✅ Microservices Roadmap for Java Full Stack Developers

🔹 Module 1: Core Concepts & Architecture

1. Introduction to Microservices ✅
2. Monolithic vs Microservices Architecture ✅
3. Advantages and Drawbacks of Microservices
4. Microservices Design Principles (Single Responsibility, Bounded Context)
5. Communication between Microservices (REST, Messaging, gRPC)
6. Synchronous vs Asynchronous Communication
7. Service Registry and Discovery (Eureka)
8. API Gateway (Spring Cloud Gateway, Zuul)
9. Load Balancing (Ribbon, Spring Cloud LoadBalancer)
10. Circuit Breaker Pattern (Resilience4j, Hystrix)
11. Fault Tolerance and Retry Mechanisms
12. Rate Limiting and Throttling

🔹 Module 2: Spring Boot + Microservices Implementation

13. Creating Microservices with Spring Boot
14. Project Structure for Microservices
15. REST Communication using RestTemplate and WebClient
16. Configuration Management using Spring Cloud Config
17. Centralized Logging with ELK Stack / Zipkin
18. Distributed Tracing with Sleuth + Zipkin
19. Event-Driven Microservices with Kafka / RabbitMQ

🔹 Module 3: Security & Best Practices

20. Securing Microservices with JWT & OAuth2

21. Role of API Gateway in Authentication & Authorization
22. Cross-Origin Resource Sharing (CORS)
23. Versioning in Microservices APIs
24. Exception Handling and Global Error Responses
25. DTO Pattern and Mapping Entities

- **Module 4: DevOps, Deployment & Testing**

26. Dockerizing Microservices
27. Introduction to Kubernetes (basic)
28. CI/CD Overview (Jenkins/GitHub Actions)
29. Unit Testing and Integration Testing for Microservices
30. Health Checks and Monitoring (Actuator, Prometheus, Grafana)

- **Final:**

✓ All Interview Questions in One Page (after completion)
✓ Real-life project examples with design patterns
✓ PDF for each topic available
✓ Step-by-step Spring Cloud implementation

✅ Microservices – Topic 1: Introduction to Microservices Architecture

---

🔹 What are Microservices?

Microservices is an architectural style where a large application is broken down into **smaller, independent services** that communicate with each other.

Each service:

- Is responsible for a **single business function**
- Can be **developed, deployed, and scaled** independently
- Communicates with others via **APIs (usually REST)**

🔹 Why Microservices?

Traditional applications were built using **monolithic architecture** (single codebase). It worked well for small apps but caused issues as they grew.

✅ Microservices solve these problems by:

- Making the system **modular**
- Allowing teams to work on **independent features**
- Scaling **only what's needed**

🔹 Real-Life Example

Take **Amazon** as an example:

- User Service – handles user accounts
- Product Service – handles product listings
- Payment Service – processes payments
- Notification Service – sends emails/SMS

Each of these is a **microservice** and can be built using different tech stacks.

🔹 Features of Microservices

Feature	Description
Independent Deployment	Each service can be deployed separately
Scalability	Only scale the services that need it
Fault Isolation	Failure in one service won't crash the entire system
Technology Flexibility	Use different languages/DBs for different services
Easier Maintenance	Small codebase = easier debugging & changes

---

🔹 **Communication in Microservices**

Services usually talk via:

- **REST APIs (HTTP/JSON)**
- **Messaging Queues (RabbitMQ, Kafka)**
- **gRPC (for internal fast communication)**

🔹 **Drawbacks of Microservices**

Drawback	Description
Complexity	Managing many services increases complexity
Distributed Debugging	Harder to trace issues across services
Network Latency	More network calls = more latency
Deployment Overhead	Requires DevOps maturity (CI/CD, Docker, Kubernetes)

✅ Microservices – Topic 2: Monolithic vs Microservices Architecture

---

🔹 1. Monolithic Architecture

In a **monolithic** system:

- The entire application is built as **one large unit**
- All modules (UI, business logic, DB access) are **tightly coupled**
- Deployed as a **single jar/war**

Example:

An e-commerce app where:

- Product listing
- Cart management
- Payments
- Order tracking
  ...are all in **one project** and **one database**.

🔹 2. Microservices Architecture

In **microservices**:

- Each business functionality is **separated into independent services**
- Every service has its own **code, database, deployment**
- Services communicate via **REST, messaging, gRPC**

Example:

E-commerce system split into:

- Product-Service
- Order-Service
- Cart-Service
- User-Service

Each can be deployed and scaled **individually**.

🔹 3. Key Differences Between Monolithic and Microservices

Aspect	Monolithic	Microservices
Codebase	Single codebase	Multiple small codebases
Deployment	Deployed as one unit	Each service is deployed independently
Scalability	Vertical scaling (whole app)	Horizontal scaling (only needed services)
Technology Stack	Same for entire app	Different services can use different techs
Fault Isolation	Entire app may fail	One service failing won't crash all
Dev Team Scaling	Hard to split among teams	Teams own separate services
Database	Often single shared DB	Services can have their own DBs

---

🔹 4. When to Use Monolith?

- Small applications
- Early-stage startups
- When speed of development matters more than scalability

🔹 5. When to Use Microservices?

- Large teams working in parallel
- Need for independent deployment
- Scaling individual components
- Real-time systems with high availability

🔹 6. Real Life Comparison

Scenario	Architecture
Social media platform with multiple features	Microservices

Scenario	Architecture
Basic blog website with login + articles	Monolith
E-commerce platform with global traffic	Microservices

✅ Microservices – Topic 3: Advantages and Drawbacks of Microservices

---

🔹 1. Advantages of Microservices

Microservices have gained popularity because they solve many real-world problems with large-scale applications. Here are the key benefits:

✅ a) Independent Development & Deployment

Each service is **developed, tested, and deployed independently**, reducing the risk of breaking the whole system.

Real-life: You can deploy a new version of PaymentService without affecting UserService or ProductService.

✅ b) Scalability

Services can be **scaled independently** based on demand.

Example: During sales, only ProductCatalogService and CartService are scaled up.

✅ c) Technology Flexibility

Different services can use **different languages, frameworks, or databases**.

Example: SearchService can use **Elasticsearch**, while others use **MySQL**.

✅ d) Better Fault Isolation

Failure in one service doesn't affect the others.

If ReviewService fails, the rest of the system still works.

✅ e) Faster Time to Market

Smaller teams working on separate services leads to **faster development and release cycles**.

✅ f) Organized Around Business

Each microservice is aligned to a **business capability**, not technical layer.

OrderService, InventoryService, ShippingService represent real business domains.

✅ g) Improved Maintainability

Smaller codebases are **easier to manage, refactor, and test**.

🔹 2. Drawbacks of Microservices

Microservices also come with challenges. These must be handled with care:

⚠️ a) Complexity

Managing multiple services, communication, and data consistency introduces **architectural complexity**.

⚠️ b) Distributed System Challenges

Includes:

- Network latency
- Failures in inter-service communication
- Versioning of APIs

⚠️ c) Deployment Overhead

You need proper DevOps tools and automation (CI/CD, Docker, Kubernetes) to manage microservices effectively.

⚠️ d) Data Management Complexity

Maintaining **data consistency** across services is harder, as each service might have its own database.

Example: Handling a single transaction across OrderService and PaymentService requires careful design (Saga Pattern, Eventual Consistency).

⚠️ e) Testing Is Difficult

Testing across multiple services requires **integration testing** with proper mocks, stubs, or test containers.

⚠️ f) Increased Resource Usage

Each service runs on its own process (containers), which can be **resource-intensive**.

⚠️ g) Operational Overhead

- Logging
- Monitoring
- Tracing
  All need to be **centralized and automated**.

✅ Microservices – Topic 4: Design Principles of Microservices

Designing microservices properly from the start is crucial. These principles help build **scalable, maintainable, and loosely coupled systems**.

🔹 1. Single Responsibility Principle (SRP)

Every microservice should do **one thing and one thing only** – a single business capability.

✅ Example:
OrderService should only handle orders – not payments, not inventory.

Benefits:

- Cleaner code
- Easier debugging
- Faster deployment of changes

🔹 2. Loose Coupling

Services should not depend heavily on each other. They must interact using **well-defined APIs** (REST/gRPC) instead of direct calls or shared data.

❌ Don't: Access another service's DB directly
✅ Do: Use REST APIs or Events for communication

🔹 3. High Cohesion

A microservice should keep related logic **together** — data, logic, rules, and operations that serve one purpose.

✅ CustomerService should handle:

- Customer creation
- Profile update
- Customer-specific rules

🔹 4. Database per Service

Each service should manage **its own database** to ensure independence and scalability.

❌ Don't use a shared database
✅ Each service uses its own DB schema

Example:

- UserService → PostgreSQL
- OrderService → MySQL
- SearchService → Elasticsearch

🔹 5. API-First Design

Design APIs **before** writing service logic. Use tools like:

- Swagger/OpenAPI
- Postman

This ensures that other teams can understand and interact with your service easily.

