**MICROSERVICES**

1. **What is microservices?**

Microservices is an architectural style that structures an application as a collection of small, independent, and loosely

1 coupled services. Each service focuses on a specific business capability, communicates with other services over a network (often using lightweight protocols like HTTP/REST or gRPC), and can be developed, deployed, scaled, and maintained independently**.**

1. **Design patterns of ms?**

**Decomposition Patterns:**

Decompose by Business Capability: Breaking down the application based on business functions (e.g., Customer Service, Order Management, Inventory).

Decompose by Subdomain: Dividing based on logical sub-areas within a business domain.

**Integration Patterns:**

API Gateway: A single entry point for clients, routing requests to appropriate services.

Backend for Frontends (BFF): Creating separate backend services tailored to specific UI experiences.

Message Broker: Using message queues (e.g., Kafka, RabbitMQ) for asynchronous communication.

**Database Patterns:**

Database per Service: Each service owns its own database.

Shared Database (Anti-pattern): Multiple services share the same database (generally discouraged).

Saga: Managing distributed transactions through a sequence of local transactions and compensating transactions.

CQRS (Command Query Responsibility Segregation): Separating read and write operations.

**Observability Patterns:**

Logging: Centralized and structured logging across services.

Metrics: Collecting and monitoring performance metrics (CPU, memory, latency).

Distributed Tracing: Tracking requests across multiple services.

Health Checks: Exposing endpoints to determine the health of each service.

**Resilience Patterns:**

Circuit Breaker: Preventing cascading failures by stopping requests to a failing service for a period.

Retry: Automatically retrying failed requests.

Bulkhead: Isolating resources to prevent failures in one service from impacting others.

**Deployment Patterns:**

Containerization (Docker): Packaging services into containers for portability and isolation.

Orchestration (Kubernetes, Docker Swarm): Managing and scaling containers.

1. **Communication between ms?**

**Synchronous Communication:**

* 1. **REST (HTTP/JSON):** The most common approach, using standard HTTP methods (GET, POST, PUT, DELETE) and JSON for data exchange. Simple to implement but can lead to tight coupling and blocking.
  2. **gRPC:** A high-performance, language-agnostic RPC framework using Protocol Buffers for serialization. More efficient than REST but can have a steeper learning curve.

**Feign Client: <artifactId>spring-cloud-starter-openfeign</artifactId>**

**No**, you **do not need** @RestController for the **Feign client itself**. It is just used to communicate with other service, no Http endpoints.

**Asynchronous Communication:**

* 1. **Message Queues (e.g., RabbitMQ, Kafka):** Services send and receive messages without requiring an immediate response. Improves decoupling, resilience, and scalability.
  2. **Event Streaming (e.g., Kafka):** Services publish and subscribe to streams of events, enabling real-time data sharing and reactive architectures.

The choice of communication depends on factors like the need for immediate responses, coupling tolerance, scalability requirements, and complexity.

1. **How to handle failure of ms?**

**Idempotency**: Designing operations so that making the same request multiple times has the same effect as making it once. This helps in handling retries.

**Use an Idempotency Key**

* The client sends a unique Idempotency-Key header with the request.
* The server stores the result of the first request.
* If the same key is received again, the server returns the **cached response**.

**Timeouts and Retries**: Implementing timeouts for requests and automatically retrying failed requests (with exponential backoff and jitter).

**Circuit Breaker**: Preventing cascading failures by stopping requests to a failing service for a period and allowing it to recover.

**@CircuitBreaker(name = "productService", fallbackMethod = "getProductDetailsFallback")**

**Sagas:** For distributed transactions spanning multiple services, use a saga pattern. This involves a sequence of local transactions in each service, with compensating transactions to rollback changes if a later transaction fails.

Choreography-based Saga: Each service listens for events from others and reacts accordingly.

Orchestration-based Saga: A central orchestrator service manages the flow of transactions.

**Eventual Consistency**: Accepting that data might be temporarily inconsistent across services. Focus on ensuring that data will eventually become consistent. This is often used with asynchronous communication.

Transaction Log Tailing: One service listens to the transaction log of another service's database to maintain eventual consistency.

**Outbox Pattern**: When a service needs to update its database and publish an event, it does both within a single local transaction. A separate process then reads the "outbox" table and publishes the events to a message broker.

1. **Challenges with ms?**

 **Increased Complexity:** Managing a distributed system with many moving parts is inherently more complex than a monolith.

