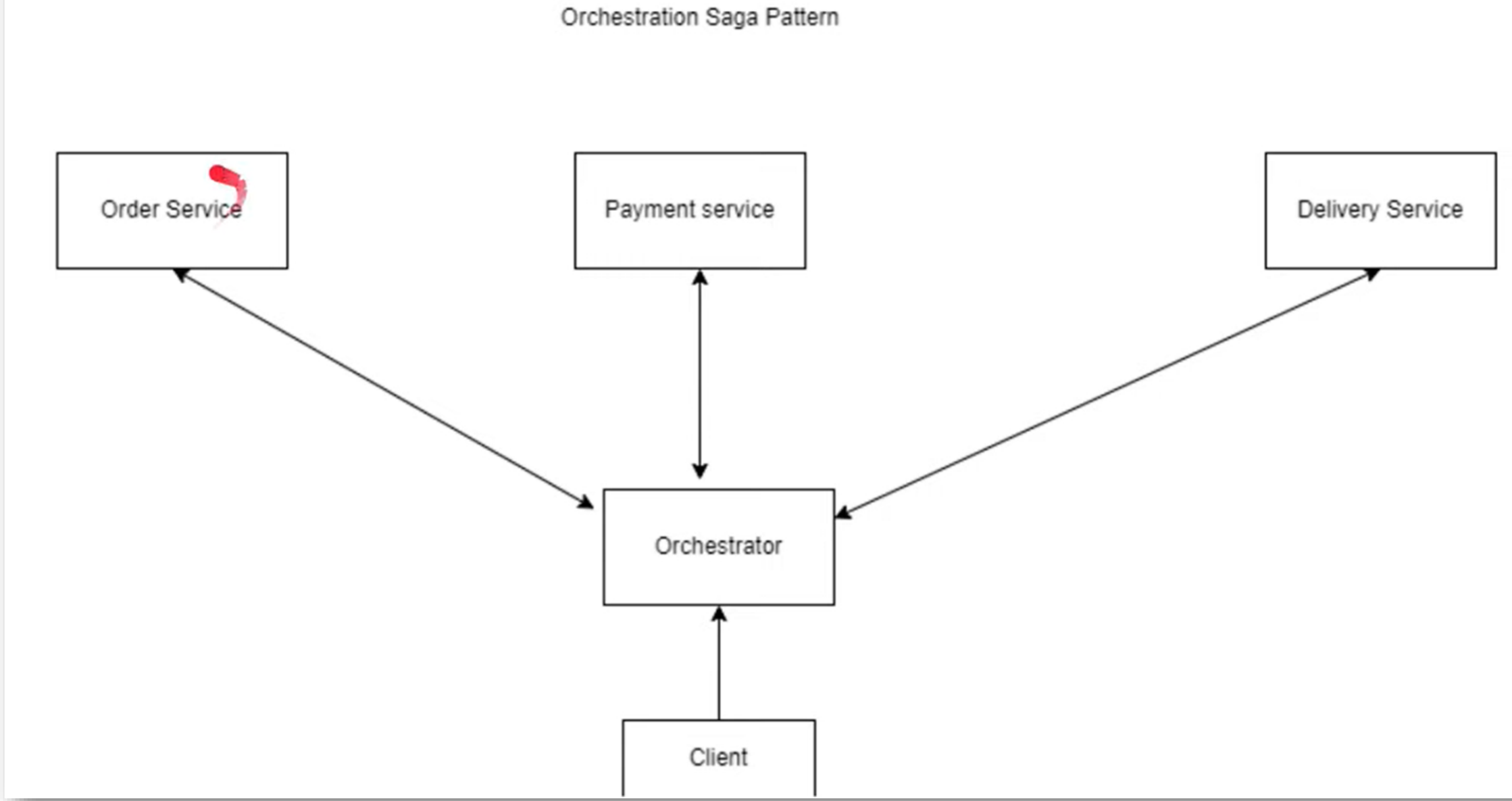
- ➔ It says that all microservices will coordinate with single message broker.
- ➔ Client says to create an order , Saga1 will do the necessary changes in Order Service and release some event for payment service execution.
- ➔ Now broker send information for payment execution , once done payment service will release another event for payment successful to broker and after that broker calls the Third saga to deliver service .