🔹 6. Statelessness

Services should not keep user sessions or temporary data in memory. Instead, use:

- JWT tokens for authentication
- Caching (Redis) for temporary data
- Databases for persistent data

Why? It helps with scalability and fault-tolerance.

🔹 7. Design for Failure

Failures will happen. Services should be:

- Retryable
- Timeout-enabled
- Protected by circuit breakers

Tools: Resilience4j, Hystrix

🔹 8. Observability

Your system should be **visible** through logs, metrics, and traces.

Use:

- Logs → ELK Stack
- Metrics → Prometheus
- Tracing → Zipkin or Jaeger

🔹 9. Versioning

When you make changes to a service's API, support **versioning** to avoid breaking old consumers.

Example:
/api/v1/users
/api/v2/users (with new response format)

🔹 10. Security First

Always ensure:

- HTTPS usage
- Input validation
- Authentication (JWT, OAuth2)
- Role-based access

✅ Microservices – Topic 5: Communication Between Microservices

In a microservices architecture, services are **independent**, but they still need to **talk to each other** to perform business operations. This communication happens through **network calls**.

There are **two main types**:

🔹 1. Synchronous Communication

👉 In this type, the caller waits for the response.

✅ **Protocols:**

- **REST (HTTP)**
- **gRPC**

🔸 a) REST (Representational State Transfer)

- Uses HTTP
- Easy to understand, widely supported
- Returns JSON or XML

Example:
OrderService calls PaymentService using a REST endpoint:
POST http://payment-service/pay

Tools:

- RestTemplate (older)
- WebClient (modern & reactive)

🔸 b) gRPC

- Uses HTTP/2 for faster communication
- Compact binary data using **Protocol Buffers**
- Useful when performance is critical or between services in different languages

Example:
Real-time stock trading system between microservices

🔹 2. Asynchronous Communication

👉 Caller **sends a message** and continues execution — doesn't wait for a response.

✅ **Message Brokers:**

- **Kafka**
- **RabbitMQ**
- **ActiveMQ**

🔸 **How It Works:**

- Services **publish** messages to a **topic or queue**
- Other services **subscribe** to that topic

Example:
OrderService publishes an event OrderPlaced to Kafka
InventoryService listens and processes it to reduce stock.

Benefits:

- Loose coupling
- Higher performance
- Reliable delivery

🔹 3. Choosing Between Synchronous & Asynchronous

Need	Best Choice
Immediate response needed	REST / gRPC
High throughput / decoupling	Kafka / RabbitMQ
Multi-language services	gRPC
Event-based workflows	Messaging

🔹 4. Real-Life Example:

Scenario: User places an order

Step	Communication Type	Description
Place Order	REST (Sync)	UI → OrderService
Payment Process	REST (Sync)	OrderService → PaymentService
Notify Inventory	Kafka (Async)	OrderService → InventoryService (via event)
Send Email	Kafka (Async)	OrderService → NotificationService (via event)

🔹 5. Tools for Communication

Type	Technology
REST	RestTemplate, WebClient
Messaging	Apache Kafka, RabbitMQ
gRPC	gRPC Java / Proto files

✅ Microservices – Topic 6: Synchronous vs Asynchronous Communication (Deep Comparison)

Understanding these two communication styles is **essential** in designing scalable and fault-tolerant microservice systems.

🔹 1. What is Synchronous Communication?

- The **caller waits** for a response from the callee before moving forward.
- Happens **in real-time**, usually using **HTTP/REST or gRPC**.

🔸 **Example:**

OrderService calls PaymentService via REST API → waits for payment response → then continues.

✅ **Pros:**

- Easy to implement
- Real-time results
- Easier to debug (request/response flow)

❌ **Cons:**

- Tight coupling
- One service's failure can **block** others
- Difficult to scale under load

🔹 2. What is Asynchronous Communication?

- The caller **sends a message/event** and **continues** immediately.
- The callee processes it **later**, usually via a **message queue**.

🔸 **Example:**

OrderService sends OrderPlaced event to Kafka
→ InventoryService consumes the event and updates stock.

✅ **Pros:**

- Loose coupling
- High performance & scalability

- Better fault tolerance (messages can be retried)

❌ **Cons:**

- Complex to implement and monitor
- Harder to trace flow across services
- Delayed response (not real-time)

🔹 **3. Head-to-Head Comparison Table**

Feature	Synchronous	Asynchronous
Communication style	Request–Response	Message–Driven
Protocol	REST, gRPC	Kafka, RabbitMQ
Real-time	Yes	No
Dependency	Tight coupling	Loose coupling
Failure propagation	Caller fails if callee is down	Caller doesn't fail immediately
Complexity	Simple	More complex
Use Case Example	Login, Payment	Email, Notification, Audit Logs
Monitoring	Easier (via logs & response)	Harder (needs tracing, monitoring)
Retry logic	Manual retry on failure	Built-in via queues
Scalability	Harder	Easier

🔹 **4. Real-World Use Cases**

Use Case	Recommended Type
Placing an order	Sync (REST)
Sending confirmation email	Async (Kafka)
Payment processing	Sync (gRPC or REST)
Inventory update	Async (Kafka)

Use Case	Recommended Type
Chat or messaging app	Sync (gRPC or WebSocket)
Logging user activity	Async (Kafka)

🔹 5. When to Choose What?

Scenario	Best Communication Type
Real-time need (user waits)	Synchronous
High volume, background processing	Asynchronous
Integration with external systems	Async (to avoid tight failures)
Business workflow with dependencies	Often a mix of both

🔹 6. Hybrid Approach

Most real-world systems use **both**:

- Use **synchronous** for direct, immediate actions
- Use **asynchronous** for decoupled, background processing

Example:

- User places order → synchronous
- Inventory update & email → asynchronous

✅ Microservices – Topic 7: Service Registration and Discovery (Eureka)

In a microservices system, there can be **dozens of services**, and they often **scale dynamically**. This makes managing their IP addresses and ports **difficult**.

To solve this, we use **Service Discovery**, which helps services **find each other automatically**.

🔹 1. What is Service Discovery?

It's the process by which one microservice **automatically discovers** the address of another microservice without hardcoding.

Instead of:

java

CopyEdit

http://localhost:8081/user-service

You use:

java

CopyEdit

http://USER-SERVICE/users

Behind the scenes, it resolves to the actual address via a service registry.

🔹 2. Why is Service Discovery Needed?

- IPs and ports of services **change dynamically** (especially in cloud and Docker)
- We want to **avoid hardcoding endpoints**
- Allows **load balancing**, **failover**, and **dynamic scaling**

🔹 3. Components

✅ a) Service Registry

Central registry where services **register themselves**.
E.g., **Netflix Eureka**, **Consul**, **Zookeeper**, **Etcd**

✅ b) Service Provider

A microservice that **registers itself** with the registry.

Example: UserService registers with Eureka

✅ **c) Service Consumer**

A microservice that **queries the registry** to get the location of other services.

Example: OrderService queries Eureka to find UserService

🔹 **4. Eureka Overview (Netflix OSS)**

👉 **Eureka Server:**

- Acts as the **service registry**
- Deployed as a Spring Boot app

👉 **Eureka Client:**

- Each microservice registers with Eureka using **@EnableEurekaClient**

🔹 **5. How It Works (Step-by-Step)**

1. You run **Eureka Server** (service registry).
2. All microservices **register** with it at startup.
3. Services **query** the registry to get addresses of other services.
4. Eureka **periodically checks** health of services.

🔹 **6. How to Set Up Eureka (Spring Boot)**

✅ **Step 1: Add Dependency in pom.xml**

xml

CopyEdit

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

✅ **Step 2: Create Eureka Server App**

java

CopyEdit

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {
  public static void main(String[] args) {
    SpringApplication.run(EurekaServerApplication.class, args);
  }
}
```

✅ **Step 3: Configure application.properties**

properties

CopyEdit

```
spring.application.name=eureka-server
server.port=8761

eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

✅ **Step 4: Add Eureka Client to Other Services**

xml

CopyEdit

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

In each microservice:

java

CopyEdit

```
@SpringBootApplication
@EnableEurekaClient
public class ProductServiceApp {
  public static void main(String[] args) {
    SpringApplication.run(ProductServiceApp.class, args);
  }
}
```

application.properties:

properties

CopyEdit

```
spring.application.name=product-service
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
```

🔹 7. Real-Life Analogy

Think of Eureka Server as **a receptionist**.
If you're looking for a service (like a department in an office), you **ask the receptionist**, and they give you the correct room number.

🔹 8. Benefits of Eureka

- Dynamic service discovery
- Load balancing support
- Health checks
- Easy integration with Spring Boot

🔹 9. Drawbacks

- Single point of failure (unless clustered)
- Network overhead due to heartbeats
- Tight coupling with Netflix ecosystem (though widely used)

✅ Microservices – Topic 8: Client-Side Load Balancing with Ribbon / Spring Cloud LoadBalancer

When multiple instances of a microservice are running, **load balancing** ensures the incoming requests are distributed evenly among them.

