Java Microservices Interview Questions & MCQs with Answers & Explanations

Concept-Based, Code-Based & Scenario-Based Questions

Q#1. Your team is considering migrating a large monolithic application to a microservices architecture. What are the potential benefits you expect to gain from this migration? *(Scenario-based, Multi-select)*

- A) Easier deployment and independent scaling of individual services.
- B) Reduced operational overhead and simplified debugging.
- C) Improved fault isolation and technology heterogeneity.
- D) Faster development cycles for the entire application.

Answer: A), C)

Explanation:

- **(A) Correct.** Microservices allow for independent deployment of services and scaling of specific services based on demand, rather than scaling the entire application.
- **(B) Incorrect.** While microservices offer benefits, they often introduce increased operational complexity due to distributed systems, and debugging can be more challenging across multiple services.
- **(C) Correct.** Faults in one microservice are less likely to bring down the entire application (improved fault isolation). Also, different services can use different technologies best suited for their specific needs (technology heterogeneity).
- **(D) Incorrect.** While individual service development can be faster, managing the entire microservices ecosystem can introduce overhead, and initial migration can be time-consuming.

Q#2. In a microservices architecture, which pattern is commonly used to handle cross-cutting concerns such as authentication, logging, and monitoring without tightly coupling them to individual services? *(Concept-based, Single-select)*

A) Database per Service
B) API Gateway
C) Circuit Breaker
D) Event Sourcing

Answer: B)

Explanation:

- **A) Incorrect.** Database per Service is a data management pattern.
- **B) Correct.** An API Gateway acts as a single-entry point for all clients, routing requests to the appropriate microservice and handling cross-cutting concerns like authentication, SSL termination, and rate limiting.
- **C) Incorrect.** Circuit Breaker is a fault tolerance pattern.
- **D) Incorrect.** Event Sourcing is a data persistence pattern.

Q#3. You are developing a Java microservice that needs to communicate with another microservice using REST. Which of the following approaches can you use to achieve this communication efficiently and robustly?
(Scenario-based, Multi-select)

- A) Using RestTemplate for synchronous HTTP calls.
- B) Employing WebClient for reactive and non-blocking HTTP calls.
- C) Implementing JMS (Java Message Service) for asynchronous messaging
- D) Directly calling the other microservice's internal methods via RMI.

Answer: A), B), C)

Explanation:

- **A) Correct**. RestTemplate is a synchronous HTTP client provided by Spring, suitable for simple REST interactions.
- **B) Correct.** WebClient is a non-blocking, reactive HTTP client from Spring WebFlux, offering better performance and scalability for microservice communication.
- **C) Correct.** JMS can be used for asynchronous communication between microservices, decoupling them and improving resilience.
- **D) Incorrect.** Directly calling internal methods via RMI (Remote Method Invocation) is generally not recommended in microservices as it creates tight coupling and breaks the service boundary.

Q#4. You have a microservice that needs to register with a Eureka server and also discover other services. Which of the following properties are essential to configure in application.properties for this microservice?
(Code-based, Multi-select)

- A) eureka.client.register-with-eureka=true
- B) eureka.client.fetch-registry=true
- C) eureka.server.enable-self-preservation=false
- D) eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka

Answer: A), B), D)

Explanation:

- **A) Correct.** This property ensures that the microservice registers itself with the Eureka server.
- **B) Correct.** This property enables the microservice to fetch the registry from the Eureka server, allowing it to discover other services.
- **C) Incorrect.** This property is related to the Eureka server's self- preservation mode, not a client's registration or discovery.
- **D) Correct.** This property specifies the URL of the Eureka server where the client should register and fetch the registry from.

Q#5. Which of the following code snippets correctly defines a Feign client for a service named user-service that exposes a GET endpoint /users/{id} ?
(Code-based, Single-select)

- **A)**

```
@FeignClient("user-service")

public interface UserClient {

    @GetMapping("/users/{id}")

     User getUserById(@PathVariable("id") Long id);

}
```

- **B)**

```
@FeignClient(url = "http://localhost:8080/user-service")

public interface UserClient {

    @RequestMapping(method = RequestMethod.GET, value =
"/users/{id}")

     User getUserById(@PathVariable Long id);

}
```

- **C)**

```
@FeignClient(name = "user-service")

public interface UserClient {
```

```
    @GetMapping("/users/{id}")
    User getUserById(Long id);
}
```

- **D)**

```
@FeignClient("user-service")
public interface UserClient {
    @PostMapping("/users/{id}")
    User getUserById(@PathVariable("id") Long id);
}
```

Answer: A)

Explanation:

- **A) Correct.** This snippet correctly uses the service name, @FeignClient with @GetMapping for the HTTP method, and @PathVariable to bind the path variable.
- **B) Incorrect.** While functional, specifying the url directly defeats the purpose of service discovery with Feign, and @RequestMapping is less specific than @GetMapping .
- **C) Incorrect.** The @PathVariable annotation is missing, which is necessary to bind the id parameter to the path variable.
- **D) Incorrect.** @PostMapping is used for POST requests, not GET requests as specified in the question.

Q#6. You are using Spring Cloud OpenFeign to communicate between microservices. What are the advantages of using Feign compared to directly using RestTemplate or WebClient for inter-service communication?
(Concept-based, Multi-select)

- A) Reduced boilerplate code and improved readability.
- B) Automatic integration with service discovery (e.g., Eureka).
- C) Built-in support for client-side load balancing (e.g., Ribbon).
- D) Enhanced performance due to asynchronous request processing by default

Answer: A), B), C)

Explanation:

- **A) Correct.** Feign uses a declarative approach, significantly reducing the boilerplate code required for HTTP client implementation compared to manual RestTemplate or WebClient usage.
- **B) Correct.** Feign integrates seamlessly with service discovery mechanisms like Eureka, allowing you to refer to services by their logical names rather than hardcoded URLs.
- **C) Correct.** Feign works in conjunction with client-side load balancers (like Ribbon, which is often included by default with Spring Cloud Netflix dependencies) to distribute requests across multiple instances of a service.
- **D) Incorrect.** Feign, by default, performs synchronous requests. While it can be configured for asynchronous operations, it's not its default or primary performance advantage over WebClient which is inherently reactive and non-blocking.

Q#7. You are setting up a Spring Cloud Config Server to manage configurations for your microservices. Which of the following backend storage options can you use to store configuration files, and what are their typical use cases?
(Concept-based, Multi-select)

- A) Git repository (e.g., GitHub, GitLab) for version control and collaboration.
- B) Local filesystem for simple development setups or testing.
- C) HashiCorp Vault for secure storage of sensitive credentials.
- D) JDBC-compliant database for dynamic configuration updates.

