17. 마이크로서비스 아키텍처 (Spring Cloud)

서비스 디스커버리: Eureka

🧩 1. 서비스 디스커버리란?

> 분산 시스템(MSA)에서 **서비스 인스턴스의 위치(IP, PORT)를 자동으로 등록하고 탐색**할 수 있도록 도와주는 기능

✅ IP가 동적으로 바뀌는 서비스 환경에서
✅ 클라이언트가 직접 주소를 알지 않고도
✅ **서비스 이름 기반으로 요청 가능**

🔍 2. Eureka란?

> Netflix가 만든 오픈소스 **서비스 디스커버리 서버**

- Spring Cloud Netflix에서 공식 지원
- **Eureka Server**: 레지스트리 서버
- **Eureka Client**: 서비스 제공자 + 탐색자

⚙️ 3. 구성 흐름

```
[Service A] ← (서비스명으로 탐색) ← Eureka ← (등록) ← [Service B]
```

1. 서비스 B가 Eureka에 자신을 등록
2. 서비스 A가 Eureka에서 “service-b”의 위치 조회
3. Eureka는 서비스 목록을 클라이언트에게 응답

📘 4. Eureka 기본 구성 요소

구성	역할
Eureka Server	레지스트리 역할: 등록된 서비스 정보를 관리
Eureka Client	등록자 + 탐색자 역할
Service Registry	인스턴스 목록, 상태 정보, heartbeat 기반 갱신
Self-preservation	대량 등록 해제 방지 보호 모드

🛠️ 5. Eureka Server 설정 (Spring Boot)

✅ 1) 의존성 추가

```
dependencies {
    implementation 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-server'
}
```

✅ 2) 어노테이션 설정

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication { ... }
```

✅ 3) application.yml

```
server:
  port: 8761

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

➡️ 실행 후 http://localhost:8761 에서 Eureka Dashboard 확인 가능

🧩 6. Eureka Client 설정

✅ 1) 의존성 추가

```
implementation 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client'
```

✅ 2) application.yml

```
spring:
  application:
    name: user-service

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

➡️ 이 설정으로 `user-service`라는 이름으로 Eureka에 자동 등록됨

🔁 7. 클라이언트에서 다른 서비스 호출하기

✅ 방법 1: RestTemplate + `@LoadBalanced`

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

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

➡️ `order-service`라는 **서비스 이름으로** 호출 가능
(Spring Cloud LoadBalancer가 Eureka에서 주소를 조회함)

✅ 방법 2: Feign Client 사용

```
implementation 'org.springframework.cloud:spring-cloud-starter-openfeign'
```

```
@FeignClient(name = "order-service")
public interface OrderClient {
    @GetMapping("/orders")
    List<OrderDto> getOrders();
}
```

➡️ Feign이 Eureka에서 `order-service`의 위치를 찾아 자동 호출

🧠 8. 실무에서의 Eureka 설계 팁

항목	전략
포트 충돌	서비스마다 랜덤 포트(`server.port=0`) 추천
health-check	actuator 연동 or `management.health` 직접 구현
다중 인스턴스	Eureka는 클러스터링 지원 (`peer-eureka-nodes`)
통신 보안	TLS 적용 or gateway 앞단 처리
탐색 주기	기본 30초 → `leaseRenewalIntervalInSeconds` 조정 가능

🚨 9. Eureka 단점 & 보완

이슈	설명	해결
Self-preservation 모드	장애 시 서비스 제거 안 됨	설정으로 꺼야 함
자동 제거 지연	실패한 인스턴스가 UI에 남음	TTL 설정 단축
Scale 문제	수천 개 이상 시 Consul/Zookeeper 고려	분산 캐시 연계

✅ 마무리 요약표

구성	설명
Eureka Server	서비스 레지스트리 서버 역할
Eureka Client	서비스 등록 + 탐색 가능
탐색 방식	서비스명 기반 → IP 동적 매핑
연동 방식	`RestTemplate + @LoadBalanced`, Feign, WebClient
실무 활용	API Gateway ↔ 서비스 탐색, 동적 스케일링 지원

구성 서버: Spring Cloud Config

🧩 1. Spring Cloud Config란?

> 분산된 마이크로서비스 환경에서 **설정 파일(application.yml)을 중앙 집중화**하여 관리할 수 있도록 해주는 Spring Cloud 구성 요소.

✅ 환경 설정(application.yml 등)을 **Git, SVN, 파일 시스템 등에 저장**
✅ 각 마이크로서비스는 해당 설정을 **Config Server로부터 실시간으로 fetch**

🔍 2. 왜 필요한가?

기존 방식 문제	Config Server의 해결
서비스마다 설정 중복/불일치	설정 중앙 집중화
배포 없이 설정 수정 불가	설정 변경 → 실시간 반영
운영환경별 설정 관리 어려움	profile 기반 환경 분리
시크릿/민감 정보 관리 어려움	암호화/복호화 지원

🧱 3. 구성 구조