There are two types:

1. **Client-Side Load Balancing** (done by caller)
2. **Server-Side Load Balancing** (done by API gateway/load balancer)

In this topic, we focus on **Client-Side Load Balancing** using:

- **Netflix Ribbon** (legacy)
- **Spring Cloud LoadBalancer** (modern alternative)

🔹 1. What is Client-Side Load Balancing?

In client-side load balancing:

- The **client is responsible** for choosing the instance of the target service.
- It gets the list of instances from **Eureka** and picks one.

Example:
OrderService wants to call PaymentService. There are 3 instances. The client picks one and makes the call.

🔹 2. Netflix Ribbon (Legacy)

⚠️ **Ribbon is now deprecated** but still appears in many existing projects.

✅ **Features:**

- Works with Eureka
- Uses **round-robin** by default
- Can be customized

🔸 **How to Use:**

Ribbon is included automatically if you use spring-cloud-starter-netflix-eureka-client.

In application.properties:

properties

CopyEdit

payment-service.ribbon.NFLoadBalancerRuleClassName=com.netflix.loadbalancer.RandomRule

Default is RoundRobinRule, other options include RandomRule, WeightedResponseTimeRule, etc.

🔹 3. Spring Cloud LoadBalancer (Modern Approach)

Spring replaced Ribbon with its own **lightweight load balancer**:

- Works seamlessly with Eureka
- Used with WebClient or RestTemplate

✅ **Add Dependency (Spring Boot 2.4+):**

xml

CopyEdit

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-loadbalancer</artifactId>
</dependency>
```

✅ **Example using RestTemplate:**

java

CopyEdit

```
@Bean
@LoadBalanced
public RestTemplate restTemplate() {
   return new RestTemplate();
}
```

Then use:

java

CopyEdit

restTemplate.getForObject("http://PAYMENT-SERVICE/pay", String.class);

💡 @LoadBalanced tells Spring to apply load balancing and resolve service names using Eureka.

🔹 4. Load Balancing Strategies

Strategy	Description
Round Robin	Calls services in order
Random	Picks a random instance
Weighted	Gives preference to faster services
Retry	Retry failed calls automatically

🔹 5. Real-Life Example

If you have:

- 3 instances of payment-service
- order-service sends 9 requests

Then with Round Robin:

- Instance 1 gets: 3 requests
- Instance 2 gets: 3 requests
- Instance 3 gets: 3 requests

🔹 6. Benefits

- Load is distributed across instances
- Better performance and fault tolerance
- Avoids overloading a single service

🔹 7. Drawbacks

- Logic handled at client-side (harder to monitor)

- No global request queue
- Ribbon no longer maintained

🔹 **8. When to Use Client-Side Load Balancing?**

- With Eureka or Consul service discovery
- When you don’t want to depend on external load balancers
- For fine-grained control in microservices communication

✅ Microservices – Topic 9: API Gateway (Spring Cloud Gateway / Zuul)

In a microservices architecture, clients should not directly call each service individually. Instead, **API Gateway** acts as a single entry point to the system.

🔹 1. What is an API Gateway?

An **API Gateway** is a server that sits in front of microservices and routes client requests to the appropriate service.

👉 Think of it like:

🛬 Airport Check-in → You go to a counter → They route your luggage and direct you
Same with Gateway:
👷 Client → Gateway → Routes request to service

🔹 2. Why Do We Use an API Gateway?

- Single entry point for all services
- Hides internal structure from the client
- Adds **cross-cutting concerns** like:
    - Routing
    - Security (JWT/Auth)
    - Rate limiting
    - Load balancing
    - Logging
    - CORS

🔹 3. Popular API Gateways in Java

Gateway	Maintained By	Status
Spring Cloud Gateway	Spring Team	✅ Actively used
Netflix Zuul	Netflix	❌ Deprecated

We'll focus on **Spring Cloud Gateway**.

🔹 **4. Spring Cloud Gateway**

✅ **Features:**

- Built on **Spring WebFlux** (Reactive)
- Works with **Eureka**
- Supports:
  - Routing
  - Filters (Pre/Post)
  - Path rewriting
  - Circuit Breakers
  - Load Balancing

🔹 **5. How Does It Work?**

1. Client sends a request to Gateway (e.g., /product-service/products)
2. Gateway checks the route configuration
3. It forwards the request to the correct service
4. Optionally applies filters (auth, logging, etc.)

🔹 **6. Spring Cloud Gateway Setup (Step-by-Step)**

✅ **Step 1: Add Dependencies**

xml

CopyEdit

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
<dependency>
```

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

✅ **Step 2: Application Properties**

properties

CopyEdit

spring.application.name=api-gateway

server.port=8080

eureka.client.service-url.defaultZone=http://localhost:8761/eureka

spring.cloud.gateway.routes[0].id=product-service

spring.cloud.gateway.routes[0].uri=lb://PRODUCT-SERVICE

spring.cloud.gateway.routes[0].predicates[0]=Path=/product/**

spring.cloud.gateway.routes[1].id=order-service

spring.cloud.gateway.routes[1].uri=lb://ORDER-SERVICE

spring.cloud.gateway.routes[1].predicates[0]=Path=/order/**

✅ **Step 3: Main Class**

java

CopyEdit

@SpringBootApplication

@EnableEurekaClient

public class ApiGatewayApplication {

  public static void main(String[] args) {

    SpringApplication.run(ApiGatewayApplication.class, args);

```
  }
}
```

🔹 7. Gateway Filters

Filters let you add pre-processing or post-processing logic.

Type	Use Case
Pre-filter	Auth, Logging, Header validation
Post-filter	Response logging, Custom headers

Example: Add JWT token validation before routing to a service.

🔹 8. Real-Life Example

Imagine:

- You have 5 microservices.
- Clients (Web or Mobile) don't need to know all URLs.
- They just call:
  /order/1 → Gateway forwards to OrderService

🔹 9. Benefits of API Gateway

- Centralized entry point
- Security and monitoring in one place
- Easier versioning
- Decouples clients from services

🔹 10. Drawbacks

- Single point of failure (unless highly available)
- Can become a bottleneck under load
- Requires proper scaling

✅ Microservices – Topic 10: Circuit Breaker and Resilience using Resilience4j

When one microservice is down or slow, it can impact the entire system. To protect the system from failure, we use **Circuit Breaker** patterns.

🔹 1. What is a Circuit Breaker?

A **Circuit Breaker** is a design pattern that prevents an application from trying to perform an operation that is likely to fail.

⚡ Real-life Analogy:

When there's an overload of electricity ⚡, the **circuit breaks** to avoid damage. Similarly, in software, when a service is continuously failing, the circuit opens to avoid cascading failures.

🔹 2. Why Do We Use It?

- To **stop calling a failed service**
- To avoid **wasting resources**
- To **quickly recover** when the service is healthy again
- Improve **system resilience**

🔹 3. Tools for Circuit Breaker

Library	Maintained By	Notes
Hystrix (Old)	Netflix	❌ Deprecated
Resilience4j	Vavr.io	✅ Lightweight and popular
Sentinel	Alibaba	Mostly used in China

✅ We'll use **Resilience4j** (recommended by Spring).

🔹 4. States of a Circuit Breaker

State	Description
Closed	Normal operation. Calls pass through. Failure count monitored.
Open	Failure threshold exceeded. Calls are not allowed for a time.
Half-Open	A few test calls allowed. If successful, it moves to closed. If fails, open.

🔹 5. Setup Resilience4j in Spring Boot

✅ Step 1: Add Dependencies

xml

CopyEdit

```
<dependency>
 <groupId>io.github.resilience4j</groupId>
 <artifactId>resilience4j-spring-boot2</artifactId>
</dependency>
```

Or use Spring Cloud:

xml

CopyEdit

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
</dependency>
```

✅ Step 2: Use with @CircuitBreaker

java

CopyEdit

```
@RestController
public class OrderController {
```

```
    @Autowired
    private PaymentService paymentService;

    @GetMapping("/order/{id}")
    @CircuitBreaker(name = "paymentBreaker", fallbackMethod = "fallbackPayment")
    public String placeOrder(@PathVariable int id) {
        return paymentService.callPaymentService(id);
    }

    public String fallbackPayment(int id, Throwable t) {
        return "Payment service is down. Please try later.";
    }
}
```

🔹 **6. Config in application.yml (Optional)**