Answer: A), B), C)

Explanation:

- **A) Correct.** Git is the most common and recommended backend for Config Server due to its version control capabilities, ease of collaboration, and ability to manage different profiles (e.g., dev, prod).
- **B) Correct.** The local filesystem can be used for quick development or testing purposes, but it's not suitable for production environments due to lack of versioning and centralized management.
- **C) Correct.** HashiCorp Vault can be integrated with Config Server to securely store and retrieve sensitive information like passwords, API keys, and tokens.
- **D) Incorrect.** While it's possible to store configurations in a database, it's not a typical or recommended backend for Spring Cloud Config Server. Git or Vault are preferred for their specific strengths.

Q#8. What is the primary function of Spring Cloud Gateway in a microservices architecture? *(Concept-based, Single-select)*

- A) To provide a centralized logging mechanism
- B) To act as an API Gateway, routing requests to appropriate microservices
- C) To manage distributed transactions
- D) To enable asynchronous communication between services

Answer: **B)**

Explanation:

- **A) Incorrect.** Centralized logging is handled by dedicated logging solutions.
- **B) Correct.** Spring Cloud Gateway is a powerful and flexible API Gateway that routes requests to various microservices, handles cross-cutting concerns, and provides features like load balancing, security, and monitoring.
- **C) Incorrect.** Distributed transaction management is a separate concern.
- **D) Incorrect.** Asynchronous communication is typically handled by message brokers.

Q#9. Which of the following configuration snippets correctly defines a route in Spring Cloud Gateway that forwards requests from /my-service/** to a service named my-service registered with Eureka? *(Code-based, Single-select)*

- A)

```
spring:
  cloud:
    gateway:
      routes:
        - id: my_service_route
          uri: lb://my-service
          predicates:
            - Path=/my-service/**
```

- B)

```
spring:
  cloud:
    gateway:
      routes:
        - id: my_service_route
          uri: http://localhost:8080/my-service
          predicates:
            - Path=/my-service/**
```

- C)

```
spring:
  cloud:
    gateway:
      routes:
        - id: my_service_route
          uri: lb://my-service
          filters:
            - RewritePath=/my-service/(?<segment>.*), /${segment}
```

- D)

```
spring:
  cloud:
    gateway:
      routes:
        - id: my_service_route
          uri: lb://my-service
          predicates:
            - Host=my-service.com
```

Answer: A)

Explanation:

- **A) Correct.** This configuration correctly defines a route with a Path predicate and uses lb://my-service to leverage client-side load balancing and service discovery for the my-service.
- **B) Incorrect.** Using a hardcoded http://localhost:8080 defeats the purpose of service discovery and makes the gateway less resilient.

- **C) Incorrect.** This snippet is missing the to define when the route should be applied. The predicates section RewritePath filter is used for path manipulation, not for defining the routing condition.
- **D) Incorrect.** While Host predicates are valid, the question specifically asks for routing based on a path (/my service/**).

Q#10. You are implementing an API Gateway using Spring Cloud Gateway. You need to apply a rate limiting filter to a specific route and also add a custom header to all requests passing through the gateway. How would you achieve this? *(Scenario-based, Multi-select)*

- A) Configure a RequestRateLimiter filter for the specific route.
- B) Implement a custom GlobalFilter to add the custom header.
- C) Use a AddRequestHeader filter for the specific route.
- D) Define a custom GatewayFilterFactory for both rate limiting and header addition.

Answer: A), B), C)

Explanation:

- **A) Correct.** Spring Cloud Gateway provides a RequestRateLimiter filter that can be configured per route to control the rate of requests.
- **B) Correct.** A custom GlobalFilter can be implemented to apply logic (like adding a custom header) to all requests passing through the gateway, regardless of the route.
- **C) Correct.** The AddRequestHeader filter can be used to add a specific header to requests for a particular route.
- **D) Incorrect.** While you can define custom GatewayFilterFactory instances, Spring Cloud Gateway already provides built-in filters for rate limiting and adding request headers, making custom factories unnecessary for these specific requirements.

Q#11. What is the main benefit of using Spring Cloud Config Server in a microservices architecture? *(Concept-based, Single-select)*

- A) To provide a centralized logging solution.
- B) To manage and distribute configuration properties for all microservices.
- C) To enable distributed tracing across services.
- D) To facilitate inter-service communication through message queues.

Answer: B)

Explanation:

- **A) Incorrect.** Centralized logging is typically handled by tools like ELK stack.
- **B) Correct.** Spring Cloud Config Server provides a centralized place to manage externalized configuration for applications across all environments.
- **C) Incorrect.** Distributed tracing is handled by tools like Zipkin or Jaeger.
- **D) Incorrect.** Message queues (like Kafka or RabbitMQ) facilitate asynchronous communication.

Q#12. In a microservices architecture, why is distributed logging more challenging than in a monolithic application? *(Concept-based, Single-select)*

- A) Because microservices use different programming languages.
- B) Because logs are scattered across multiple independent services and machines.
- C) Because microservices generate less log data.
- D) Because logging frameworks are not compatible with microservices.

Answer: **B**)

Explanation:

- **A) Incorrect.** While possible, language differences are not the primary challenge for distributed logging.
- **B) Correct.** The independent nature and distribution of microservices mean that logs are generated by many different services, often running on different hosts, making centralized collection and analysis difficult.
- **C) Incorrect.** Microservices can generate a significant amount of log data.
- **D) Incorrect.** Standard logging frameworks are compatible; the challenge lies in aggregation.

Q#13. You have a Java microservice deployed on Kubernetes, and you need to ensure that it can gracefully shut down when Kubernetes sends a termination signal (SIGTERM). What is the best practice for handling this in your Spring Boot application? *(Scenario-based, Multi-select)*

- A) Ignore the SIGTERM signal, as Kubernetes will forcefully terminate the pod after a timeout.
- B) Implement a preStop hook in your Kubernetes Deployment to execute a shutdown script.
- C) Configure your Spring Boot application to listen for ContextClosedEvent and perform cleanup tasks.
- D) Set a very short terminationGracePeriodSeconds in your Kubernetes Deployment to speed up termination.

Answer: B), C)

Explanation:

- **A) Incorrect**. Ignoring SIGTERM can lead to abrupt terminations, data loss, or corrupted states.
- **B) Correct**. A preStop hook is executed immediately before a container is terminated. This allows you to send a signal to your application (e.g., an HTTP request to a shutdown endpoint) to initiate a graceful shutdown.
- **C) Correct**. Spring Boot applications can listen for ContextClosedEvent to perform graceful shutdown tasks like closing connections, flushing logs, and completing ongoing requests. This is a crucial part of making your application Kubernetes-aware.
- **D) Incorrect**. A short terminationGracePeriodSeconds can lead to forceful termination if the application doesn't shut down quickly enough.

Q#14. Which of the following best describes the two main approaches to implementing the Saga pattern? *(Concept-based, Single-select)*

- A) Choreography and Orchestration.
- B) Synchronous and Asynchronous.
- C) Request- Reply and Publish-Subscribe.
- D) Event Sourcing and CQRS.