```
[Git Repo] ← (설정 파일)
   ↑
[Config Server]
   ↓
[Service A], [Service B], ...
```

- Git 저장소에 설정 저장 (`application.yml`, `user-service.yml`, ...)
- Config Server가 Git에서 설정을 읽어 서비스에 전달
- 각 서비스는 자신의 설정을 `/config` 서버에서 fetch

⚙️ 4. Config Server 구성

✅ 1) 의존성 추가

```
implementation 'org.springframework.cloud:spring-cloud-config-server'
```

✅ 2) 설정 및 어노테이션

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

✅ 3) application.yml

```
server:
  port: 8888

spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/my-org/my-config-repo
          default-label: main
```

> GitHub, GitLab, Bitbucket, 사설 Git 모두 지원

✅ 4) Git 저장소 파일 구조 예시

```
/config-repo/
├── application.yml             # 공통 설정
├── user-service.yml            # user-service 전용 설정
├── order-service-dev.yml       # order-service + dev 프로파일
```

🔄 5. Config Client 설정

✅ 1) 의존성 추가

```
implementation 'org.springframework.cloud:spring-cloud-starter-config'
```

✅ 2) bootstrap.yml 또는 application.yml

```
spring:
  application:
    name: user-service
  cloud:
    config:
      uri: http://localhost:8888
      profile: dev
      label: main
```

> `application.name`이 Config Server에서 검색되는 파일명에 대응됨: `user-service-dev.yml`

📲 6. 실행 후 동작

서비스 시작 시:

```
GET http://localhost:8888/user-service/dev
→ user-service-dev.yml + application.yml 병합 후 반환
```

Spring Boot는 이를 내부 설정으로 적용

💥 7. 설정 변경 → 실시간 반영

✅ Actuator + Bus 연동 필요

- Config Server: RabbitMQ 등 메시지 브로커 연동
- Client에 `spring-boot-starter-actuator`, `spring-cloud-starter-bus-amqp` 추가

```
curl -X POST http://localhost:8080/actuator/refresh
```

> 최신 Spring Cloud Bus 사용 시 `/actuator/busrefresh`로 다수 인스턴스 동시 반영 가능

🔐 8. 민감 정보(시크릿) 암호화

✅ 서버에 대칭키 설정