```
yaml
CopyEdit
resilience4j:
  circuitbreaker:
    instances:
      paymentBreaker:
        slidingWindowSize: 5
        failureRateThreshold: 50
        waitDurationInOpenState: 5s
```

📌 **Explanation:**

- If 3 out of 5 calls fail (60%), the circuit opens.
- It stays open for 5 seconds before trying again.

🔹 7. Other Features of Resilience4j

Feature	Description
Retry	Automatically retry failed calls
RateLimiter	Limit number of requests per second
Bulkhead	Isolate service calls (like thread pool)
TimeLimiter	Timeout slow responses

You can also combine them!

🔹 8. Real-Life Example

If your payment-service is down and the order-service keeps calling it:

- Without a Circuit Breaker: system hangs, crashes.
- With a Circuit Breaker: it quickly fails over to fallback response.

🔹 9. Benefits

- Avoid unnecessary calls
- Prevents system crash
- Improves user experience
- Increases availability

🔹 10. Drawbacks

- Adds complexity to service logic
- Requires tuning (thresholds, timers)
- Can be overused or misconfigured

✅ Microservices – Topic 11: Centralized Configuration using Spring Cloud Config Server

In microservices, each service usually has its own application.properties or application.yml. Managing configs across multiple services can become a nightmare — especially in different environments (dev, test, prod).

That's where **Spring Cloud Config Server** comes in.

🔹 1. What is Spring Cloud Config Server?

It is a **central place to manage all configuration files** of all microservices in one place — typically from a **Git repository**.

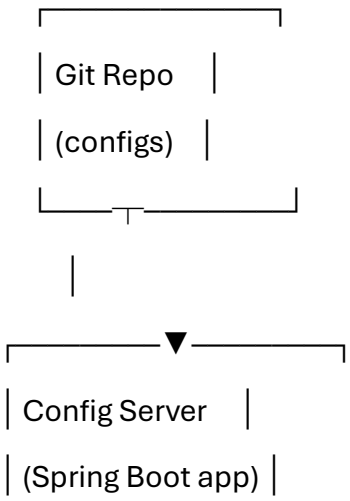
🔹 2. Why Do We Use It?

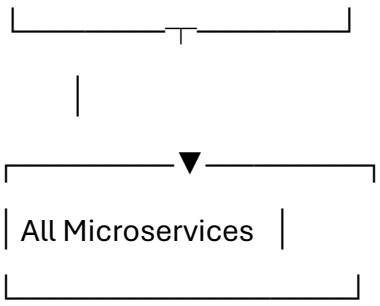
Without Config Server	With Config Server
Repeated properties	Centralized properties
Manual changes in each	One change updates all
Restart required on change	Supports auto-refresh

🔹 3. How It Works

scss

CopyEdit





🔹 **4. Project Setup – Step-by-Step**

✅ **Step 1: Create a Git Repo**

Create a Git repository like:

arduino

CopyEdit

https://github.com/yourname/microservice-configs

And add files:

scss

CopyEdit

order-service.yml

product-service.yml

application.yml (for common configs)

✅ **Step 2: Create Spring Cloud Config Server**

🔸 **Add Dependencies**

xml

CopyEdit

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

🔸 **Main Class**

java

CopyEdit

```
@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {
  public static void main(String[] args) {
    SpringApplication.run(ConfigServerApplication.class, args);
  }
}
```

🔸 **application.yml of Config Server**

yaml

CopyEdit

```
server:
 port: 8888

spring:
 cloud:
  config:
   server:
    git:
     uri: https://github.com/yourname/microservice-configs
     default-label: main
```

✅ **Step 3: Setup in Client Services**

Add dependency in all microservices:

xml

CopyEdit

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

🔸 **Bootstrap config (add in each service)**

yaml

CopyEdit

```
spring:
 application:
  name: order-service
 cloud:
  config:
   uri: http://localhost:8888
```

It will fetch config from:
http://localhost:8888/order-service/default

🔹 **5. Refresh Without Restart**

To update config **without restarting** the microservice:

1. Add spring-boot-starter-actuator
2. Enable refresh:

yaml

CopyEdit

```
management:
 endpoints:
  web:
   exposure:
```

include: refresh

3. Call this endpoint via Postman:

bash

CopyEdit

POST http://localhost:{port}/actuator/refresh

🔹 **6. Real-Life Example**

- You want to change DB credentials for all services in dev environment.
- Instead of modifying 10 apps, just push a change in Git.
- All services pull the new config dynamically.

🔹 **7. Benefits**

- Centralized config management
- Supports Git, Vault, JDBC
- Environment-specific configs
- Works with Docker, Kubernetes

🔹 **8. Drawbacks**

- Config server downtime affects all services
- Git repo must be secured
- Requires additional service to maintain

✅ Microservices – Topic 12: Service Communication: RESTTemplate, WebClient & FeignClient

In a microservices architecture, services often need to **talk to each other**. There are multiple ways to enable communication:

🔹 1. Why Service-to-Service Communication?

Suppose you have:

- Order-Service that needs details from Product-Service

The **order service must call** the product service API to get product details.

🔹 2. Three Common Ways to Call Another Microservice:

Method	Type	Suitable For
RESTTemplate	Synchronous	Legacy, simple use cases
WebClient	Reactive	Async, modern apps
FeignClient	Declarative	Clean & powerful

🔹 3. RESTTemplate (Legacy but still used)

✅ How It Works:

java

CopyEdit

```
@RestController
public class OrderController {

  @Autowired
  private RestTemplate restTemplate;

  @GetMapping("/order")
  public String getOrder() {
```

```
    String response = restTemplate.getForObject("http://localhost:8081/product", String.class);
    return "Order placed with product: " + response;
  }
}
```

✅ **Bean Config:**

java

CopyEdit

```
@Bean
public RestTemplate restTemplate() {
  return new RestTemplate();
}
```

⚠️ **Drawback:**

- Blocking (not async)
- Deprecated in favor of WebClient

🔹 **4. WebClient (Modern, Reactive)**

✅ **Setup:**

Add this dependency if not present:

xml

CopyEdit

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

✅ **Usage:**

java

CopyEdit

```
@Bean
public WebClient webClient() {
   return WebClient.builder().build();
}
```

java

CopyEdit

```
@Autowired
private WebClient webClient;


public Mono<String> getProduct() {
   return webClient.get()
         .uri("http://localhost:8081/product")
         .retrieve()
         .bodyToMono(String.class);
}
```

✅ **Features:**

- Asynchronous & Non-blocking
- Suitable for high-concurrency

🔹 **5. FeignClient (Declarative, Recommended)**

✅ **Add Dependency:**

xml

CopyEdit

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

✅ **Enable Feign:**

In the main class:

java

CopyEdit

@EnableFeignClients

✅ **Interface:**

java

CopyEdit

```
@FeignClient(name = "product-service", url = "http://localhost:8081")
public interface ProductClient {
   @GetMapping("/product")
   String getProduct();
}
```

✅ **Usage:**

java

CopyEdit

```
@Autowired
private ProductClient productClient;

@GetMapping("/order")
public String getOrder() {
   return "Order placed with product: " + productClient.getProduct();
}
```

🔹 **6. Using Feign with Service Discovery (Eureka)**

java

CopyEdit

```
@FeignClient(name = "product-service") // No need for URL
```

Now FeignClient uses the Eureka registry to locate the service.

🔹 7. Comparison Table

Feature	RESTTemplate	WebClient	FeignClient
Sync/Async	Sync	Async (Reactive)	Sync (Declarative)
Ease of Use	Manual URLs	More code	Very simple interface
Modern	❌ Deprecated	✅ Recommended	✅ Highly used
Best For	Legacy code	Reactive systems	Microservice communication

---

🔹 8. Real-Life Example

- A **payment-service** calls the **order-service** via FeignClient.
- If you're building a dashboard app and need fast async calls → use WebClient.
- If you have old codebase → you might still use RESTTemplate.

🔹 9. Benefits

- Enables microservice collaboration
- Allows clean separation of concerns
- FeignClient improves readability and reduces boilerplate

🔹 10. Drawbacks

- Adds network latency
- Needs fallback handling (Circuit Breaker!)
- WebClient can be complex for beginners

✅ Microservices – Topic 13: Eureka Server – Service Registry & Discovery

When microservices grow in number, hardcoding URLs becomes a maintenance nightmare. That's where **Eureka** helps — it acts as a **phone book** for services.

🔹 1. What is Eureka?

- A **service registry** where all microservices **register themselves**.
- Other services **discover** them by name instead of IP/port.

Eureka is a part of **Netflix OSS** and integrated with Spring Cloud.

🔹 2. Why Do We Use It?

Problem Without Eureka	Solution With Eureka
Hardcoded URLs between services	Use logical names (e.g., order-service)
Manual IP change on deployment	Eureka tracks it dynamically
No load balancing	Eureka + Ribbon handles it

🔹 3. Eureka Setup

✅ Step 1: Create Eureka Server

🔸 Dependencies:

xml

CopyEdit

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

🔸 Main Class:

java

CopyEdit

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApp {
   public static void main(String[] args) {
      SpringApplication.run(EurekaServerApp.class, args);
   }
}
```

🔸 **application.yml:**

```
yaml
CopyEdit
server:
 port: 8761

eureka:
 client:
   register-with-eureka: false
   fetch-registry: false
```

✅ **Step 2: Register Microservices (Eureka Clients)**

🔸 **Dependencies:**

```
xml
CopyEdit
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

🔸 **Main Class:**