Answer: A)

Explanation:

- **A) Correct.** The two main approaches are Choreography (where services communicate directly via events) and Orchestration (where a central orchestrator coordinates the saga).
- **B) Incorrect.** These describe communication styles, not Saga implementation approaches.
- **C) Incorrect.** These describe messaging patterns, not Saga implementation approaches.
- **D) Incorrect.** These are architectural patterns, but not directly the two main approaches to Saga implementation.

Q#15. You are designing an event- driven microservices system where services communicate asynchronously via Kafka. Which of the following are key considerations for ensuring reliable and scalable message processing? *(Scenario-based, Multi-select)*

- A) **Idempotent Consumers:** Designing consumers to process messages multiple times without causing unintended side effects
- B) **Message Ordering:** Ensuring that messages within a topic partition are processed in the order they were produced

- C) **Consumer Groups:** Using consumer groups to distribute message consumption across multiple instances of a service
- D) **Synchronous Producers:** Configuring producers to wait for acknowledgment from Kafka brokers before sending the next message

Answer: A), B), C), D)

Explanation:

- **A) Correct.** Idempotent consumers are crucial for fault tolerance in distributed systems. If a message is redelivered (e.g., due to a consumer crash and restart), an idempotent consumer will process it without creating duplicate data or incorrect state.
- **B) Correct.** Kafka guarantees message ordering within a single partition. To ensure global ordering for related events, careful partitioning strategies are required.
- **C) Correct.** Consumer groups allow multiple consumer instances to share the workload of reading from topics, enabling horizontal scaling of message processing.
- **D) Correct.** While synchronous producers can reduce throughput, configuring them to wait for acknowledgments (e.g., acks=all) ensures that messages are durably written to Kafka, improving reliability. This is a trade-off between latency/throughput and data durability.

Q#16: You have deployed several Java microservices, and you want to set up a robust logging and monitoring solution using the ELK stack. Which of the following considerations are important for effectively implementing this solution? *(Concept-based, Multi-select)*

- A) **Structured Logging:** Ensuring your microservices emit logs in a structured format (e.g., JSON) for easier parsing
- B) **Centralized Log Collection:** Using agents (like Filebeat) on each microservice instance to ship logs to Logstash or Elasticsearch
- C) **Retention Policies:** Defining how long logs should be stored in Elasticsearch to manage storage costs
- D) **Real-time Dashboards:** Creating Kibana dashboards to visualize key metrics and log patterns in real-time

Answer: A), B), C), D)

Explanation:

- **A) Correct.** Structured logging makes it significantly easier for Logstash to parse and extract meaningful fields from your logs, which is crucial for effective searching and visualization in Kibana.
- **B) Correct.** Centralized log collection is fundamental to the ELK stack. Agents like Filebeat are commonly used to efficiently collect logs from various sources and forward them to Logstash or directly to Elasticsearch.
- **C) Correct.** Log data can grow very rapidly. Implementing retention policies is essential to manage storage costs and ensure compliance requirements are met.
- **D) Correct.** Kibana dashboards provide real-time visibility into the health and performance of your microservices, allowing for quick identification of issues and trends.

Q#17. Which one is the standard way to define a REST client using OpenFeign in a Spring Boot Microservices application?
(Code-based, Single-select)

- A)

```
@FeignClient(name = "example-service")
public interface ExampleClient {

    @GetMapping("/endpoint")
    String getEndpointData();
}
```

- B)

```
@FeignClient("example-service")
public interface ExampleClient {

   @RequestMapping(method = RequestMethod.GET, value = "/endpoint")
   String getEndpointData();
}
```

- C)

```
@FeignClient(name = "example-service", url = "http://localhost:8080")
public interface ExampleClient {

    @GetMapping("/endpoint")
    String getEndpointData();
```

```
}
```

- D)

```
@FeignClient(name = "example-service")
public interface ExampleClient {

    @PostMapping("/endpoint")
    String postEndpointData(@RequestBody String data);
}
```

Answer: A)

Explanation:

- **A) Correct.** This is the standard way to define a REST client using OpenFeign, specifying the service name and using @GetMapping for a GET request.
- **B) Incorrect.** While valid, this option uses @RequestMapping which is less specific compared to @GetMapping.
- **C) Incorrect.** Adding the url property is not typical for defining a REST client with OpenFeign, as it is usually configured via application properties.
- **D) Incorrect.** This example uses @PostMapping, but the question specifies a GET request, so it does not match the requirement.

Q#18. Given the following Feign client interface, what does it do?
(Code-based, Single-select)

```
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;

@FeignClient(name = "service-b")
public interface ServiceBClient {

    @GetMapping("/serviceB")
    String getServiceBResponse();
}
```

- A) Defines a REST client for interacting with Service B
- B) Configures a database connection for Service B
- C) Sets up distributed tracing for Service B
- D) Enables service registration for Service B

Answer: A)

Explanation: This interface defines a REST client for interacting with Service B, using Feign to simplify the HTTP communication.

Q#19. How do you define a Dockerfile to build a container image for a Java microservice? *(Code-based, Single-select)*

- A) dockerfile:

```
FROM openjdk:11-jre
COPY target/my-service.jar /app/my-service.jar
ENTRYPOINT ["java", "-jar", "/app/my-service.jar"]
```

- B) dockerfile:

```
FROM ubuntu:20.04
COPY my-service.jar /app/my-service.jar
ENTRYPOINT ["java", "-jar", "/app/my-service.jar"]
```

- C) dockerfile:

```
FROM openjdk:11
WORKDIR /app
COPY target/my-service.jar .
ENTRYPOINT ["java", "-jar", "my-service.jar"]
```

- D) dockerfile:

```
FROM openjdk:8
COPY target/my-service.jar /opt/my-service.jar
ENTRYPOINT ["java", "-jar", "/opt/my-service.jar"]
```

Answer: A)

Explanation:

- **A) Correct.** This Dockerfile uses an appropriate base image (*openjdk:11-jre*), copies the JAR file to the container, and sets the entry point to run the JAR file.
- **B) Incorrect.** Using *ubuntu:20.04* is less specific and may require additional setup for Java.
- **C) Incorrect.** *openjdk:11* is a complete JDK image and may include unnecessary components for a runtime-only container.
- **D) Incorrect.** *openjdk:8* is outdated compared to *openjdk:11-jre*, which is preferable for current Java versions.

Q#20. You are troubleshooting a complex issue that involves multiple microservices communicating with each other. Which of the following tools or techniques would be most effective for understanding the flow of a request across these services and identifying bottlenecks or failures? *(Scenario-based, Multi-select)*

- A) **Centralized Log Aggregation:** Collecting logs from all services into a single system (e.g., ELK Stack).
- B) **Distributed Tracing:** Using correlation IDs to track a single request as it propagates through multiple services (e.g., Zipkin, Jaeger).
- C) **Health Checks:** Regularly monitoring the health endpoints of individual services.
- D) **Metrics Monitoring:** Collecting performance metrics (e.g., CPU usage, memory) from each service.