 **Operational Overhead:** Deploying, monitoring, and managing numerous services requires significant operational effort.

 **Distributed Debugging:** Tracing requests and debugging issues across multiple services can be difficult.

 **Network Latency:** Communication between services over a network introduces latency.

 **Data Consistency:** Maintaining data consistency across independent databases is challenging.

 **Testing:** Testing a distributed system requires more sophisticated strategies (integration tests, contract tests).

 **Security:** Securing inter-service communication and managing authentication/authorization across services is complex.

 **Service Discovery:** Services need a mechanism to find and communicate with each other.

 **Configuration Management:** Managing configuration across multiple services can be challenging.

 **Deployment Complexity:** Orchestrating the deployment of multiple independent services requires robust tooling.

1. **Monolithic vs ms vs service**

**A screenshot of a computer

AI-generated content may be incorrect.**

1. **Role of API gateway**

An API Gateway acts as a single entry point for all client requests to the microservices. Its responsibilities include:

* **Request Routing**: Directing incoming requests to the appropriate backend microservice.
* **Authentication and Authorization**: Verifying the identity of clients and ensuring they have permission to access the requested resources.
* **Rate Limiting and Throttling**: Protecting backend services from being overwhelmed by excessive requests.
* **Request/Response Transformation**: Modifying request headers, bodies, or response formats as needed.
* **Protocol Translation**: Handling different client protocols (e.g., HTTP, WebSocket) and translating them to the protocols used by backend services.
* **Caching:** Caching frequently accessed responses to reduce load on backend services.
* **Observability:** Collecting metrics and logs for API traffic.

The API Gateway simplifies the client-side interaction with the microservices architecture by providing a unified interface.

1. **How to handle transaction in ms**

**Two-Phase Commit (2PC):** A distributed transaction protocol that aims for atomicity across multiple resource managers. However, it can suffer from performance and availability issues (blocking). Not widely used in modern microservices.

**Sagas (as mentioned in point 4):** A more common approach that breaks down a distributed transaction into a sequence of local transactions, with compensating transactions to handle failures.

* Choreography: Services publish events that trigger actions in other services.
* Orchestration: A central service (the orchestrator) manages the steps of the transaction.

**Outbox Pattern (as mentioned in point 4):** Ensures that database updates and event publishing happen atomically within a single local transaction.

Best Effort 1PC: Each service performs its local transaction and makes a best effort to inform other relevant services about the outcome. This relies on retries and idempotency to handle failures and eventual consistency**.**

1. **What is service discovery and why is it important**

Service discovery is the process by which services in a microservices architecture can automatically locate and communicate with each other without hardcoding network locations (IP addresses and ports).

1. **Circuit breaker**

**Closed State**: When the service is healthy, requests are allowed through.

**Open State**: If the number of failures exceeds a certain threshold within a specific time window, the circuit breaker "opens," and subsequent requests are immediately failed (without even trying to call the failing service). This prevents the client from wasting resources and potentially overloading the failing service.

**Half-Open State**: After a timeout period in the open state, the circuit breaker transitions to a "half-open" state. In this state, it allows a limited number of "test" requests to try and reach the service. If these requests are successful, the circuit breaker closes; otherwise, it goes back to the open state.

1. **How to log activity across multiple microservices**

 **Centralized Logging:** Sending logs from all services to a central logging system (e.g., ELK stack - Elasticsearch, Logstash, Kibana; or Grafana Loki). This allows you to aggregate and analyze logs from all parts of the system in one place.

 **Structured Logging:** Logging in a structured format (e.g., JSON) makes it easier to parse and query logs.

 **Correlation IDs:** Assigning a unique ID to each incoming request and propagating this ID across all subsequent service calls. This allows you to trace a single user request through the entire system. The correlation ID should be included in all logs related to that request.

 **Log Levels:** Using appropriate log levels (DEBUG, INFO, WARN, ERROR) to control the verbosity of the logs.

 **Contextual Information:** Including relevant context in logs, such as service name, instance ID, request details, and user information.

1. **What tool to use for distributed tracing**

Zipkin (OpenZipkin): Another popular open-source distributed tracing system, originally developed by Twitter.

1. **Security in ms, inter ms security and external security**

**External Security (Securing Client Access to the System):**

* **Authentication**: Verifying the identity of the client (e.g., using OAuth 2.0, JWT). An API Gateway often handles this.
* **Authorization**: Determining what resources the authenticated client is allowed to access (e.g., using RBAC, ABAC). This can be enforced at the API Gateway and/or within individual services.
* **HTTPS**: Encrypting communication between clients and the API Gateway.

**Inter-Microservice Security (Securing Communication Between Services):**

* **Mutual TLS (mTLS):** Each service authenticates the identity of the other using digital certificates, ensuring that communication is between trusted services and is encrypted.
* **JWT (JSON Web Tokens):** Propagating security context (user identity and permissions) as signed tokens in HTTP headers or message payloads. Services can verify these tokens to authorize requests from other services.
* **Network Segmentation:** Isolating microservices within secure network segments to limit the impact of a security breach.
* **Service Mesh (e.g., Istio, Linkerd):** Provides built-in security features like mTLS, traffic encryption, and authorization policies.
* **Dedicated Authentication/Authorization Service:** A central service responsible for issuing tokens and managing permissions.