```
java
```

CopyEdit

```
@EnableEurekaClient
@SpringBootApplication
public class ProductServiceApplication {
  public static void main(String[] args) {
    SpringApplication.run(ProductServiceApplication.class, args);
  }
}
```

🔸 **application.yml:**

yaml

CopyEdit

```
spring:
 application:
  name: product-service

eureka:
 client:
  service-url:
   defaultZone: http://localhost:8761/eureka
```

🔹 **4. Accessing Eureka Dashboard**

- Run Eureka server
- Visit: http://localhost:8761
- You'll see registered services listed

🔹 **5. Discovering Services**

Example:

If order-service needs to call product-service, you don't hardcode URL.

Instead of:

java

CopyEdit

```
@GetMapping("/call")
public String call() {
   return restTemplate.getForObject("http://localhost:8081/product", String.class);
}
```

Use:

java

CopyEdit

```
@GetMapping("/call")
public String call() {
   return restTemplate.getForObject("http://product-service/product", String.class);
}
```

Here, product-service is discovered from Eureka.

✅ **Required Bean:**

java

CopyEdit

```
@Bean
@LoadBalanced
public RestTemplate restTemplate() {
   return new RestTemplate();
}
```

@LoadBalanced enables service name lookup from Eureka.

🔹 **6. Real-Life Example**

- **Netflix** has hundreds of services. Eureka helps them register and talk to each other without IP headaches.
- In your apps: payment-service, order-service, and user-service all register with Eureka and discover each other easily.

🔹 7. Benefits

- Centralized service registry
- No hardcoded IPs
- Works well with Feign, Ribbon, etc.
- Enables load balancing

🔹 8. Drawbacks

- Single point of failure (can use Eureka clusters)
- Delay in updating service info
- Only works well with Spring ecosystem

✅ **Microservices – Topic 14: Client-Side Load Balancing with Ribbon and Spring Cloud LoadBalancer**

🔹 **1. What is Load Balancing?**

When multiple instances of a service run (e.g., 3 instances of product-service), a **load balancer** distributes incoming traffic among them.

✅ Ensures performance, reliability, and scalability.

🔹 **2. Types of Load Balancing**

Type	Works On	Tools
Client-side	Caller decides route	Ribbon, LoadBalancer
Server-side	Gateway/load balancer	Nginx, HAProxy

In microservices, we mostly use **client-side load balancing** with **Ribbon** or **Spring Cloud LoadBalancer**.

🔹 **3. Ribbon (Legacy but still used)**

Ribbon is a Netflix client-side load balancer that works with **RestTemplate** or **FeignClient**.

✅ **How It Works:**

When you call:

java

CopyEdit

restTemplate.getForObject("http://product-service/products", String.class);

Ribbon automatically fetches all instances of product-service from Eureka and picks one using a **load balancing algorithm** (default: Round Robin).

🔹 **4. Spring Cloud LoadBalancer (Modern Replacement)**

✅ **Add Dependency:**

Spring Boot 2.4+ recommends this over Ribbon.

xml

CopyEdit

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-loadbalancer</artifactId>
</dependency>
```

🔹 5. Configuration for RestTemplate

java

CopyEdit

```
@Bean
@LoadBalanced
public RestTemplate restTemplate() {
   return new RestTemplate();
}
```

✅ Now any call using RestTemplate to http://<service-name> will go through load balancing.

🔹 6. FeignClient + Load Balancer

FeignClient **automatically** uses Spring LoadBalancer (or Ribbon if you have it in classpath):

java

CopyEdit

```
@FeignClient(name = "product-service")
public interface ProductClient {
   @GetMapping("/products")
   String getProduct();
```

}

✅ This will also automatically distribute calls across instances of product-service.

🔹 7. Load Balancing Algorithms

Default is **Round Robin**, but you can also configure:

- **Random**
- **Weighted Response**
- **Custom Rules**

With Ribbon:

yaml

CopyEdit

product-service:

ribbon:

NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule

With Spring LoadBalancer (custom logic needed)

🔹 8. Real-Life Example

You deploy 5 instances of payment-service. Without load balancing, only the first one gets all traffic. With Ribbon or LoadBalancer, all 5 get equal or balanced traffic.

🔹 9. Benefits

- Automatic traffic distribution
- No manual service instance selection
- Works with Eureka and Feign

🔹 10. Drawbacks

- Ribbon is deprecated
- Doesn’t work without Eureka unless configured manually

- Can overload instances if not configured properly

✅ Microservices – Topic 15: API Gateway using Spring Cloud Gateway (or Zuul)

🔹 1. What is an API Gateway?

An **API Gateway** is the **entry point** for all clients in a microservices system.

Instead of calling each microservice directly, the client sends requests to the gateway, which routes them to the right service.

🔹 2. Why Use an API Gateway?

Without Gateway	With Gateway
Client calls each microservice	Client only talks to one URL
Manages routing manually	Central routing, filtering, throttling
No centralized auth/logging	Centralized security, monitoring

🔹 3. Popular Gateways

Tool	Notes
Zuul (Netflix)	Legacy, still used in older projects
Spring Cloud Gateway	Modern, reactive, replaces Zuul
Kong, Nginx, etc.	External tools, infrastructure-level

We'll focus on **Spring Cloud Gateway** (recommended by Spring).

🔹 4. Spring Cloud Gateway Setup

✅ **Dependency:**

xml

CopyEdit

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

✅ **Main Class:**

java

CopyEdit

```
@SpringBootApplication
public class ApiGatewayApp {
   public static void main(String[] args) {
      SpringApplication.run(ApiGatewayApp.class, args);
   }
}
```

🔹 **5. Configuration (application.yml)**

yaml

CopyEdit

```
server:
 port: 8080

spring:
 application:
   name: api-gateway

 cloud:
   gateway:
```

```
routes:
 - id: product-service
   uri: lb://product-service
   predicates:
    - Path=/products/**
 - id: order-service
   uri: lb://order-service
   predicates:
    - Path=/orders/**
```

lb:// tells it to use **LoadBalancer** (i.e., Eureka service name)

Now:

- http://localhost:8080/products → routed to product-service
- http://localhost:8080/orders → routed to order-service

🔹 6. Common Features of Gateway

Feature	Use Case Example
Routing	Send to correct microservice
Load Balancing	Works with Eureka to balance requests
Authentication	JWT validation at gateway level
Rate Limiting	Prevent abuse by limiting API calls
Logging	Log and monitor all requests centrally
Filtering	Add/remove headers, request transformation

🔹 7. Filters in Gateway

✅ Pre and Post Filters

java

CopyEdit

```
@Component
public class CustomFilter implements GlobalFilter {

  @Override
  public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
    // Before request
    System.out.println("Request: " + exchange.getRequest().getURI());

    return chain.filter(exchange).then(Mono.fromRunnable(() -> {
      // After response
      System.out.println("Response code: " +
exchange.getResponse().getStatusCode());
    }));
  }
}
```

🔹 8. Real-Life Example

In real-world apps:

- Clients (Web/Mobile) call **Gateway**
- Gateway routes to services like:
  - /auth/** → Auth Service
  - /cart/** → Cart Service
  - /checkout/** → Payment Service

🔹 9. Benefits

- Single entry point
- Centralized security/logging
- Load balancing + Eureka integration

- Easy API versioning and throttling

🔹 **10. Drawbacks**

- Single point of failure (unless clustered)
- Slight latency added
- Needs proper scaling if traffic is high

✅ Microservices – Topic 16: Spring Cloud Config – Centralized Configuration Management

---

🔹 1. What Is Spring Cloud Config?

Spring Cloud Config is a centralized **configuration management system** for microservices.

It lets you manage **all config files from a single location**, typically a **Git repository**.

🔹 2. Why Use It?

Without Config Server	With Config Server
Each service has its own application.yml	One central config repo for all services
Hard to update config in production	Just update Git and refresh
Risk of version mismatch	All services read config from same version

---

🔹 3. How It Works

- You create a Git repo with config files like:

css

CopyEdit

product-service.yml

order-service.yml

application.yml

- Services fetch config from **Spring Cloud Config Server** at startup (or on refresh).

🔹 4. Components Involved

1. **Spring Cloud Config Server**
2. **Spring Cloud Config Client**
3. **Git Repository** (or local folder, Vault, etc.)