# Disadvantages of Choreography Saga Pattern?

- Workflow can become confusing when adding new steps, as it's difficult to track which saga participants listen to which commands.
- There's a risk of cyclic dependency between saga participants because they have to consume each other's commands
- Integration testing is difficult because all services must be running to simulate a transaction.

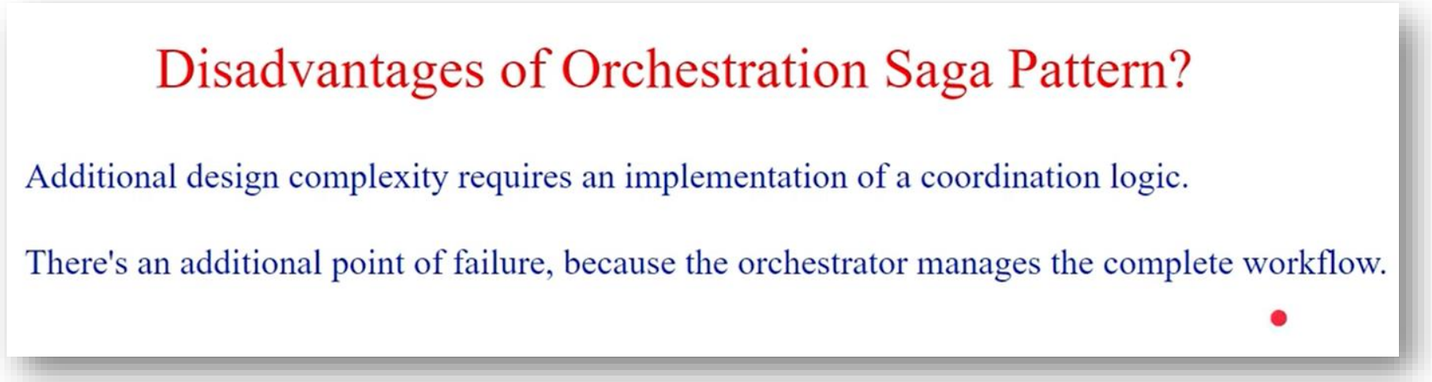
# What is Orchestration Saga Pattern?

- Orchestration is a way to **coordinate sagas where a centralized controller** tells the saga participants what local transactions to execute.
- The saga **orchestrator handles all the transactions** and **tells the participants which operation to perform based on events**.
- The orchestrator
    - executes saga requests,
    - stores and interprets the states of each task,
    - handles failure recovery with compensating transactions.


Orchestration Saga Pattern
Order Service
Payment service
Delivery Service
Orchestrator
Client