Answer: A), B).

Explanation:

- **A) Correct.** Centralized log aggregation is essential for collecting all logs in one place, making it easier to search and analyze them, especially when an issue spans multiple services.
- **B) Correct.** Distributed tracing is specifically designed to visualize the end-to-end flow of a request across service boundaries, showing latency, dependencies, and potential failure points.
- **C) Incorrect.** Health checks indicate if a service is up or down, but don't provide insight into the request flow or bottlenecks.
- **D) Incorrect.** Metrics monitoring provides insights into the performance of individual services but doesn't directly show the path of a single request across multiple services.

Q#21. In Spring Cloud, which library is commonly used to implement the Circuit Breaker pattern? *(Concept-based, Single-select)*

- A) Spring Cloud Stream
- B) Spring Cloud Sleuth
- C) Spring Cloud Hystrix (or Resilience4j)
- D) Spring Cloud Bus

Answer: C)

Explanation:

- **A) Incorrect.** Spring Cloud Stream is for building event driven microservices.
- **B) Incorrect.** Spring Cloud Sleuth is for distributed tracing.
- **C) Correct.** Spring Cloud Hystrix was the original Netflix library for circuit breakers, now largely superseded by Resilience j in newer Spring Cloud versions.
- **D) Incorrect.** Spring Cloud Bus links nodes of a distributed system with a message broker.

Q#22. You are designing a microservice that calls an external, potentially unreliable, third-party API. To improve the resilience of your microservice, which fault tolerance strategies would you consider implementing?
(Scenario-based, Multi-select)

- A) **Retry Mechanism:** Automatically re-attempting failed API calls a few times.
- B) **Bulkhead Pattern:** Isolating resource pools for different services to prevent one from exhausting resources for others.
- C) **Bulkhead Pattern:** Isolating resource pools for different services to prevent one from exhausting resources for others.
- D) **Rate Limiting:** Restricting the number of requests your microservice sends to the third-party API.

Answer: A), B), C), D)

Explanation:

- **A) Correct.** A retry mechanism can help overcome transient network issues or temporary unavailability of the external API.
- **B) Correct.** While often applied to internal services, the Bulkhead pattern can also be applied to external calls by isolating the threads or connection pools used for different external dependencies, preventing one slow external service from impacting others.
- **C) Correct.** Timeouts are crucial to prevent your microservice from hanging indefinitely if the external API is slow or unresponsive.
- **D) Correct.** Rate limiting your calls to the third-party API can prevent you from overwhelming it and getting blocked, which contributes to overall resilience.

Q#23. You are designing an e commerce application with microservices for Order, Payment, and Inventory. When a customer places an order, you need to ensure that the payment is processed and inventory is updated. If any step fails, all previous steps must be compensated. Which of the following are key considerations when implementing a Saga for this scenario?
(Scenario-based, Multi-select)

- A) **Defining Compensation Transactions:** For each successful step, a corresponding compensation transaction must be defined to undo its effects if a subsequent step fails
- B) **Eventual Consistency:** Understanding that data consistency will be achieved eventually, not immediately, across all services
- C) **Idempotency of Operations:** Ensuring that each operation in the saga can be safely retried without causing duplicate side effects
- D) **Using a Distributed Transaction Coordinator (DTC):** Employing a DTC to manage the two-phase commit across all services

Answer: A), B), C)

Explanation:

- **A) Correct.** A core principle of Saga is the ability to compensate for failed transactions. Each successful step must have a corresponding compensation action to revert its changes if the saga fails.
- **B) Correct.** Sagas typically lead to eventual consistency, meaning that at any given moment, data across services might be temporarily inconsistent, but will eventually converge to a consistent state.
- **C) Correct.** Idempotency is crucial for robustness. If a message or operation is retried (e.g., due to network issues or service restarts), it should not lead to incorrect state or duplicate actions.
- **D) Incorrect.** The Saga pattern is specifically designed to avoid the use of a Distributed Transaction Coordinator (DTC) and two-phase commit, which are often problematic in distributed microservices environments due to their blocking nature and single point of failure.

Q#24. You are considering adopting a serverless approach for certain parts of your Java microservices application, such as processing image uploads or handling infrequent background tasks. What are the potential benefits and challenges you should consider? *(Concept-based, Multi-select)*

- A) **Benefits:** Reduced operational costs due to pay-per-execution billing and automatic scaling
- B) **Benefits:** Simplified debugging and monitoring due to centralized logging
- C) **Challenges:** Potential for vendor lock-in and cold start latencies

- D) **Challenges:** Easier management of long-running, stateful applications

Answer: A), C)

Explanation:

- **A) Correct.** Serverless platforms typically charge based on execution time and resource consumption, leading to cost savings for intermittent workloads. They also provide automatic scaling to handle varying loads.
- **B) Incorrect.** While logging can be centralized, debugging serverless functions can be more complex due to their ephemeral nature and distributed execution.
- **C) Correct.** Adopting a specific serverless platform can lead to vendor lock-in. Also, cold starts (the delay when a function is invoked after a period of inactivity) can introduce latency.
- **D) Incorrect.** Serverless is generally not ideal for long-running, stateful applications due to its stateless nature and execution limits. Managing state across serverless functions can be complex.

Q#25. What is the primary advantage of deploying Java microservices in the cloud? *(Concept-based, Single-select)*

A) Reduced development time for all applications
B) Increased on-demand scalability and elasticity
C) Elimination of all security concerns
D) Elimination of all security concerns

Answer: B)

Explanation:

- **A) Incorrect.** While cloud services can accelerate development, it's not a guaranteed reduction for all applications.
- **B) Correct.** The cloud provides the ability to scale resources up or down on demand, which is a key benefit for microservices that have varying workloads.
- **C) Incorrect.** Cloud security is a shared responsibility, and security concerns are not eliminated.
- **D) Incorrect.** Debugging distributed systems in the cloud can be more complex than in on-premise environments.

Q#26. Your organization is planning to migrate its on-premise Java microservices to a public cloud provider. What are the key factors you should consider during this cloud adoption process? *(Scenario-based, Multi-select)*

A) **Cost Management:** Analyzing the cloud provider's pricing models and implementing cost optimization strategies.
B) **Vendor Lock-in:** Assessing the risk of becoming dependent on a single cloud provider's proprietary services.
C) **Security and Compliance:** Ensuring that the cloud provider meets your organization's security and compliance requirements.
D) **Refactoring Applications:** Determining whether to refactor applications to be more cloud-native or to use a lift-and-shift approach.