🔹 **5. Create a Config Server**

✅ **Step 1: Add Dependency**

xml

CopyEdit

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

✅ **Step 2: Main Class**

java

CopyEdit

```
@EnableConfigServer
@SpringBootApplication
public class ConfigServerApp {
   public static void main(String[] args) {
      SpringApplication.run(ConfigServerApp.class, args);
   }
}
```

✅ **Step 3: application.yml**

yaml

CopyEdit

```
server:
 port: 8888

spring:
 cloud:
   config:
    server:
```

```
    git:
      uri: https://github.com/your-user/your-config-repo
```

🔹 6. Create a Config Client (e.g., product-service)

✅ Add Dependency

xml

CopyEdit

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

✅ Bootstrap or Application YAML

yaml

CopyEdit

```
spring:
 application:
  name: product-service
 cloud:
  config:
   uri: http://localhost:8888
```

Now product-service will fetch config from:

bash

CopyEdit

```
http://localhost:8888/product-service.yml
```

🔹 7. Example Git Repo Structure

arduino

CopyEdit

```
/config-repo
  ├── product-service.yml
  ├── order-service.yml
  └── application.yml
```

🔹 8. Refresh Configuration at Runtime

You can make services **refresh config** at runtime using:

- @RefreshScope annotation
- Hitting: POST /actuator/refresh

🔹 9. Real-Life Example

In large teams:

- Developers push only to config repo for updates
- Ops teams monitor config from one place
- Restart not required due to dynamic refresh

🔹 10. Benefits

- Single source of truth
- Supports Git, Vault, JDBC, etc.
- Reduces duplication across services
- Easy version control (Git)

🔹 11. Drawbacks

- Config Server itself needs to be highly available
- Initial setup can be complex
- If Git is down, config can't load (unless cached)

✅ Microservices – Topic 17: Circuit Breaker using Resilience4j (or Hystrix)

🔹 1. What Is a Circuit Breaker?

A **Circuit Breaker** is a design pattern used to detect failures and **prevent calls to failing services**. It helps make the system **resilient and fault-tolerant**.

It "breaks" the call flow temporarily when a downstream service is unavailable or slow.

🔹 2. Real-Life Analogy

Imagine you're calling a friend, and their phone is always busy. After trying a few times, you stop calling for a while (circuit opens), then try again later (circuit half-open), and if it works, you resume (closed again).

🔹 3. Why Use Circuit Breaker?

Problem	Solution
Downstream service is slow	Stop sending requests temporarily
Repeated failure calls	Save resources using fallback logic
Service timeout	Avoid cascading failure

🔹 4. Popular Circuit Breaker Libraries

Library	Status	Notes
Hystrix	Deprecated	Netflix's old library
Resilience4j	Actively used	Modern, lightweight, Spring Boot compatible

We'll use **Resilience4j**, which works well with Spring Boot.

🔹 5. Add Resilience4j Dependency

xml

CopyEdit

```
<dependency>
 <groupId>io.github.resilience4j</groupId>
 <artifactId>resilience4j-spring-boot2</artifactId>
</dependency>
```

Also add Actuator for monitoring:

xml

CopyEdit

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

🔹 6. Enable Circuit Breaker in Service

java

CopyEdit

```
@RestController
public class ProductController {

  @Autowired
  private ProductService productService;

  @GetMapping("/products")
  @CircuitBreaker(name = "productService", fallbackMethod = "fallbackGetProducts")
  public String getProducts() {
    return productService.fetchProducts();
  }

  public String fallbackGetProducts(Exception ex) {
```

```
    return "Product Service is currently unavailable. Please try later.";
  }
}
```

🔹 7. Configuration (application.yml)

yaml

CopyEdit

```
resilience4j:
 circuitbreaker:
  instances:
   productService:
    registerHealthIndicator: true
    slidingWindowSize: 5
    failureRateThreshold: 50
    waitDurationInOpenState: 5s
    permittedNumberOfCallsInHalfOpenState: 3
```

🔹 8. Circuit Breaker States

State	Meaning
Closed	Normal operation, all calls pass
Open	Service is failing; calls are blocked for some time
Half-Open	Few calls are allowed to check if service is back

🔹 9. Real-Life Example

Imagine the payment-service is down. Instead of retrying every second and overloading it, the circuit breaker stops trying for 5 seconds, then allows a few calls to check if it's up again.

🔹 10. Benefits

- Prevents cascading failures
- Gives fallback responses
- Improves system stability
- Helps in graceful degradation

🔹 11. Drawbacks

- Adds some complexity
- Misconfiguration can block healthy services
- Circuit breaker logic may need tuning

✅ Microservices – Topic 18: Service Discovery with Eureka

🔹 1. What Is Service Discovery?

In microservices, **services need to find each other** to communicate. Hardcoding IPs or URLs is not scalable.

Service Discovery solves this by allowing services to register themselves and discover other services dynamically.

🔹 2. What Is Eureka?

Eureka is a **Service Registry** created by Netflix and integrated with Spring Cloud.

- Services **register themselves** with Eureka.
- Other services **discover them** using their name.

🔹 3. Key Components

Component	Role
Eureka Server	Registry where all services register
Eureka Client	A service that registers itself with Eureka and discovers others

---

🔹 4. Eureka Server Setup

✅ Step 1: Add Dependency

xml

CopyEdit

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

✅ Step 2: Main Class

java

CopyEdit

```
@EnableEurekaServer
@SpringBootApplication
public class EurekaServerApp {
   public static void main(String[] args) {
      SpringApplication.run(EurekaServerApp.class, args);
   }
}
```

✅ **Step 3: application.yml**

yaml

CopyEdit

```
server:
 port: 8761

spring:
 application:
   name: eureka-server

eureka:
 client:
   register-with-eureka: false
   fetch-registry: false
```

Run the server → Visit http://localhost:8761 to see Eureka Dashboard.

🔹 **5. Eureka Client Setup (e.g., product-service)**

✅ **Step 1: Add Dependency**

xml

CopyEdit

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

✅ **Step 2: Main Class**

java

```
@SpringBootApplication
@EnableDiscoveryClient
public class ProductServiceApp {
   public static void main(String[] args) {
      SpringApplication.run(ProductServiceApp.class, args);
   }
}
```

✅ **Step 3: application.yml**

yaml

```
server:
 port: 8081

spring:
 application:
   name: product-service

eureka:
 client:
   service-url:
     defaultZone: http://localhost:8761/eureka/
```

Now, the product-service will register itself with Eureka.

🔹 6. Service Discovery Example

A gateway-service can discover and route to product-service dynamically:

yaml

CopyEdit

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

lb:// uses **LoadBalancer** + Eureka to discover service instance.

🔹 7. Real-Life Example

- Netflix uses Eureka to register 100s of microservices.
- Your app can register auth-service, payment-service, etc., and call each other via service names.

🔹 8. Benefits

- Dynamic service discovery
- No hardcoding IPs
- Load balancing supported
- Integrated with Spring Cloud

🔹 9. Drawbacks

- Single point of failure (need replication)
- Slight overhead to maintain registry
- Slower in very large-scale systems (use Consul/Kubernetes)

✅ Microservices – Topic 19: Client-Side Load Balancing with Spring Cloud LoadBalancer

---

🔹 1. What Is Load Balancing?

When you have **multiple instances** of a microservice, **load balancing** ensures that incoming requests are **distributed evenly**.

There are 2 types:

- **Client-side**: Load balancing happens in the calling service
- **Server-side**: A central load balancer like NGINX or HAProxy handles it

---

🔹 2. What Is Spring Cloud LoadBalancer?

It is the **default client-side load balancer** used in Spring Cloud (replacing Ribbon).

It allows your service to **choose one of the registered service instances** from Eureka and call it.

🔹 3. When Does It Come Into Play?

When you use:

yaml

CopyEdit

uri: lb://service-name

Or in code:

java

CopyEdit

RestTemplate.getForObject("http://user-service/users", String.class);

Spring automatically resolves user-service to an instance using the load balancer.

🔹 4. Setup in Spring Boot

✅ Step 1: Add Dependencies

xml

CopyEdit

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>

<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-loadbalancer</artifactId>
</dependency>
```

🔹 **5. Enable Load-Balanced RestTemplate**

java

CopyEdit

```
@Configuration
public class AppConfig {

   @Bean
   @LoadBalanced
   public RestTemplate restTemplate() {
      return new RestTemplate();
   }
}
```

Now you can use service name instead of host URL:

java

CopyEdit

```
String result = restTemplate.getForObject("http://order-service/orders", String.class);
```

🔹 6. How It Works

1. RestTemplate sends a request to order-service.
2. Spring LoadBalancer queries Eureka for all available instances of order-service.
3. It picks one instance based on a **load balancing algorithm** (Round Robin by default).
4. Sends the request.

🔹 7. Real-Life Example

You have 3 instances of payment-service running on different ports:

- http://localhost:9001
- http://localhost:9002
- http://localhost:9003

When a request comes in via http://payment-service, Spring chooses one of them **automatically** using load balancing.

🔹 8. Supported Load Balancing Strategies

- Round Robin (default)
- Random
- Custom Strategy (by implementing ReactorServiceInstanceLoadBalancer)

🔹 9. Benefits

- No need to hardcode host and port
- Automatic failover if one instance is down
- Works with Eureka + RestTemplate / WebClient / Feign

🔹 10. Drawbacks

- Client handles logic, adding some complexity

- Not as powerful as server-side load balancers (NGINX, HAProxy)

✅ Microservices – Topic 20: API Gateway using Spring Cloud Gateway

---

🔹 1. What Is an API Gateway?

An **API Gateway** is a **single entry point** for all client requests in a microservices architecture.

It routes requests to appropriate microservices, applies filters (e.g., security, logging), and handles cross-cutting concerns like rate limiting, retries, etc.

🔹 2. Why Use an API Gateway?

Without Gateway	With API Gateway
Client must know all services	One entry point only
Complex client logic	Central routing handled by gateway
No central control	Common filters/security applied

🔹 3. What Is Spring Cloud Gateway?

Spring Cloud Gateway is a powerful, modern API Gateway built on top of **Spring WebFlux**.

It provides:

- Dynamic routing
- Filters (Pre/Post)
- Integration with Eureka and Load Balancer
- Security (JWT, OAuth2)
- Rate limiting

🔹 4. Add Dependency

xml

CopyEdit

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>

<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

🔹 **5. application.yml (Routing Configuration)**

yaml

CopyEdit

