**MicroServices  
  
1. What are Microservices?**

* Microservices architecture divides an application into small, loosely coupled services.
* Each service performs a specific business function and can be developed, deployed, and maintained independently.
* Services communicate with each other via APIs using lightweight protocols.
* Supports use of different languages and technologies per service.

**2. When and Why Use Microservices?**

* Ideal for large or complex applications requiring scalability and flexibility.
* Supports independent development and deployment of services, speeding up delivery.
* Useful to modernize or optimize monolithic applications.
* Enables cloud-native development and efficient resource utilization.
* Not suitable for simple or small applications where monolithic architecture suffices.

**3. Java Microservices Architecture Overview**

* Divides app into small processes/sub-processes, each with its own functionality.
* Processes communicate via lightweight protocols (e.g., REST, messaging).
* Enhances scalability, maintainability, and service coordination**.**

**4. Pros and Cons of Java Microservices**

**Pros:**

* Technology diversity (different languages, frameworks per service).
* Independent development, deployment, and scaling.
* Better fault isolation; failure in one service doesn’t affect others.
* Enhanced security by isolating services

**.**

Cons:

* Complex inter-service communication.
* Managing many services can be challenging.
* Increased configuration and infrastructure overhead.
* Difficulties with distributed transactions and security management.
* Network latency and maintenance complexity.

**5. Main Features of Java Microservices**

* Deployment: Independent deployment of small services.
* Decentralization: Each service manages its own database/data.
* Loosely Coupled: Failure in one service doesn’t impact others.
* Security: Role-based access control (RBAC) and service-level security.
* Scalability: Services can be scaled independently.

**6. What is Monolithic Architecture?**

* Single application with all modules tightly integrated in one codebase.
* Deployed as a single unit (e.g., WAR file on a server like Tomcat).
* Simple to develop initially but harder to scale and maintain as app grows.

**7. What is SOA (Service-Oriented Architecture)?**

* Collection of loosely coupled services communicating over standardized protocols (e.g., SOAP).
* Designed for reusability and integration of services across large enterprises.
* Typically heavier and more complex than microservices.

**9. Design Patterns in Java Spring Boot Microservices**

* Service Registry & Discovery: Services register with central registry (e.g., Eureka) for dynamic discovery.
* API Gateway: Single entry point managing routing, authentication, rate limiting, etc.
* Circuit Breaker: Monitors service availability, handles failures gracefully (e.g., Hystrix).
* CQRS (Command Query Responsibility Segregation): Separate models for read/write operations.
* Saga Pattern: Manages distributed transactions through a series of local transactions.
* Database per Service: Each service owns its own database for isolation and scalability.
* Asynchronous Messaging: Services communicate via message brokers (Kafka, RabbitMQ) for loose coupling.

**10. Main Components of Java Spring Boot Microservices**

* Services: Core business logic implemented as microservices.
* Service Registry: Centralized registry for service discovery.
* API Gateway: Entry point to route client requests to services.
* Cloud Infrastructure: Cloud platforms (AWS, Azure, GCP) hosting services.
* Containerization & Orchestration: Tools like Docker and Kubernetes for deployment and management.
* Message Broker: Middleware for asynchronous communication between services.
* Security: Authentication, authorization, and encryption mechanisms.
* Monitoring: Tools to monitor service health, performance, and logs.

**11. Common Tools for Spring Boot Microservices**

* **Docker:** Containerization.
* **Kubernetes:** Container orchestration.
* **Spring Cloud:** Framework for microservice patterns (service discovery, config, etc.)

**12. Communication Between Microservices**

* HTTP/REST (most common, lightweight)
* Message Queues (Kafka, RabbitMQ)
* RPC frameworks (gRPC)

**13. Request/Response Processing**

* Use JSON (preferred) or XML as data exchange formats.
* Both services must agree on format.

**14. WebClient (Spring 5 Reactive)**

* Non-blocking, reactive HTTP client.
* Replacement for RestTemplate.
* Used to call other services asynchronously.

**15. RestTemplate**

* Synchronous REST client.
* Simple and widely used but blocking.

**16. FeignClient (Spring Cloud OpenFeign)**

* Declarative REST client.
* Interface-based with annotations.
* Simplifies calling REST endpoints.

**17. Client-Side Load Balancing**

* Client decides which service instance to call.
* Spring Cloud LoadBalancer used.
* Use LoadBalancerClient and RestTemplate.

**18. Load Balancing with Netflix Ribbon**

* Client-side load balancer by Netflix.
* Integrates with FeignClient.
* Requires adding Ribbon dependencies and configuring servers.

**19. Eureka Server & Client Communication**

* Service discovery mechanism.
* Eureka Server registers service instances.
* Clients query Eureka to find services.
* Configured via eureka.client.service-url.defaultZone.

**20. API Gateway Development**

* Use Spring Cloud Gateway.
* Provides routing, security, filtering.
* Supports circuit breakers and service discovery integration.

**21-23. Service Discovery Types**

* **Client-Side:** Client queries registry (Eureka, Zookeeper, Consul).
* **Server-Side:** Load balancer or gateway routes traffic (NGINX, AWS ELB).

**24. Why Use Spring Cloud?**

* Offers tools for config management, service discovery, load balancing, circuit breakers.
* Simplifies building cloud-native microservices.

**26. Why Choose Spring Boot for Microservices?**

* Embedded server
* Auto-configuration & dependency management
* Annotation-driven, minimal boilerplate
* Open source, active community
* Supports loose coupling & cloud-native apps

**27. Circuit Breaker Pattern**

* Prevents failures cascading through system.
* States: Closed, Open, Half-Open.
* Ensures fault tolerance, automatic recovery, fallback mechanisms.

**28. Deployment Techniques**

* Single machine multiple services
* Multiple machines multiple services (scale horizontally)
* Containerization (Docker)
* Orchestration (Kubernetes)
* Serverless (AWS Lambda, etc.)

**29. Role of Docker & Deployment Steps**

* Containerize each microservice.
* Build images & push to registry.
* Deploy via Docker Compose or Kubernetes.
* Monitor and manage containers.

**30. Deploying Spring Boot Microservices on AWS**

* Use EC2 instances and Application Load Balancer.
* Create ECS clusters and tasks.
* Configure repositories and networking.
* Use AWS tools for orchestration and scaling.