Answer: A), B), C), D)

Explanation:

- **A) Correct.** Cloud costs can quickly escalate if not managed properly. It's crucial to understand the pricing models and implement strategies like right-sizing instances and using reserved instances to control costs.
- **B) Correct.** Vendor lock-in is a significant concern. Using open standards and multi-cloud strategies can help mitigate this risk.
- **C) Correct.** Security and compliance are paramount. You must ensure that the cloud provider's security measures and compliance certifications align with your organization's needs.
- **D) Correct.** The migration strategy is a key decision. A lift-and-shift approach is faster but may not fully leverage cloud benefits, while refactoring for cloud-native architectures can be more complex but offers greater long-term advantages.

Q#27. What is the primary goal of adopting DevOps practices in a microservices environment? *(Concept-based, Single-select)*

A) To reduce the number of microservices in an application
B) To automate and integrate the processes between software development and IT operations teams
C) To replace all manual testing with automated testing
D) To eliminate the need for version control systems

Answer: B)

Explanation:

- **A) Incorrect.** DevOps focuses on process improvement, not reducing service count.
- **B) Correct.** DevOps aims to shorten the systems development life cycle and provide continuous delivery with high software quality by fostering collaboration and automation between development and operations.
- **C) Incorrect.** While automation is a key part of DevOps, it doesn't eliminate all manual testing.

- **D) Incorrect.** Version control systems are fundamental to DevOps.

Q#28. Which of the following tools is commonly used in a DevOps pipeline for continuous integration (CI) of Java microservices? *(Concept-based, Single-select)*

A) Apache Maven
B) Jenkins
C) Docker Compose
D) Ansible

Answer: B)

Explanation:

- **A) Incorrect.** Apache Maven is a build automation tool, often used within a CI process, but not a CI server itself.
- **B) Correct.** Jenkins is a popular open-source automation server widely used for continuous integration and continuous delivery (CI/CD) pipelines, including building and testing Java applications.
- **C) Incorrect.** Docker Compose is for defining and running multi- container Docker applications locally.
- **D) Incorrect.** Ansible is an automation engine for configuration management, application deployment, and orchestration.

Q#29. What is a common approach to integrate Artificial Intelligence (AI) or Machine Learning (ML) models into a microservices architecture? *(Concept-based, Single-select)*

A) Embed the entire AI/ML training pipeline within each microservice
B) Deploy AI/ML models as separate, dedicated microservices
C) Use AI/ML models only for batch processing, not real-time
D) Replace all traditional business logic with AI/ML models

Answer: B)

Explanation:

- **A) Incorrect.** Embedding the entire training pipeline in each microservice would make them too large and complex.
- **B) Correct.** Deploying AI/ML models as dedicated microservices (often called "model serving" or "inference services") allows for independent scaling, deploymen t, and management of the models.
- **C) Incorrect.** AI/ML models can be used for both batch and real-time processing in microservices.
- **D) Incorrect.** AI/ML models augment, rather than replace, traditional business logic.

Q#30. You are building a recommendation engine as a microservice that uses a machine learning model. Which of the following considerations are important for operationalizing this AI/ML microservice effectively?
(Scenario-based, Multi-select)

A) **Model Versioning and Deployment:** Managing different versions of the ML model and deploying them independently.
B) **Data Drift Monitoring:** Continuously monitoring the input data to the model for changes that might degrade performance.
C) **Explainability and Interpretability:** Ensuring that the model's predictions can be understood and explained, especially in critical applications.
D) **GPU Acceleration:** Utilizing GPUs for model inference to improve performance, especially for deep learning models.

Answer: A), B), C), D)

Explanation:

- **A) Correct.** Just like code, ML models evolve. Versioning and independent deployment of models are crucial for A/B testing, rollback capabilities, and managing model updates without affecting the entire application.
- **B) Correct.** Data drift (changes in the distribution of input data over time) can significantly degrade model performance. Monitoring for data drift is essential for maintaining model accuracy in production.
- **C) Correct.** For many AI/ML applications, especially in regulated industries, understanding *why* a model made a certain prediction (explainability) and being able to interpret its behavior is critical for trust and debugging.
- **D) Correct.** For computationally intensive models, particularly deep learning models, leveraging GPU acceleration can drastically improve inference speed and throughput, which is vital for real-time microservices.

Q#31. Which of the following are common challenges encountered when adopting a microservices architecture? *(Concept-based, Multi-select)*

A) Increased complexity in deployment and operations.
B) Difficulty in achieving technology heterogeneity.
C) Challenges with distributed data management and consistency.
D) Reduced need for robust monitoring and logging.

Answer: A), C)

Explanation:

- **A) Correct.** Microservices introduce complexity in managing many independent services, their deployments, and their interactions.

- **B) Incorrect.** Microservices *enable* technology heterogeneity, allowing different services to use different technologies best suited for their needs.
- **C) Correct.** Maintaining data consistency across multiple independent databases in a distributed microservices environment is a significant challenge, often addressed with patterns like Saga.
- **D) Incorrect.** The distributed nature of microservices *increases* the need for robust monitoring, logging, and tracing to understand system behavior and troubleshoot issues.

Q#32. Your organization has decided to break down a large monolithic application into microservices. One of the key decisions is how to divide the monolith. Which principle should guide the decomposition of the monolith into services? *(Scenario-based, Single-select)*

- A) Decompose by technical layers (e.g., UI, business logic, data access).
- B) Decompose by business capabilities or bounded contexts.
- C) Decompose by team size, with each team owning a small service.
- D) Decompose by database tables, with each service owning a few tables.

Answer: B)

Explanation:

- **A) Incorrect.** Decomposing by technical layers often leads to distributed monoliths, where changes in one layer still require changes across multiple services.
- **B) Correct.** Decomposing by business capabilities or bounded contexts (as defined in Domain-Driven Design) ensures that each microservice is cohesive, loosely coupled, and responsible for a specific business function, making it easier to develop, deploy, and scale independently.
- **C) Incorrect.** While team size can influence service boundaries, it should not be the primary driver for decomposition.
- **D) Incorrect.** Decomposing by database tables can lead to tight coupling if multiple services need to access the same data.

Q#33. Which of the following are key features provided by Netflix Eureka for service discovery? *(Concept-based, Multi-select)*

A) **Client-side load balancing:** Eureka provides built-in load balancing capabilities for registered services.
B) **Service registration:** Microservices can register themselves with the Eureka server upon startup.
C) **Service lookup:** Clients can query the Eureka server to find the network locations of registered services.

D) **Health checks:** Eureka performs active health checks on registered instances to remove unhealthy ones from the registry.

Answer: B), C), D)

Explanation:

- **A) Incorrect.** While Eureka works *with* client-side load balancers (like Ribbon), Eureka itself does not provide load balancing. It provides the information for load balancers to work.
- **B) Correct.** Microservices register their network locations (IP address, port) with the Eureka server.
- **C) Correct.** Clients (other microservices or API Gateways) can query Eureka to get a list of available instances for a particular service.
- **D) Correct.** Eureka performs basic health checks to ensure that registered instances are still alive and responsive. If an instance becomes unhealthy, it is de-registered.

