**Distributed Systems and Microservices (General)**

1. **What is a distributed system?** A distributed system is a collection of independent computers that appears to its users as a single coherent system. These systems work together to achieve a common goal. Each node in the system is autonomous, and the nodes communicate via a network to coordinate actions. For example, in a ride-sharing application, one service may manage user authentication, another may handle ride requests, and another may track real-time location. Each of these services is deployed on separate servers, making the system distributed. Challenges include maintaining consistency, handling failures, and coordinating state.
2. **What are the advantages of microservices architecture?** Microservices architecture breaks down an application into smaller, loosely coupled, and independently deployable services. Key benefits include:
   * **Scalability:** Individual services can scale independently.
   * **Flexibility in Tech Stack:** Teams can choose the best tool for the job.
   * **Fault Isolation:** A failure in one service doesn't necessarily bring down the whole system.
   * **Independent Deployment:** Faster release cycles and reduced deployment risk. For example, an e-commerce platform may have separate microservices for product catalog, payment, and shipping, each developed and deployed independently.
3. **What challenges do you face in distributed systems?** Distributed systems come with several challenges:
   * **Network latency and partitions**
   * **Data consistency**
   * **Service discovery**
   * **Monitoring and logging**
   * **Security and authorization** For example, if a payment microservice goes down during a transaction, retry logic and proper compensation mechanisms must be in place to ensure user trust.
4. **How do you ensure resiliency in a distributed system?** Resiliency refers to the ability of the system to recover from failures. Techniques include:
   * **Circuit Breakers** (e.g., Hystrix)
   * **Retries with exponential backoff**
   * **Timeouts and bulkheads**
   * **Graceful degradation** For instance, if a weather API fails in a travel booking system, the application can fall back to a cached result or show a default message.
5. **Explain eventual consistency.** Eventual consistency is a consistency model used in distributed systems to achieve high availability. It guarantees that if no new updates are made to a given data item, eventually all accesses will return the last updated value. For example, in DynamoDB or Cassandra, writes are eventually propagated to all replicas, so reads might return stale data temporarily, but the system will converge.
6. **What is service discovery in microservices?** Service discovery is the process by which a microservice client automatically detects the network locations of service instances. There are two main types:
   * **Client-side discovery:** The client queries a service registry to get available instances.
   * **Server-side discovery:** A load balancer queries the registry and forwards the request. Tools like Consul, Eureka, and Kubernetes' internal DNS help in service discovery. Example: In Netflix OSS, Eureka is used for service registration and discovery.
7. **What is the difference between synchronous and asynchronous communication in microservices?**
   * **Synchronous communication** is where services communicate in real-time, usually via HTTP REST APIs. The client waits for a response.
   * **Asynchronous communication** involves sending messages through a queue or message broker (e.g., Kafka, RabbitMQ) without waiting for an immediate response. Example: When a user places an order, the order service may synchronously call the payment service but asynchronously notify the inventory service.
8. **What is a circuit breaker pattern and why is it important?** The circuit breaker pattern prevents an application from trying to invoke a remote service or a piece of code that's likely to fail. It has three states: **closed**, **open**, and **half-open**.
   * **Closed:** Everything works as normal.
   * **Open:** Calls are not made to the failing service.
   * **Half-open:** A limited number of test requests are allowed. Example: Netflix's Hystrix library implements this pattern to avoid cascading failures.
9. **How do you monitor and debug microservices in production?** Observability in microservices requires:
   * **Centralized logging** (e.g., ELK stack)
   * **Distributed tracing** (e.g., OpenTelemetry, Jaeger, Zipkin)
   * **Metrics collection** (e.g., Prometheus + Grafana)
   * **Health checks** and alerting Example: You use Prometheus to track service metrics and Grafana to visualize performance dashboards. Jaeger traces a request from the API Gateway through multiple services.
10. **What is the role of an API Gateway in microservices?** An API Gateway sits between clients and services. It provides:

* **Routing and composition**
* **Authentication and authorization**
* **Rate limiting and throttling**
* **Caching and logging** Example: In a shopping app, the API Gateway receives a user request, validates the token, and routes the call to appropriate backend services such as catalog and user profile services.