1. **Asynchronous communication**

Asynchronous communication involves services sending messages to each other without expecting an immediate response. This is typically achieved using message brokers (e.g. Kafka) or event streaming platforms**.**

1. **How to handle versioning of ms api**

**A screenshot of a chat

AI-generated content may be incorrect.**

1. **Best practices for implementing ms at scale**

 **Decoupling:** Design services to be as independent as possible.

 **Automation:** Automate everything – build, test, deploy, monitoring, scaling. Use CI/CD pipelines.

 **Observability:** Implement robust logging, metrics, and distributed tracing.

 **Resilience:** Design for failure using patterns like circuit breakers, retries, and bulkheads.

 **Stateless Services:** Make services stateless whenever feasible to improve scalability. Store state in external data stores.

 **Idempotency:** Ensure operations are idempotent.

 **Service Discovery:** Implement a reliable service discovery mechanism.

 **API Gateway:** Use an API Gateway for a single entry point and cross-cutting concerns.

 **Containerization and Orchestration:** Use Docker and Kubernetes (or similar) for packaging and managing services.

 **Database per Service:** Favor database per service to ensure isolation.

 **Polyglot Persistence:** Choose the right database technology for each service's needs.

 **Contract Testing:** Ensure that services can communicate correctly by testing the contracts between them.

 **Security:** Implement security at every layer (authentication, authorization, encryption).

 **Team Autonomy:** Empower small, independent teams to own and operate their services.

 **Continuous Learning:** The microservices landscape is constantly evolving, so continuous learning is crucial.

1. **How to manage data in ms when diff services uses different db(polygot persist)**
2. **How to deploy multiple ms in Jenkins**

**Version Control (e.g., Git):** Each microservice should have its own repository (or potentially reside in a monorepo with clear separation).

**Jenkins Pipeline:** Define a Jenkins Pipeline (using Declarative or Scripted syntax) for each microservice.

**Build Stage:**

* Checkout the code from the respective Git repository.
* Build the microservice (e.g., using Maven, Gradle for Java; npm build for Node.js).
* Run unit tests.
* Build the Docker image for the microservice.
* Push the Docker image to a container registry (e.g., Docker Hub, AWS ECR, Google GCR).

Test Stage (Optional but Recommended):

* Deploy the new Docker image to a testing environment (e.g., using Kubernetes or Docker Compose).
* Run integration tests and contract tests against the deployed service and its dependencies.

Deploy Stage:

* Deploy the new Docker image to the production environment. This might involve:
  + Updating the Kubernetes deployment with the new image tag.
  + Using Docker Compose to pull and restart containers.
  + Using other deployment tools specific to your infrastructure.

**Rollback Strategy:** Implement a rollback mechanism to revert to the previous version in case of deployment failures.

1. **Spring cloud config**

Spring Cloud Config provides centralized externalized configuration management for distributed applications (like microservices). It allows you to store configuration for all your services in a central repository (e.g., Git, HashiCorp Vault, native filesystem) and provides a way for services to retrieve their configuration at startup or runtime.

A screenshot of a computer

AI-generated content may be incorrect.

1. **Ms health monitoring**

 **Health Check Endpoints:** Each microservice exposes a dedicated endpoint (e.g., /actuator/health in Spring Boot Actuator) that reports its health status. This endpoint typically checks the status of critical dependencies (e.g., database connections, message brokers).

 **Centralized Monitoring Systems:** Tools like Prometheus, Grafana, Nagios, Zabbix, and cloud-specific monitoring services (AWS CloudWatch, Azure Monitor, Google Cloud Monitoring) can periodically poll these health check endpoints.

 **Alerting:** Configure alerts in the monitoring system to notify operations teams when a service becomes unhealthy.

 **Log Analysis:** Monitoring logs for error patterns and exceptions can also indicate health issues.

 **Distributed Tracing:** Analyzing trace data can reveal performance bottlenecks and failures in the call chain.

1. **How to optimize performance of ms**
2. **How to implement load balancing and autoscaling**

**A screenshot of a computer program

AI-generated content may be incorrect.**

1. **Rest Call in ms**

* . A client sends a POST request to the /orders endpoint of the **Order Service**, providing the productId and quantity.
* The **OrderController i**n the Order Service calls the **createOrder** method in the OrderService.
* The OrderService needs product details, so it calls the **getProductDetails** method of the **ProductServiceClient**.
* The ProductServiceClient uses RestTemplate to make a synchronous HTTP GET request to the /products/{productId} endpoint of the Product Service.
* The **Order** Service waits for the Product Service to respond.
* The Product Service receives the request, retrieves the product details from its database (in this example, an in-memory HashMap), and sends back an HTTP response containing the Product object in JSON format.
* The **RestTemplate** in the Order Service receives the response.
* The **ProductServiceClient** processes the response and returns the Product object to the **OrderService**.
* The **OrderService** then calculates the **totalPrice** and creates the Order object.
* Finally, the **OrderController** sends an HTTP response back to the client, indicating the success or failure of the order creation.