```
encrypt:
  key: my-secret-key
```

✅ Git 설정 파일에 암호화된 값 입력

```
db-password: '{cipher}d43ed3...'
```

➡️ 클라이언트에서 자동 복호화됨

🧠 9. 실무 설계 팁

항목	전략
Git 브랜치	`main`, `dev`, `prod` 분리 + `label`로 매핑
구조	`application.yml` → 공통, 서비스별 파일은 덮어쓰기
동적 변경	`@RefreshScope`로 Bean 재로딩 가능
보안	Config Server에 인증 필터 추가 (`Spring Security`)
캐시	`spring.cloud.config.cache` 설정 고려

✅ 마무리 요약표

항목	설명
Config Server	설정 파일을 중앙에서 제공하는 서버
설정 저장소	Git, SVN, 파일 시스템 등 가능
Client 구성	`spring.application.name` + `profile` + `label`
실시간 반영	`/actuator/refresh` or Spring Cloud Bus
암호화 지원	`{cipher}` + 대칭키 기반 복호화
실무 적용	마이크로서비스 공통 설정, 외부환경 변수 관리, 시크릿 분리

API Gateway: Spring Cloud Gateway

🧩 1. Spring Cloud Gateway란?

> Spring Cloud에서 제공하는 **마이크로서비스용 API Gateway 솔루션**

✅ 모든 외부 요청을 **중앙 관문(Gateway)**을 통해 처리
✅ 라우팅, 필터링, 인증, 속도 제한, 로깅 등
✅ Netflix Zuul의 대체로, Spring 5의 WebFlux 기반 비동기 처리 지원

🎯 2. 왜 API Gateway가 필요한가?

문제	Gateway 역할
서비스 URL 노출	단일 진입점 제공 (`api.domain.com`)
인증 중복	공통 인증 필터 처리
CORS 문제	Gateway에서 일괄 설정 가능
요청 로깅/추적	일관된 로깅 및 트래픽 제어
동적 라우팅	Eureka 연동으로 자동 라우팅

🧱 3. 기본 구성 흐름

```
[Client]
  ↓
[Spring Cloud Gateway]
  ↓          ↓           ↓
Auth Filter  Logging     Rate Limit
  ↓
[Route] → [User Service, Order Service, ...]
```

⚙️ 4. 기본 설정

✅ 1) 의존성 추가

```
implementation 'org.springframework.cloud:spring-cloud-starter-gateway'
```

> 필요 시: `spring-cloud-starter-netflix-eureka-client`도 함께

✅ 2) application.yml 라우팅 설정

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: user-service
          uri: http://localhost:8081
          predicates:
            - Path=/user/**
          filters:
            - AddRequestHeader=X-Request-Gateway, Gateway
```

> `/user/**` 요청 → `http://localhost:8081`으로 전달

🗺️ 5. 핵심 요소 정리

요소	설명
Route	요청 URL, 조건, 목적지 URI 설정
Predicate	경로/메서드/헤더 등 조건 지정 (`Path`, `Method`, `Header`)
Filter	전/후처리 작업 수행 (`AddHeader`, `StripPrefix`, 커스텀 필터 등)
URI	LB (`lb://user-service`), HTTP 등 목적지 지정 방식
Global Filter	전체 요청에 적용되는 필터 (@Component)

🧭 6. Eureka 연동 + 서비스명 기반 라우팅

```yaml
spring:
  cloud:
    gateway:
      discovery:
        locator:
          enabled: true
          lower-case-service-id: true
```

> `http://gateway/api/user-service/**` → `lb://user-service`로 전달됨
> (서비스명 기반 동적 라우팅)

🔐 7. 인증/보안 적용 예

✅ 1) 커스텀 인증 필터 작성

```
@Component
public class AuthFilter implements GlobalFilter {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        String token = exchange.getRequest().getHeaders().getFirst("Authorization");

        if (!isValid(token)) {
            exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
            return exchange.getResponse().setComplete();
        }

        return chain.filter(exchange);
    }
}
```

✅ 2) CORS, HTTPS, JWT, OAuth2도 Gateway에서 일괄 설정 가능

```
spring:
  cloud:
    gateway:
      globalcors:
        corsConfigurations:
          '[/**]':
            allowedOrigins: "*"
            allowedMethods: "*"
```

📊 8. 고급 필터 예시

```
filters:
  - name: RequestRateLimiter
    args:
      redis-rate-limiter.replenishRate: 10
      redis-rate-limiter.burstCapacity: 20
```

➡️ IP당 초당 10건, 최대 버스트 20건 → 속도 제한

```
filters:
  - name: Retry
    args:
      retries: 3
      statuses: BAD_GATEWAY
```