Q#34. Your Eureka server is experiencing network partitioning, where some microservices can reach it, and others can't. What is the default behavior of Eureka during such a network partition, and why is it designed this way? *(Scenario-based, Single-select)*

A) Eureka will shut down to prevent inconsistent data.
B) Eureka will enter "self- preservation mode" to avoid evicting healthy instances.
C) Eureka will immediately evict all instances that haven't sent heartbeats.
D) Eureka will elect a new leader and continue normal operation.

Answer: B)

Explanation:

- **A) Incorrect.** Eureka is designed for resilience and does not shut down during network partitions.
- **B) Correct.** Eureka is designed for availability over consistency (AP in CAP theorem). In a network partition, it enters self- preservation mode, where it stops evicting instances even if they fail to send heartbeats. This prevents healthy instances from being de-registered due to network issues, ensuring that clients can still discover them.
- **C) Incorrect.** This is the behavior Eureka tries to avoid in self-preservation mode.
- **D) Incorrect.** Eureka is not a leader- election system like ZooKeeper or others.

Q#35. When using Spring Cloud OpenFeign, how does it typically resolve the actual network address of the target microservice?
(Concept-based, Single-select)

A) By directly using the url property specified in @FeignClient.
B) By querying a DNS server for the service name.
C) By integrating with a service discovery mechanism like Eureka or Consul.
D) By reading the host and port from a local configuration file.

Answer: C)

Explanation:

- **A) Incorrect.** While url can be specified, it's generally discouraged in a dynamic microservices environment. Feign's power comes from service discovery.
- **B) Incorrect.** While DNS is involved in network resolution, Feign primarily relies on service discovery for dynamic service location.
- **C) Correct.** OpenFeign integrates seamlessly with service discovery mechanisms (like Eureka, Consul, or Kubernetes service discovery) to resolve the logical service name (specified in @FeignClient) to actual physical network addresses of service instances.
- **D) Incorrect.** This would lead to static and inflexible configurations.

Q#36. Which of the following are benefits of using Spring Cloud Config Server for managing microservice configurations? *(Concept-based, Multi-select)*

A) **Centralized Management:** All configurations are stored in a single, version- controlled location.
B) **Dynamic Updates:** Configuration changes can be pushed to running microservices without requiring a restart.
C) **Environment-Specific Configurations:** Easily manage different configurations for various environments (e.g., development, production).
D) **Reduced Security Risk:** Eliminates the need for secure storage of sensitive credentials.

Answer: A), B), C)

Explanation:

- **A) Correct.** Config Server centralizes configuration, often backed by Git, providing version control and a single source of truth.
- **B) Correct.** With Spring Cloud Bus and @RefreshScope , microservices can dynamically refresh their configurations without a full restart.
- **C) Correct.** Config Server allows for profile- specific configurations, enabling different settings for different environments.

- **D) Incorrect.** While Config Server can integrate with secure stores like HashiCorp Vault, it doesn't eliminate the need for secure storage; it provides a mechanism to manage it.

Q#37. You have a Spring Boot microservice configured to fetch properties from a Spring Cloud Config Server. You've updated a property in the Git repository backing the Config Server. How can you ensure your running microservice picks up this new configuration without a full restart?
(Scenario-based, Single-select)

A) Manually restart the microservice instance.
B) Send a POST request to the /actuator/refresh endpoint of the microservice.
C) The microservice will automatically detect and load the new configuration.
D) Update the property directly in the microservice's application.properties file.

Answer: B)

Explanation:

- **A) Incorrect.** While restarting works, it defeats the purpose of dynamic configuration.
- **B) Correct.** By enabling Spring Boot Actuator and including Spring Cloud Bus, sending a POST request to /actuator/refresh (or /actuator/bus-refresh if using Spring Cloud Bus) will trigger the microservice to re fetch its configuration from the Config Server.
- **C) Incorrect.** Microservices do not automatically detect changes in the Config Server without a trigger.
- **D) Incorrect.** This would be a local override and not leverage the centralized Config Server.

Q#38. Which of the following are key features provided by Spring Cloud Gateway? *(Concept-based, Multi-select)*

A) **Routing:** Directing incoming requests to the correct microservice instances
B) **Filtering:** Applying cross-cutting concerns like security, rate limiting, and logging to requests
C) **Load Balancing:** Distributing requests across multiple instances of a microservice
D) **Service Discovery:** Registering and de-registering microservice instances

Answer: A), B), C)

Explanation:

- **A) Correct.** Routing is the core function of an API Gateway, directing requests based on various predicates (path, host, headers, etc.).
- **B) Correct.** Filters allow for pre- and post-processing of requests, enabling features like authentication, rate limiting, and header manipulation.
- **C) Correct.** Spring Cloud Gateway integrates with client-side load balancers (like Ribbon) to distribute requests among available service instances.
- **D) Incorrect.** While Gateway uses service discovery, it does not *provide* service discovery itself; it consumes information from service discovery mechanisms like Eureka.

Q#39. You want to implement a custom filter in Spring Cloud Gateway to add a unique request ID to every incoming request before it is routed to a downstream microservice. Which interface would you implement to achieve this? *(Code-based, Single-select)*

A) GatewayFilter
B) WebFilter
C) Ordered
D) GlobalFilter

Answer: D)

Explanation:

- **A) Incorrect.** GatewayFilter is for filters applied to specific routes.
- **B) Incorrect.** WebFilter is a more general Spring WebFlux filter, but is specific to Spring Cloud Gateway.
- **C) Incorrect.** Ordered is an interface for defining the order of execution for components, not for implementing a filter itself.
- **D) Correct.** GlobalFilter is the interface to implement for filters that apply to all routes managed by the Spring Cloud Gateway, making it suitable for adding a unique request ID to every incoming request.

Q#40. Your microservice relies on a third-party payment gateway. During peak hours, the payment gateway occasionally becomes slow or unresponsive, leading to timeouts in your service. To prevent this external dependency from causing your entire service to fail, you decide to implement a Circuit Breaker. When the Circuit Breaker is in the 'open' state, what typically happens to subsequent requests to the payment gateway? *(Scenario-based, Multi-select)*

A) Requests are immediately retried until successful.
B) Requests are immediately failed without attempting to call the payment gateway.
C) Requests are queued and processed once the payment gateway recovers.
D) Requests are redirected to a fallback service.

Answer: B), D)

Explanation:

- **A) Incorrect.** Retries are typically part of the 'closed' state or a separate retry mechanism, not the 'open' state.
- **B) Correct.** When the Circuit Breaker is in the 'open' state, it prevents further calls to the failing service, immediately failing subsequent requests. This gives the failing service time to recover and prevents the calling service from wasting resources on calls that are likely to fail.
- **C) Incorrect.** Queuing requests can lead to backpressure and resource exhaustion.
- **D) Correct.** While not the *only* thing that happens, redirecting to a fallback service is a common strategy when a circuit breaker is open, providing a degraded but still functional experience to the user.