```
server:
 port: 8080

spring:
 application:
  name: api-gateway

 cloud:
  gateway:
   routes:
    - id: student-service
      uri: lb://student-service
      predicates:
```

```
      - Path=/student/**
    - id: course-service
      uri: lb://course-service
      predicates:
        - Path=/course/**

eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
```

Any request like /student/getAll will be routed to the student-service.

🔹 6. Filters in Gateway

You can apply filters like:

- **Logging**
- **Authentication**
- **Rate Limiting**
- **Header modification**

Example:

yaml

CopyEdit

```
filters:
  - AddRequestHeader=MyHeader, MyValue
```

🔹 7. Real-Life Example

Netflix uses Zuul (older version) as an API Gateway.
Modern systems use Spring Cloud Gateway to expose:

- /login → Auth Service

- /products → Product Service
- /orders → Order Service

🔹 **8. Custom Filter (Example)**

java

CopyEdit

```
@Component
public class LoggingFilter implements GlobalFilter {

  @Override
  public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
    System.out.println("Request Path: " + exchange.getRequest().getPath());
    return chain.filter(exchange);
  }
}
```

🔹 **9. Benefits**

- Centralized routing and control
- Works well with Eureka and LoadBalancer
- Built-in support for reactive, fast performance
- Easily customizable

🔹 **10. Drawbacks**

- Slight learning curve with WebFlux
- Performance bottleneck if not scaled properly
- Not suitable for heavy computation logic

✅ Microservices – Topic 21: Centralized Logging with ELK Stack

---

🔹 1. What Is Centralized Logging?

In a **microservices system**, logs are spread across many services and servers. **Centralized logging** brings all logs into **one place** for easier monitoring, debugging, and analysis.

🔹 2. What Is ELK Stack?

ELK is a combination of:

Component	Role
Elasticsearch	Search and analytics engine
Logstash	Log collector and processor
Kibana	Visualization and dashboard tool

🔹 3. Why Use ELK in Microservices?

- View all logs from all services in one place
- Filter logs by service, level, time, etc.
- Detect errors and performance issues quickly
- Create dashboards and alerts

🔹 4. How ELK Works Together

1. Services write logs (e.g., to file or console)
2. **Logstash** collects logs
3. Logstash sends them to **Elasticsearch**
4. **Kibana** displays logs in real time

🔹 5. Log Format Example

Spring Boot logs:

log

CopyEdit

2024-06-01 10:00:00 INFO OrderService: Order created successfully

These logs are parsed by Logstash and sent to Elasticsearch with fields like:

- timestamp
- log level
- service name
- message

🔹 **6. Setting Up ELK Stack (High-level steps)**

✅ **Step 1: Install ELK (using Docker or manually)**

bash

CopyEdit

docker-compose up -d

✅ **Step 2: Configure Logstash**

Example logstash.conf:

conf

CopyEdit

```
input {
 file {
   path => "/var/log/microservices/*.log"
   start_position => "beginning"
 }
}
filter {
 json {
   source => "message"
```

```
 }
}
output {
 elasticsearch {
  hosts => ["http://localhost:9200"]
  index => "microservice-logs"
 }
}
```

✅ **Step 3: Access Kibana**

Visit http://localhost:5601

Create index pattern microservice-logs* to view logs.

🔹 **7. Spring Boot Integration (using Filebeat)**

Install **Filebeat** to forward logs to Logstash:

- Filebeat watches .log files
- Sends them to Logstash in real time

🔹 **8. Real-Life Example**

Imagine you have 10 services:

- auth-service
- product-service
- order-service
- etc.

All their logs are stored in one searchable dashboard, making it easy to find:

- Errors
- Slow API calls
- System crashes

🔹 **9. Benefits**

- Single dashboard for all logs
- Easy debugging and filtering
- Helps in incident response
- Scales with microservices

🔹 **10. Drawbacks**

- Complex initial setup
- Consumes memory and CPU
- Requires storage and scaling as logs grow

✅ Microservices – Topic 22: Centralized Configuration with Spring Cloud Config

---

🔹 1. What Is Centralized Configuration?

In a microservices architecture, you may have 10+ services. Each has:

- application.yml or .properties files
- Environment-specific configurations (dev, prod, etc.)

Managing these files individually becomes hard and error-prone.

👉 **Spring Cloud Config** lets you manage all service configs **from a central Git repo**.

🔹 2. What Is Spring Cloud Config?

Spring Cloud Config provides:

- **Centralized config server**
- Externalized config stored in **Git, SVN, or filesystem**
- Support for dynamic refresh of config using **Spring Cloud Bus**

🔹 3. Why Use Spring Cloud Config?

Problem Without Config Server	With Spring Cloud Config
Duplicated config files	Single Git repo for all configs
Manual update and redeployment	Auto refresh with Spring Cloud Bus
No history/versioning	Git history for audit & rollback

🔹 4. How It Works

🏛️ **Components:**

- **Config Server**: Reads configs from Git and serves them
- **Config Client**: A microservice that fetches config from the server at runtime

---

🔹 5. Setting Up Spring Cloud Config

✅ **Step 1: Create a Config Server**

Add dependency:

xml

CopyEdit

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

Enable it:

java

CopyEdit

```
@EnableConfigServer
@SpringBootApplication
public class ConfigServerApplication {}
```

application.yml

yaml

CopyEdit

```
server:
 port: 8888

spring:
 cloud:
  config:
   server:
    git:
     uri: https://github.com/your-user/your-config-repo
```

✅ **Step 2: Create a Git Repo with Config Files**

Structure:

arduino

CopyEdit

your-config-repo/

├── student-service.yml

├── order-service.yml

└── application.yml

Each file contains that service's config.

✅ **Step 3: Set Up a Client**

Add dependency:

xml

CopyEdit

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

application.yml:

yaml

CopyEdit

```
spring:
 application:
   name: student-service


 cloud:
   config:
    uri: http://localhost:8888
```

Now student-service will fetch config from:

bash

CopyEdit

http://localhost:8888/student-service/default

🔹 6. Real-Life Example

Suppose you want to change DB URL for payment-service.
Rather than editing and redeploying the service, you update payment-service.yml in the Git repo.
All services fetch the updated config from Git **without redeployment**.

🔹 7. Dynamic Refresh with Spring Cloud Bus

Use:

java

CopyEdit

@RefreshScope

on beans to auto-refresh values when /actuator/refresh is called or when using Kafka/RabbitMQ via Spring Cloud Bus.

🔹 8. Benefits

- Manage all configs from one place
- Environment-based configuration (dev, test, prod)
- Version control via Git
- Dynamic refresh without restart

🔹 9. Drawbacks

- Requires Git setup
- Added complexity with refresh/bus setup
- Centralized point of failure (needs HA)

✅ Microservices – Topic 23: Circuit Breaker with Resilience4j

---

🔹 1. What Is a Circuit Breaker?

In microservices, if **one service fails**, it can slow down or crash other services that depend on it.

A **Circuit Breaker**:

- Prevents calls to a **failing service**
- Allows the system to recover faster
- Avoids **cascading failures**

🔹 2. Real-Life Analogy

Think of a **circuit breaker in your house**:

- If there's an overload, it **trips** (breaks) to protect the system.
- Once things are safe, you can **reset** it.

🔹 3. Circuit Breaker States

State	Meaning
Closed	All calls go to the service
Open	All calls are blocked for some time
Half-Open	Some calls are tested; if successful, move back to Closed

---

🔹 4. What Is Resilience4j?

Resilience4j is a lightweight fault tolerance library for Java.

Features:

- Circuit Breaker ✅
- Retry
- Rate Limiter

- Bulkhead
- Time Limiter
- Cache

It works well with Spring Boot and is a better alternative to Netflix Hystrix (which is now in maintenance mode).

🔹 5. Add Dependencies

xml

CopyEdit

```
<dependency>
 <groupId>io.github.resilience4j</groupId>
 <artifactId>resilience4j-spring-boot2</artifactId>
</dependency>

<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

🔹 6. Basic Usage Example

java

CopyEdit

```
@RestController
public class OrderController {

   @GetMapping("/order")
   @CircuitBreaker(name = "orderService", fallbackMethod = "fallbackOrder")
   public String placeOrder() {
```

```
    // simulate external service call
    throw new RuntimeException("Service down");
  }

  public String fallbackOrder(Exception e) {
    return "Fallback: Order service is currently unavailable!";
  }
}
```

🔹 7. Configure Circuit Breaker

application.yml

yaml

CopyEdit

```
resilience4j:
 circuitbreaker:
  instances:
   orderService:
    registerHealthIndicator: true
    slidingWindowSize: 5
    minimumNumberOfCalls: 3
    failureRateThreshold: 50
    waitDurationInOpenState: 10s
```

🔹 8. Real-Life Use Case

Amazon or Flipkart:

- If the **payment gateway** service is slow or down, instead of retrying and hanging, the system shows a fallback message:
  “Payment service is temporarily unavailable. Try again later.”