# Advantages of Orchestration Saga Pattern?

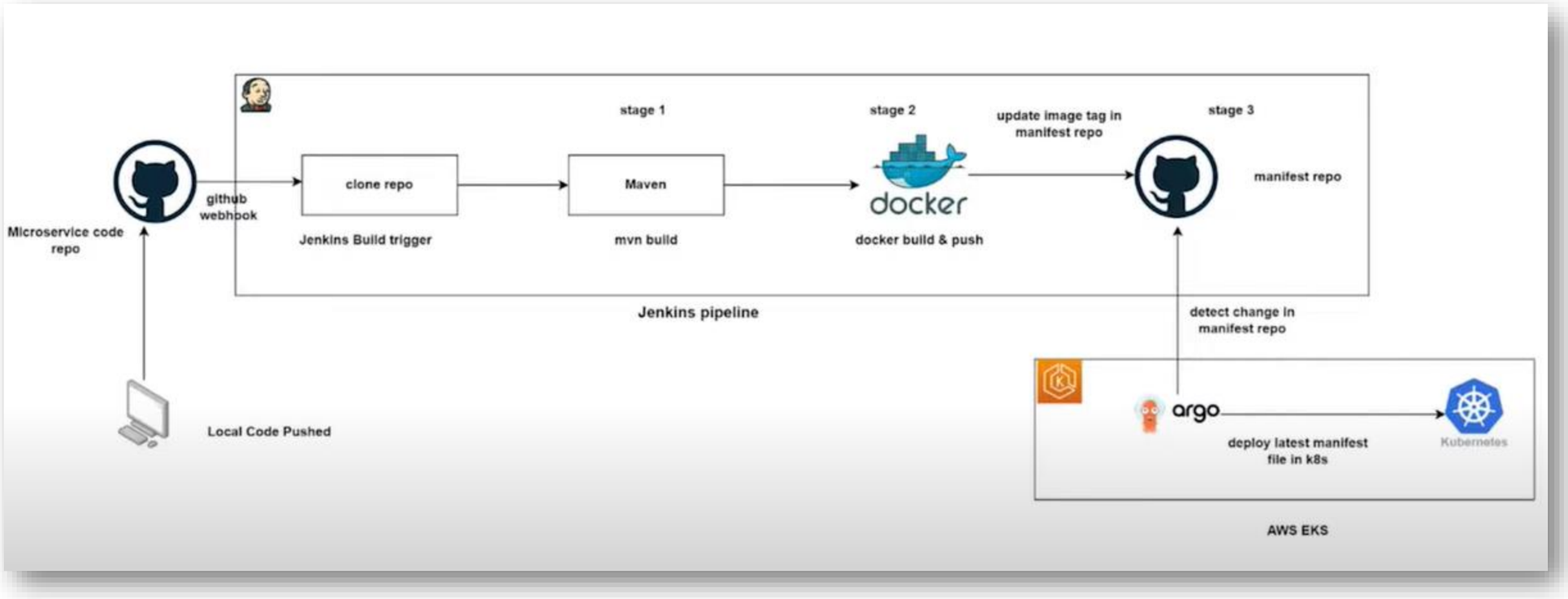
- Good for complex workflows involving many participants or new participants added over time.
- Suitable when there is control over every participant in the process, and control over the flow of activities.
- Doesn't introduce cyclic dependencies, because the orchestrator unilaterally depends on the saga participants.
- Saga participants don't need to know about commands for other participants. Clear separation of concerns simplifies business logic.




**Ques) How to Handle Data Consistency in a Microservices Architecture ?**

🔧 **Key Techniques:**

- 🔄 **Synchronous Communication**
  → Ensure consistency by making direct service-to-service calls, waiting for responses.
- 📩 **Asynchronous Communication**
  → Use message queues or event buses for decoupled communication between services.
- ⚖️ **CQRS (Command Query Responsibility Segregation)**
  → Separate read and write responsibilities to scale and manage consistency efficiently.
- 📜 **Event Sourcing**
  → Persist state changes as a sequence of events, allowing full state reconstruction.
- 🔗 **Distributed Transactions**
  → Maintain consistency across services using protocols like two-phase commit (2PC).
- 🌀 **Saga Pattern**
  → Handle long-running transactions by coordinating a sequence of local transactions with compensation logic for rollbacks.
- 🧠 **Monitoring and Logging**
  → Use observability tools to detect anomalies and ensure system health and data accuracy.


stage 1
stage 2
update image tag in manifest repo
stage 3
Microservice code repo
github webhook
clone repo
Maven
docker
manifest repo
Jenkins Build trigger
mvn build
docker build & push
Jenkins pipeline
detect change in manifest repo
Local Code Pushed
argo
deploy latest manifest file in k8s
Kubernetes
AWS EKS

## 🔧 Ques) Fault Tolerance and Resilience

### 🚨 Challenge

Distributed systems are inherently prone to failures — whether due to:

- 🌐 **Network issues**
- 💥 **Service crashes**
- 🔗 **Dependency failures**

A failure in one service can cascade and bring down the entire system if not properly managed.