**Microservices One-Word Answers / Basics**

1. **Microservices design pattern – Decomposition**
2. **Microservices communication method – HTTP**
3. **Common API style in microservices – REST**
4. **Message format for microservices – JSON**
5. **Inter-service communication type – Synchronous**
6. **Async communication method – Messaging**
7. **Distributed tracing tool – Jaeger**
8. **Containerization platform – Docker**
9. **Container orchestration tool – Kubernetes**
10. **Service discovery tool – Consul**
11. **API Gateway tool – Kong**
12. **Service mesh tool – Istio**
13. **Decentralized configuration management – Spring Cloud Config**
14. **Database type per service – Polyglot**
15. **Event-driven architecture tool – Kafka**
16. **Load balancing strategy – RoundRobin**
17. **Database per service approach – Decentralized**
18. **Circuit breaker pattern – Hystrix**
19. **Centralized logging tool – ELK Stack**
20. **Centralized monitoring tool – Prometheus**
21. **Microservices security standard – OAuth**
22. **Microservices authentication protocol – JWT**
23. **Microservices authorization tool – RBAC**
24. **Inter-service failure tolerance pattern – Retry**
25. **Scaling microservices automatically – Horizontal**
26. **Microservices testing strategy – ContractTesting**
27. **Distributed database type – NoSQL**
28. **CI/CD tool for microservices – Jenkins**
29. **Versioning strategy for APIs – Semantic**
30. **Fault tolerance pattern – Fallback**
31. **Deployment strategy for microservices – Canary**
32. **API rate limiting tool – RateLimiter**
33. **Microservices gateway pattern – API Gateway**
34. **Asynchronous processing tool – RabbitMQ**
35. **Event sourcing tool – EventStore**
36. **Key-value store for microservices – Redis**
37. **Single responsibility principle – SRP**
38. **Two-phase commit for distributed systems – 2PC**
39. **Synchronous call timeout tool – Timeout**
40. **Service resilience tool – Resilience4J**
41. **API documentation standard – OpenAPI**
42. **Microservices lifecycle tool – Helm**
43. **Microservices orchestration tool – Docker Swarm**
44. **Microservices routing tool – Nginx**
45. **Scaling microservices approach – AutoScaling**
46. **Service failure isolation strategy – Bulkhead**
47. **Microservices pattern for shared data – SharedDatabase**
48. **Event-driven architecture protocol – Kafka**
49. **Multi-cloud microservices deployment – Hybrid**
50. **Microservices testing type – EndToEnd**

**Async communication method – Messaging**

**Explanation:  
In microservices, asynchronous communication allows services to send messages to each other without waiting for a response. This improves system resilience and decouples services. The two common ways to achieve this are:**

* **Message Queues (e.g., RabbitMQ): Services can send messages to a queue, and other services can consume messages when they are ready.**
* **Event Streaming (e.g., Kafka): This allows services to publish events to topics, and other services can subscribe and react to those events.**

**Example:**

* **RabbitMQ: A service sends a message to a queue ("orderQueue") to notify the inventory service that an order has been placed. The inventory service reads this message and updates the stock asynchronously.**
* **Kafka: A service publishes an event (e.g., "OrderPlaced") to a Kafka topic. The downstream services can subscribe to this topic and act accordingly.**

**Benefit: This decouples services, allowing them to process requests at their own pace and ensures that services are resilient even if other services are temporarily unavailable.**

**2. Distributed tracing tool – Jaeger**

**Explanation:  
Distributed tracing is used to track requests as they flow through a distributed system. It helps monitor and debug microservices by providing a visual representation of how requests traverse across various services. Jaeger is an open-source distributed tracing tool used to trace and monitor applications, helping developers understand how requests are processed and identify performance bottlenecks.**

**Example:**

* **A user makes an HTTP request that triggers multiple microservices. Jaeger traces the request from the API Gateway through each microservice, showing the time each microservice took to process the request. This helps identify slow services or steps in the workflow.**

**Benefit: Jaeger helps identify latency issues, service dependencies, and helps optimize system performance by providing insights into where requests are slow or failing.**