Microservices Scenario-Based Interview Questions

1. **Your Order service calls the Payment service via Feign client, but sometimes Payment is down. How will you ensure the Order service continues to work without crashing?**

   - ✓ **Use Circuit Breaker**: Implement Resilience4j or Spring Cloud Circuit Breaker with Feign.
   - ✓ **Fallback Methods**: Provide a fallback response to continue Order service operations.
   - ✓ Prevents cascading failures, ensures system resilience.

2. **You need to change the discount rate in your Pricing service at runtime without restarting the service. How would you achieve this?**

   - ✓ **Spring Cloud Config Server**: Centralized configuration.
   - ✓ **@RefreshScope Bean**: Enables runtime refresh.
   - ✓ **Actuator** /refresh endpoint: Refresh properties dynamically.

   So, we achieve, Zero downtime config updates, centralized property management.

3. **You have multiple instances of the Inventory service registered in Eureka. How will you ensure requests from the Product service are evenly distributed?**

   - ✓ Use Ribbon (legacy) or Spring Cloud LoadBalancer (modern).
   - ✓ @LoadBalanced RestTemplate or WebClient**: Ensures requests are distributed.**

4. **You want only premium users to access the /checkout API in your Order service. How will you secure this endpoint?**

   - ✓ **Spring Security with JWT/OAuth2**: Token-based authentication.

- ✓ **Role-based access control**: @PreAuthorize("hasRole('PREMIUM')")

5. **When a user places an order, both Order and Payment services need to update their databases. How would you ensure that either both operations succeed or both fail?**

- ✓ **Avoid 2-phase commit**: Not recommended in microservices.
- ✓ **Use Saga Pattern**:
  - ▪ *Choreography-based*: Each service publishes events; other services react.
  - ▪ *Orchestration-based*: A Saga orchestrator handles transactions and compensations.
- ✓ **Event-driven with Kafka/RabbitMQ**: Ensure eventual consistency.

6. **A user reports that their order is delayed, and you need to trace the request across multiple services (Order, Payment, Notification). How would you implement distributed tracing and logging?**

- ✓ **Spring Cloud Sleuth:** Adds trace IDs to logs.
- ✓ **Zipkin/Jaeger:** Centralized tracing dashboard.
- ✓ **Structured Logging:** Use JSON logs for easier analysis.

7. **During a flash sale, your Order service receives thousands of requests per second. How would you scale your services and prevent downtime?**

- ✓ **Horizontal Scaling:** Kubernetes/Docker Swarm.
- ✓ **Caching:** Redis/Caffeine for frequently accessed data.
- ✓ **Rate Limiting:** Spring Cloud Gateway or API Gateway.
- ✓ **Queue Requests:** Use Kafka to handle spikes asynchronously.
- ✓ **Circuit Breaker:** Avoid cascading failures.

8. **Your Product service has an existing API /products used by mobile apps. You want to release a new version without breaking old clients. How would you implement API versioning?**

- ✓ **URL Versioning:** /v1/products, /v2/products
- ✓ **Header Versioning: Accept:** application/vnd.app.v2+json
- ✓ **Backward Compatibility**: Keep old endpoints while supporting new features.

9. **Your Notification service consumes Kafka messages to send emails. Sometimes it fails to process messages. How would you ensure no messages are lost and the system recovers gracefully?**

- ✓ **Enable retries:** Spring Kafka retry mechanism.
- ✓ **Dead Letter Topic (DLT):** Capture failed messages for later processing.
- ✓ **Idempotent Consumers:** Avoid duplicate processing.

10. **Your Payment and Inventory services consume the same OrderCreated Kafka event. Sometimes Payment processes successfully, but Inventory fails. How would you prevent inconsistent data across services?**

- ✓ Use Saga Pattern (Choreography or Orchestration) for distributed transaction handling.
- ✓ Ensure idempotent event consumers for safe retries.
- ✓ Use Dead Letter Topic (DLT) for failed events.