---

## ✅ How to Address

### 🛑 Circuit Breaker Pattern

**Implement the Circuit Breaker pattern using libraries like Hystrix or Resilience4j**

🔒 Temporarily blocks failed service calls to prevent cascading failures and allow recovery.

---

### 🔁 Retries and Backoff

**Use retry and backoff mechanisms to handle transient failures.**

⏱️ Implement with proper limits and exponential backoff to avoid overwhelming services.

---

### ⏳ Timeouts and Failover

**Set appropriate timeouts for service calls.**

✔️ Ensure services fail gracefully if they cannot reach other services.

---

### 🧱 Bulkheads

**Use the Bulkhead pattern to isolate failures to a single part of the system.**

🛡️ Prevent one failing service from bringing down the entire system.

---

### 💉 Chaos Engineering

**Adopt Chaos Engineering to simulate failures.**

🧬 Proactively test the resilience of your microservices architecture.

**Ques) You have multiple microservices communicating synchronously via REST. A service in the middle of a request chain fails. How do you handle failure recovery and ensure data consistency?**

➔ If a microservice in the middle of a synchronous REST request chain fails, failure recovery and data consistency can be maintained using various **resilience patterns to prevent cascading failures** and improve system stability.

## ✅ Strategies to Handle This:

| Resilience Pattern | Description |
|---|---|
| Circuit Breaker | Prevents repeated failures by **temporarily halting calls** to the failing service, allowing it time to recover. Libraries: *Hystrix, Resilience4j*. |
| Retry with Backoff | Automatically retries failed requests using **exponential backoff** to avoid overwhelming the service. |
| Timeout Management | Define **timeouts** for REST calls to ensure the system doesn't hang waiting for responses from slow or dead services. |
| Fallback Mechanisms | Provide **fallback responses** (e.g., cached or default data) when a service is unavailable, maintaining a good user experience. |
| Idempotency | Ensure operations are **idempotent** so that retries do not corrupt or duplicate data. |
| Logging & Monitoring | Use tools to **log failures and monitor** services to detect issues early and alert the team. |

## 🧩 Key Recovery Strategies Summary

| 🎯 Scenario | 💡 Solution | 🛠️ Tool/Tech |
|---|---|---|
| **Service fails temporarily** | Retry with backoff | `Spring Retry` |
| **Service is down for a long time** | Circuit breaker | `Resilience4j` |
| **Service is slow or unresponsive** | Timeouts | `RestTemplate` Config |
| **Service fails but data is needed** | Fallback response or cache | `Redis`, `Resilience4j` |
| **Avoid cascading failures** | Stop further calls | Circuit Breaker (`Resilience4j`) |
| **Prevent blocking during issues** | Use asynchronous messaging | `Kafka`, `RabbitMQ` |

## 🔁 What is Transaction Propagation?

**Transaction Propagation** defines how **transactional behavior** should be applied when a **method is executed within an existing transaction context**.

In simpler terms:

**It controls how a method should behave if there's already a running transaction — should it join it, suspend it, start a new one, or throw an error?**

### 🧪 Common Propagation Types in Spring

| 🏷️ Propagation Type | 🔍 Description |
|---|---|
| REQUIRED (default) | Joins the **existing transaction** if one exists; otherwise, **starts a new one**. |
| REQUIRES_NEW | **Suspends** the existing transaction (if any) and **starts a new one**. |
| NESTED | Executes within a **nested transaction** if a current one exists; otherwise, behaves like `REQUIRED`. Useful with savepoints. |
| SUPPORTS | Joins an existing transaction **if present**; if not, **executes non-transactionally**. |
| NOT_SUPPORTED | **Suspends** the current transaction and runs **non-transactionally**. |
| NEVER | Throws an exception if there is an active transaction. Only runs **non-transactionally**. |
| MANDATORY | Must run within an existing transaction; otherwise, throws an exception. |

## 💡 Use Cases

- Use `REQUIRED` for most scenarios.
- Use `REQUIRES_NEW` when a method should **commit independently** of the caller's transaction (e.g., audit logging).
- Use `NESTED` when you want **partial rollback** (e.g., rollback this part but continue outer flow).