🔹 **9. Benefits**

- Prevents cascading failures
- Quick failure response improves UX
- Auto-recovery using Half-Open state
- Easy to integrate with Spring Boot

🔹 **10. Drawbacks**

- Requires careful configuration (thresholds, durations)
- Can delay actual recovery if not tuned well
- Adds slight latency due to internal state checks

✅ Microservices – Topic 24: Retry Mechanism with Resilience4j

---

🔹 1. What Is a Retry Mechanism?

A **retry mechanism** automatically tries a failed operation again after a short delay.

It's useful when a **temporary issue** occurs (e.g., network glitch, short DB downtime, API timeout).

🔹 2. Why Use Retry in Microservices?

In distributed systems, failures are **inevitable**:

- Timeout
- Rate limit
- Connection reset

Instead of immediately giving up, retrying gives the system a **chance to recover**.

🔹 3. What Is Resilience4j Retry?

Resilience4j Retry:

- Retries a method call upon failure
- Supports configuring:
    - Number of retry attempts
    - Wait time between attempts
    - Exception types to retry

🔹 4. Add Dependency

Already added with Circuit Breaker:

xml

CopyEdit

<dependency>

```
 <groupId>io.github.resilience4j</groupId>
 <artifactId>resilience4j-spring-boot2</artifactId>
</dependency>
```

🔹 5. Usage Example

java

CopyEdit

```
@RestController
public class PaymentController {

  @GetMapping("/pay")
  @Retry(name = "paymentRetry", fallbackMethod = "paymentFallback")
  public String processPayment() {
    System.out.println("Trying payment...");
    throw new RuntimeException("Temporary payment failure");
  }

  public String paymentFallback(Exception e) {
    return "Fallback: Payment service is down!";
  }
}
```

🔹 6. Configure Retry

application.yml

yaml

CopyEdit

```
resilience4j:
 retry:
```

```
instances:
  paymentRetry:
    maxAttempts: 3
    waitDuration: 2s
    retryExceptions:
      - java.lang.RuntimeException
```

👉 This will:

- Try up to 3 times
- Wait 2 seconds between each try
- Retry only on RuntimeException

🔹 **7. Real-Life Example**

Let's say a service is calling a **third-party payment API**.

Scenario:

- API fails on first try
- But succeeds on second try

✅ Retry helps complete the operation successfully **without user noticing a failure**.

🔹 **8. Retry with Exponential Backoff (Advanced)**

Add increasing wait times:

yaml

CopyEdit

```
waitDuration: 1s
intervalFunction:
  type: exponential_backoff
  exponentialBackoffMultiplier: 2
```

Now retries happen after 1s, 2s, 4s...

🔹 9. Benefits

- Helps avoid failure on temporary issues
- Increases reliability without manual retries
- Works well with Circuit Breaker

🔹 10. Drawbacks

- Can overload a failing system (use with Circuit Breaker!)
- Increases latency for the user
- May retry when it's not helpful (must configure carefully)

✅ Microservices – Topic 25: Rate Limiting with Resilience4j

🔹 1. What Is Rate Limiting?

Rate Limiting controls how many requests a user or service can make in a given period of time.

Example:
Allow only **10 requests per second** per user to protect the system from overload.

🔹 2. Why Do We Need Rate Limiting?

🔥 Without Rate Limiting:

- One client can **overload** your service
- Can lead to **denial of service (DoS)**
- Consumes bandwidth, memory, and CPU unnecessarily

✅ With Rate Limiting:

- Fair use policy
- Prevent abuse
- Maintain system stability

🔹 3. What Is Resilience4j Rate Limiter?

Resilience4j RateLimiter:

- Limits the number of permitted calls within a time interval
- Throws an exception or fallback if the limit is exceeded

🔹 4. Add Dependency

Already included with:

xml

CopyEdit

<dependency>

```
 <groupId>io.github.resilience4j</groupId>
 <artifactId>resilience4j-spring-boot2</artifactId>
</dependency>
```

🔹 5. Usage Example

java

CopyEdit

```
@RestController
public class UserController {

  @GetMapping("/users")
  @RateLimiter(name = "userLimiter", fallbackMethod = "userFallback")
  public String getUsers() {
    return "User data sent!";
  }

  public String userFallback(Throwable t) {
    return "Too many requests! Please try again later.";
  }
}
```

🔹 6. Configuration

application.yml

yaml

CopyEdit

```
resilience4j:
 ratelimiter:
  instances:
```

```
userLimiter:
  limitForPeriod: 5
  limitRefreshPeriod: 10s
  timeoutDuration: 0
```

This means:

- Allow 5 requests every 10 seconds
- If limit is hit, no wait (timeoutDuration = 0), fallback is triggered

---

🔹 7. Real-Life Example

🔐 **Login API:**

Limit login attempts to **3 per minute** per user.

If someone tries to brute-force passwords, rate limiter will **block** them temporarily to prevent abuse.

🔹 8. Benefits

- Protects services from being overloaded
- Enforces usage policies
- Increases fairness and security

🔹 9. Drawbacks

- Needs careful tuning
- If shared limit across users, one user may block others
- May cause user frustration if limit is too strict

✅ Microservices Interview Questions – All in One Place

🔹 1. Microservices Basics

1. What are Microservices?
2. Monolithic vs Microservices – What's the difference?
3. What are the main benefits of using Microservices?
4. What are the key challenges in Microservice architecture?
5. Explain the principle of "Single Responsibility" in Microservices.

🔹 2. Architecture & Design Patterns

6. What is Domain-Driven Design (DDD) and how does it apply to Microservices?
7. What is Bounded Context in Microservices?
8. How do Microservices communicate with each other?
9. What are synchronous and asynchronous communication patterns?
10. What is the role of REST APIs in Microservices?

🔹 3. Spring Boot + Microservices

11. How do you create Microservices using Spring Boot?
12. What annotations are required to expose a REST endpoint?
13. How do you register a Microservice with Eureka Server?
14. How does Spring Boot simplify Microservices development?
15. What is the role of application.yml in a Microservice?

🔹 4. Service Discovery (Eureka)

16. What is Service Discovery?
17. What is Eureka Server and Eureka Client?
18. How do you register a Microservice with Eureka?

19. How does a client discover services from Eureka?
20. What are the advantages of using Eureka?

- **5. Load Balancing (Ribbon)**

21. What is Client-Side Load Balancing?
22. What is Ribbon and how does it work?
23. How does Ribbon choose which service instance to use?
24. Is Ribbon still supported in Spring Boot 3.x?
25. What can replace Ribbon in newer projects?

- **6. API Gateway (Spring Cloud Gateway)**

26. What is an API Gateway?
27. Why do we need API Gateway in Microservices?
28. What is Spring Cloud Gateway?
29. How do you route traffic using Spring Cloud Gateway?
30. How can you apply filters in Spring Cloud Gateway?

- **7. Configuration Management (Spring Cloud Config)**

31. What is Spring Cloud Config?
32. How does it help in managing configuration in Microservices?
33. How can you refresh configuration without restarting the service?
34. What is the role of @RefreshScope?
35. What happens if Config Server is down?

- **8. Fault Tolerance (Resilience4j)**

36. What is fault tolerance in Microservices?
37. What is a Circuit Breaker?
38. What are the three states of a Circuit Breaker?

39. How does Resilience4j Circuit Breaker work?
40. What is a Retry mechanism in Resilience4j?
41. How can you configure number of retries and delay?
42. What is Rate Limiting and why is it important?
43. What is Bulkhead and Time Limiter in Resilience4j?

- **9. Security**

44. How do you secure Microservices?
45. What is OAuth2 and how does it help in Microservices security?
46. What is JWT (JSON Web Token)?
47. How does authentication and authorization work in Microservices?
48. What is a centralized authentication service?

- **10. Database Strategies**

49. Should each Microservice have its own database?
50. What is Database Per Service pattern?
51. How do you manage data consistency across services?
52. What is SAGA pattern?

- **11. Testing & Deployment**

53. How do you test Microservices?
54. What is contract testing?
55. How do you deploy Microservices?
56. What is containerization? Why is Docker used?
57. What is the role of Kubernetes in Microservices?

- **12. Advanced**

58. What is Event-Driven Architecture?

59. What is a Message Broker? Examples?
60. What is Centralized Logging? Tools?
61. How do you monitor Microservices?
62. What is API Versioning?