Q#41. Which of the following are common challenges associated with distributed tracing in a microservices environment? *(Concept-based, Multi-select)*

- A) **Instrumentation Overhead:** Adding tracing libraries and code can introduce performance overhead
- B) **Context Propagation:** Ensuring that trace IDs and span IDs are correctly passed across service boundaries
- C) **Data Volume:** The sheer volume of trace data generated can be challenging to store and analyze
- D) **Lack of Standardization:** Inconsistent tracing standards across different programming languages or frameworks

Answer: A), B), C), D)

Explanation:

- **A) Correct.** While modern tracing libraries are optimized, there is always some overhead associated with instrumenting code and collecting trace data.
- **B) Correct.** Correctly propagating the trace context (trace ID, span ID, parent span ID) across various communication protocols (HTTP, message queues, gRPC) is crucial and can be complex.
- **C) Correct.** Distributed tracing can generate a massive amount of data, requiring robust storage and analysis solutions.
- **D) Correct.** While OpenTelemetry is gaining traction, historically, different tracing systems (e.g., Zipkin, Jaeger) had their own standards, leading to interoperability challenges.

Q#42. In a Spring Boot application using Spring Cloud Sleuth, how would you typically inject the current TraceId into a custom log message?
(Code-based, Single-select)

A) logger.info("Request MappedDiagnosticContext.get("traceId")); processed: " +
processed:
B) " + logger.info("Request Tracer.currentSpan().context().traceId());
C) logger.info("Request processed: " + Span.current().context().traceId());
D) logger.info("Request processed: " + ThreadContext.get("traceId"));

Answer: B)

Explanation:

- **A) Incorrect.** MappedDiagnosticContext (MDC) is used by logging frameworks, and Sleuth populates it, but directly accessing it like this is not the most idiomatic way to get the trace ID from Sleuth's Tracer.
- **B) Correct.** Spring Cloud Sleuth integrates with Brave (and now Micrometer Tracing). The Tracer interface provides access to the current Span , from which you can get the TraceContext and then the traceId.
- **C) Incorrect.** Span.current() is part of OpenTelemetry API, but the question implies Spring Cloud Sleuth, which uses Brave/Micrometer Tracing.
- **D) Incorrect.** ThreadContext is a Log j concept, similar to MDC, but not the direct API for Sleuth's tracing context.

Q#43. Which of the following are common use cases for Kibana in a microservices monitoring setup with the ELK stack?
(Concept-based, Multi-select)

A) **Real-time Log Analysis:** Searching, filtering, and analyzing logs from various microservices.
B) **Performance Monitoring:** Visualizing metrics like response times, error rates, and throughput.
C) **Security Analytics:** Detecting unusual patterns or potential security threats in log data.
D) **Database Management:** Performing CRUD operations on relational databases.

Answer: A), B), C)

Explanation:

- **A) Correct.** Kibana provides powerful search and visualization capabilities for real-time log analysis, allowing users to quickly identify issues and trends.
- **B) Correct.** By ingesting metrics data (e.g., from Prometheus or directly from applications) into Elasticsearch, Kibana can be used to create dashboards for performance monitoring.

- **C) Correct.** Kibana's ability to search and visualize large volumes of log data makes it suitable for security analytics, helping to identify suspicious activities or breaches.
- **D) Incorrect.** Kibana is a visualization tool for Elasticsearch, not a database management tool for relational databases.

Q#44. Your microservices are generating a high volume of logs, and you're concerned about the performance impact of sending all logs directly to Logstash for processing. What alternative lightweight component from the Elastic Stack could you use to ship logs from your microservice instances to Elasticsearch more efficiently? *(Scenario-based, Single-select)*

A) Apache Kafka
B) Redis
C) Filebeat
D) Log4j

Answer: C)

Explanation:

- **A) Incorrect.** Apache Kafka is a message broker, not a log shipper.
- **B) Incorrect.** Redis is an in-memory data structure store, not primarily a log shipper.
- **C) Correct.** Filebeat is a lightweight shipper for forwarding log data from files to Elasticsearch or Logstash. It's designed to be resource-friendly and efficient for log collection at the source.
- **D) Incorrect.** Log4j is a logging framework within applications, not a log shipper.

Q#45. You are designing a new feature for your e-commerce platform where order updates (e.g., status changes) need to be propagated to multiple downstream microservices (e.g., shipping, notification, analytics) in a reliable and decoupled manner. Which Kafka feature would be most suitable for ensuring that all interested services receive these updates without direct point-to-point communication? *(Scenario-based, Single-select)*

A) Kafka Connect
B) Kafka Streams
C) Kafka Topics with multiple consumer groups
D) Kafka MirrorMaker

Answer: C)

Explanation:

- **A) Incorrect.** Kafka Connect is for connecting Kafka to other systems (databases, file systems).
- **B) Incorrect.** Kafka Streams is a client library for building stream processing applications.
- **C) Correct.** By publishing order updates to a Kafka topic, and having each downstream microservice subscribe to this topic using its own consumer group, each service will receive its own copy of all messages, ensuring reliable and decoupled propagation of updates.
- **D) Incorrect.** Kafka MirrorMaker is for replicating data between Kafka clusters.

Q#46. Which of the following are characteristics of a Saga in a microservices architecture? *(Concept-based, Multi-select)*

A) **Atomicity:** All operations within a Saga are guaranteed to either complete successfully or rollback entirely
B) **Long-lived transactions:** Sagas are designed to manage business processes that span multiple services and may take a long time to complete
C) **Compensation:** Each step in a Saga has a corresponding compensation action to undo its effects if a subsequent step fails
D) **Isolation:** Each service involved in a Saga operates in isolation, without knowledge of other services' internal state

Answer: B), C), D)

Explanation:

- **A) Incorrect.** Sagas do not guarantee atomicity in the traditional ACID sense. They achieve eventual consistency through compensation.
- **B) Correct.** Sagas are specifically designed for long-running business processes that involve multiple distributed services.
- **C) Correct.** The ability to define and execute compensation transactions is a fundamental aspect of the Saga pattern, allowing for rollback of partial failures.
- **D) Correct.** While services communicate, they maintain their autonomy and do not expose their internal state to other services, adhering to the principles of microservices.

