**Saga Pattern:-**

1.It is used in microservices.

2.it is use to distribute transaction into mutiple service

3.in java (RestTemplate , WebClient) used

**Orchestrator** :- it call each services in squence and handling success/failure

**Workflow:-**

[1] Saga Orchestrator starts

↓

[2] Call Order Service → Place Order

↳ success → continue

↳ failure → stop

↓

[3] Call Payment Service → Deduct Payment

↳ success → continue

↳ failure → call Order Cancel

↓

[4] Call Inventory Service → Reduce stock

↳ success → ✅ Saga Completed

↳ failure → call Payment Refund → then call Order Cancel

**Using WebClient:-**

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-webflux</artifactId> <!-- For WebClient -->

</dependency>

</dependencies>

**WebClientConfig.java**

@Configuration

public class WebClientConfig {

@Bean

public WebClient.Builder webClientBuilder() {

return WebClient.builder();

}

}

**SagaService.java**

@Service

public class SagaService {

@Autowired

private WebClient.Builder webClientBuilder;

public Mono<String> startOrderFlow(String orderId) {

return createOrder(orderId)

.flatMap(res1 -> deductPayment(orderId))

.flatMap(res2 -> reduceInventory(orderId))

.doOnSuccess(res3 -> System.out.println("✅ Saga completed successfully"))

.onErrorResume(e -> {

System.out.println("❌ Saga failed: " + e.getMessage());

return rollback(orderId);

});

}

public Mono<String> createOrder(String orderId) {

return webClientBuilder.build()

.post()

.uri("http://localhost:8081/order/create")

.bodyValue(orderId)

.retrieve()

.bodyToMono(String.class);

}

public Mono<String> deductPayment(String orderId) {

return webClientBuilder.build()

.post()

.uri("http://localhost:8082/payment/deduct")

.bodyValue(orderId)

.retrieve()

.bodyToMono(String.class);

}

public Mono<String> reduceInventory(String orderId) {

return webClientBuilder.build()

.post()

.uri("http://localhost:8083/inventory/reduce")

.bodyValue(orderId)

.retrieve()

.bodyToMono(String.class);

}

public Mono<String> rollback(String orderId) {

return refundPayment(orderId)

.then(cancelOrder(orderId))

.then(Mono.just("Compensation completed"));

}

public Mono<String> refundPayment(String orderId) {

return webClientBuilder.build()

.post()

.uri("http://localhost:8082/payment/refund")

.bodyValue(orderId)

.retrieve()

.bodyToMono(String.class);

}

public Mono<String> cancelOrder(String orderId) {

return webClientBuilder.build()

.post()

.uri("http://localhost:8081/order/cancel")

.bodyValue(orderId)

.retrieve()

.bodyToMono(String.class);

}

}

**SagaController.java**

@RestController

@RequestMapping("/saga")

public class SagaController {

@Autowired

private SagaService sagaService;

@PostMapping("/start")

public Mono<String> startSaga(@RequestBody String orderId) {

return sagaService.startOrderFlow(orderId);

}

}

**🔸 Order Service (Port: 8081)**

@RestController

@RequestMapping("/order")

public class OrderController {

@PostMapping("/create")

public String createOrder(@RequestBody String orderId) {

System.out.println("✅ Order created: " + orderId);

return "Order Created";

}

@PostMapping("/cancel")

public String cancelOrder(@RequestBody String orderId) {

System.out.println("↩ Order cancelled: " + orderId);

return "Order Cancelled";

}

}

**🔸 Payment Service (Port: 8082)**

@RestController

@RequestMapping("/payment")

public class PaymentController {

@PostMapping("/deduct")

public String deduct(@RequestBody String orderId) {

System.out.println("💳 Payment deducted for: " + orderId);

return "Payment Deducted";

}

@PostMapping("/refund")

public String refund(@RequestBody String orderId) {

System.out.println("💰 Payment refunded for: " + orderId);

return "Payment Refunded";

}

}

**🔸 Inventory Service (Port: 8083)**

@RestController

@RequestMapping("/inventory")

public class InventoryController {

@PostMapping("/reduce")

public String reduce(@RequestBody String orderId) {

System.out.println("📦 Inventory reduced for: " + orderId);

// Uncomment this line to simulate failure

// throw new RuntimeException("Inventory service failed!");

return "Inventory Reduced";

}

}

**Choreography : -** 1. It is decentralized

2.one service publish event and another will listen it

3. messaging system used(Kafka , rabitmq ,ApacheMQ)

**Workflow of Choreography Saga (Example)**

**Scenario: Order → Payment → Inventory**

[1] **Order Service**

- Creates order

- Publishes event: OrderCreated

↓ (event)

[2] **Payment Service** (listens to OrderCreated)

- Deducts payment

- Publishes event: PaymentCompleted

↓ (event)

[3**] Inventory Service** (listens to PaymentCompleted)

- Reduces stock

- Publishes event: InventoryUpdated

↓

[4] **Notification Service** (optional)

- Sends email/SMS on InventoryUpdated