### 🧳 Transaction Propagation Types & Use Cases

| 🏷️ Propagation Type | 🔍 What It Does | 💼 Use Case |
|---|---|---|
| REQUIRED *(default)* | Joins the existing transaction; if none, starts a new one. | ✅ **General usage.** Most service methods use this. For example, saving a new customer and their address together. |
| REQUIRES_NEW | Suspends the current transaction and starts a new one. | 📜 **Audit logging or sending emails.** Even if the main transaction fails, audit logs or notifications should be committed independently. |
| NESTED | Runs in a nested transaction. Rolls back only its part without affecting the outer transaction (if supported by DB). | 🔁 **Partial rollback scenarios.** For example, processing a batch of records — if one fails, roll it back but continue others. |
| SUPPORTS | Runs within a transaction if one exists; otherwise, runs without a transaction. | 📖 **Read-only operations.** Fetching data where you don't care about transactional boundaries. |
| NOT_SUPPORTED | Suspends any current transaction and runs without one. | 🌐 **External API calls or caching.** You don't want transactions for non-DB tasks like HTTP calls or setting cache. |
| NEVER | Throws an exception if a transaction exists. | ❌ **Validation-only methods.** Ensure the method runs completely outside of a transaction context. |
| MANDATORY | Requires an existing transaction; throws an exception if none exists. | ⚠️ **Security or integrity checks** that must be part of an active transaction initiated by the caller. |

**Ques) Suppose I have 3 microservices which are running independently and fetch response from some Kafka Topic**

**But I want to update my db after getting response from all these 3 services . How to do ?**

Simulating async calls using `CompletableFuture`.

java Copy Edit

```java
@Service
public class ExternalServiceClient {

    @Async
    public CompletableFuture<ServiceResponse> callServiceA(String orderId) {
        // Simulate remote call
        return CompletableFuture.completedFuture(new ServiceResponse(orderId, "SERVICE_A", true));
    }

    @Async
    public CompletableFuture<ServiceResponse> callServiceB(String orderId) {
        return CompletableFuture.completedFuture(new ServiceResponse(orderId, "SERVICE_B", true));
    }

    @Async
    public CompletableFuture<ServiceResponse> callServiceC(String orderId) {
        return CompletableFuture.completedFuture(new ServiceResponse(orderId, "SERVICE_C", true));
    }
}
```

```java
@Autowired
private ExternalServiceClient client;

@Autowired
private OrderRepository repository;

public void processOrder(String orderId) {
    CompletableFuture<ServiceResponse> serviceA = client.callServiceA(orderId);
    CompletableFuture<ServiceResponse> serviceB = client.callServiceB(orderId);
    CompletableFuture<ServiceResponse> serviceC = client.callServiceC(orderId);

    CompletableFuture<Void> allFutures = CompletableFuture.allOf(serviceA, serviceB, serviceC);

    allFutures.thenAcceptAsync(voidRes -> {
        try {
            ServiceResponse resA = serviceA.get();
            ServiceResponse resB = serviceB.get();
            ServiceResponse resC = serviceC.get();

            if (resA.isSuccess() && resB.isSuccess() && resC.isSuccess()) {
                updateOrder(orderId, "COMPLETED");
            } else {
                updateOrder(orderId, "FAILED");
            }

        } catch (Exception e) {
            updateOrder(orderId, "FAILED");
```