Q#47. In a Choreography-based Saga, how do services typically communicate to coordinate the distributed transaction? *(Concept-based, Single-select)*

A) Through a centralized orchestrator service.
B) By directly calling each other's REST APIs synchronously.
C) By publishing and subscribing to events via a message broker.
D) By sharing a common database and using database transactions

Answer: C)

Explanation:

- **A) Incorrect.** This describes an Orchestration-based Saga.
- **B) Incorrect.** Synchronous calls can lead to tight coupling and cascading failures, which Sagas aim to avoid.
- **C) Correct.** In a Choreography-based Saga, each service publishes events upon completing its part of the transaction, and other services react to these events to perform their subsequent steps, leading to a decentralized coordination.
- **D) Incorrect.** Sharing a common database violates microservices principles and can lead to tight coupling.

Q#48. Which of the following are benefits of deploying Java microservices on Kubernetes? *(Concept-based, Multi-select)*

A) **Automated Scaling:** Kubernetes can automatically scale microservice instances based on demand.
B) **Self-Healing:** Kubernetes can detect and restart failed containers, ensuring high availability.
C) **Service Discovery and Load Balancing:** Kubernetes provides built-in mechanisms for service discovery and load balancing.
D) **Simplified Database Management:** Kubernetes automatically manages database backups and replication.

Answer: A), B), C)

Explanation:

- **A) Correct.** Kubernetes supports horizontal pod autoscaling, allowing it to automatically adjust the number of microservice instances based on CPU utilization or custom metrics.
- **B) Correct.** Kubernetes constantly monitors the health of containers and can automatically restart or reschedule failed ones, contributing to the self-healing capabilities of the system.
- **C) Correct.** Kubernetes has its own DNS- based service discovery and can distribute traffic among healthy pods for a given service.
- **D) Incorrect.** While Kubernetes can orchestrate database containers, it does not automatically manage database backups or replication; these are typically handled by separate database management solutions or operators.

Q#49. Below are three statements about the Strangler Pattern in microservices:

1. The Strangler Pattern is used to add new features to a microservice.
2. It helps in gradually refactoring a monolithic system into microservices.

3. The Strangler Pattern requires rewriting the entire application at once.

Which of the following is the correct option? *(Concept-based, Single-select)*

- A) Statements 1 & 3 are correct
- B) Statements 2 & 3 are correct
- C) Statement 2 is correct
- D) All statements are correct

Answer: C)

Explanation:

- The Strangler Pattern is used for migrating from a monolithic architecture to microservices, not just for adding new features (Statement 1 is incorrect).
- It allows for gradual refactoring, replacing parts of the monolith with microservices over time (Statement 2 is correct).
- This approach does not require rewriting the entire application at once (Statement 3 is incorrect).

Q#50. How do Prometheus and Grafana work together for monitoring microservices? *(Concept-based, Single-select)*

A) Prometheus provides alerts based on predefined thresholds, while Grafana visualizes these alerts.
B) Grafana collects and stores metrics, while Prometheus visualizes these metrics.
C) Prometheus and Grafana both perform data collection and storage independently.
D) Prometheus collects and stores metrics from microservices, while Grafana visualizes these metrics.

Answer: D)

Explanation:

- **A) Incorrect.** Prometheus provides alerting capabilities, but Grafana is used for visualization and does not handle alerts directly.
- **B) Incorrect.** Grafana does not collect metrics; it relies on Prometheus or other sources for this data.
- **C) Incorrect.** Prometheus and Grafana are not independent in terms of data collection; Prometheus handles this, and Grafana focuses on visualization.
- **D) Correct.** Prometheus is responsible for collecting and storing metrics from various sources, while Grafana is used to create dashboards and visualize these metrics.

📗 Want to Master Java Microservices?

You have just explored a few of Java Microservices interview questions, MCQs with explanations. But this is just the beginning!

🎓 Join Udemy Course for Practicing Java Microservices

- 🎯 **Practice Mode** to learn interactively with instant feedback

- ⏱️ **Exam Mode** to simulate real-time interviews or assessments

- ✅ **~400 Questions with Answers & detailed explanations**
- ✅ **Covers all concepts of Java Microservices**
- ✅ **Concept-based, Scenario-based, Hands-On Coding-based Questions**
- ✅ **Interview-Focused Content for all interviews**
- ✅ **Includes Dual (Practice & Exam) Mode for complete preparation.**
- ✅ **Lifetime Access with Certification**
- ✅ **Learn Anytime, Anywhere**
- ✅ **Community Q&A + Instructor Support**
- ✅ **Value for Money**

🧪 Practice Mode vs 🧠 Exam Mode (Included in the Course!)

Feature	Practice Mode	Exam Mode
Purpose	Learn by solving questions at your pace	Test your knowledge under real-time pressure
Timing	No time limit	Time-bound (simulates real exams)
Feedback	Immediate answer explanations	Feedback after completing the set
Flexibility	Pause, retry, and review	One-shot attempt like in real exams
Best For	Beginners and revision sessions	Interview preparation and assessment

👉 Enroll Now:

Java Microservices Interview Practice Test & Interview Questions.

📩 For support or queries, reach out at: javatechonline@gmail.com

📌 You may also check complete tutorial with concepts & examples on Java Microservices by visiting Java Microservices Tutorial.

🎯 Practice Set for Other Java Related Technologies: **Java Question Hub**

🧠 *Keep learning. Keep building. Master patterns like a pro.*

Topics Covered in the Practice Tests

Below are the topics which are covered in the Java Microservices Practice Tests & Interview Questions on Udemy.

1. Key principles of Microservices
2. Monolithic vs Microservices
3. gRPC for efficient binary communication
4. API Gateway pattern
5. Asynchronous communication (Message Queues like Kafka, RabbitMQ)
6. OpenFeign for REST clients
7. Load balancing strategies (Ribbon, OpenFeign Load Balancer)
8. Database per service pattern
9. Event sourcing and CQRS patterns
10. Handling distributed transactions (Sagas, Two-Phase Commit)
11. Data consistency (eventual consistency, strong consistency)
12. Caching mechanisms (Redis, Memcached)

13. Circuit Breaker Pattern (e.g., using Resilience4j)
14. Retry mechanisms
15. Bulkhead pattern
16. Timeout handling
17. Distributed tracing (Zipkin, Sleuth)
18. Docker for containerization of microservices
19. Service discovery with Eureka or Consul
20. Load balancing strategies (Client-side, Server-side)
21. API Gateway patterns (Netflix Zuul, Spring Cloud Gateway)
22. JWT (JSON Web Token)
23. Role-Based Access Control (RBAC)
24. Security practices for inter-service communication
25. Distributed logging (ELK Stack: Elasticsearch, Logstash, Kibana)
26. Monitoring (Prometheus, Grafana)
27. Distributed tracing (Zipkin, Jaeger)
28. Health checks (Spring Boot Actuator)
29. Jenkins or GitLab for Continuous Integration
30. Horizontal vs vertical scaling
31. Stateless vs stateful services
32. Autoscaling in Kubernetes
33. Load testing and tuning (Apache JMeter)
34. Event streaming with Kafka
35. Event Sourcing and Command Query Responsibility Segregation (CQRS)
36. Async messaging patterns