➡️ 502 에러 발생 시 최대 3회 재시도

💡 9. 실무 설계 팁

항목	전략
서비스 이름	Eureka 등록명과 동일하게 설정
인증	JWT 필터를 Gateway에 공통 적용
로깅	GlobalFilter에서 요청/응답 로깅 처리
Rate Limit	Redis 기반 RequestRateLimiter 사용
장애 격리	Fallback route 설정 가능 (`Hystrix` 제거됨 → Resilience4j 추천)

✅ 마무리 요약표

항목	설명
Gateway 역할	MSA 외부 진입점, 라우팅 + 인증/필터
Predicate	요청 조건 정의 (`Path`, `Header`, `Method`)
Filter	전후처리 작업 (`Auth`, `RateLimit`, `Logging`)
Eureka 연동	`lb://user-service` 기반 라우팅 자동 구성
실무 활용	공통 인증, 로깅, 보안 처리, 동적 URI 구성

로드 밸런싱: Spring Cloud LoadBalancer

🧩 1. Spring Cloud LoadBalancer란?

> **Spring Cloud가 제공하는 클라이언트 사이드 로드 밸런서**
> 마이크로서비스 환경에서 **서비스명으로 요청을 보내면**,
> **자동으로 여러 인스턴스 중 하나를 선택해서 요청**을 전달해줌.

✅ 클라이언트 요청 → 로컬 로드 밸런서 → 인스턴스 선택 → 서비스 호출

🔁 2. 왜 필요한가?

문제	해결
특정 인스턴스에 트래픽 집중	트래픽 분산
IP/PORT 관리 복잡	서비스명 기반 호출
스케일 아웃된 인스턴스 자동 탐색	Eureka, Consul 등과 연동

🔁 3. 기존 Ribbon과의 차이

항목	Ribbon (Deprecated)	Spring Cloud LoadBalancer
라이브러리	Netflix Ribbon	Spring 자체 구현
유지보수	중단됨	✅ 활성
동기화	RestTemplate 등과 연동	✅ RestTemplate, ✅ WebClient, ✅ Feign 지원
커스터마이징	제한적	전략 구성 완전 가능

> Spring Boot 2.4 이후 기본 로드밸런서는 **Spring Cloud LoadBalancer**

⚙️ 4. 기본 동작 구조

```
[RestTemplate or WebClient or Feign]
        ↓
lb://user-service (서비스명)
        ↓
ServiceInstanceListSupplier
        ↓
LoadBalancerClient (기본은 RoundRobin)
        ↓
http://IP:PORT 로 요청
```

🔌 5. RestTemplate 연동 예시

✅ 1) 설정

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

✅ 2) 사용

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

➡️ `order-service`는 Eureka에 등록된 여러 인스턴스 중 하나가 선택됨

🌐 6. WebClient 연동 예시

```
@Bean
@LoadBalanced
public WebClient.Builder webClientBuilder() {
    return WebClient.builder();
}
```

```
String res = webClientBuilder.build()
    .get()
    .uri("http://payment-service/pay")
    .retrieve()
    .bodyToMono(String.class)
    .block();
```

🤝 7. Feign Client 연동

```
@FeignClient(name = "inventory-service")
public interface InventoryClient {
    @GetMapping("/inventory/check")
    boolean isAvailable();
}
```

➡️ 자동으로 Spring Cloud LoadBalancer를 통해 인스턴스 선택

🔁 8. 라운드로빈 전략 (기본값)

```
@Bean
public ReactorLoadBalancer<ServiceInstance> loadBalancer(Environment env,
    LoadBalancerClientFactory factory) {
    String name = env.getProperty(LoadBalancerClientFactory.PROPERTY_NAME);
    return new RoundRobinLoadBalancer(factory.getLazyProvider(name,
ServiceInstanceListSupplier.class), name);
}
```

➡️ 인스턴스들을 순차적으로 돌아가며 선택

⚙️ 9. 커스텀 로드밸런싱 전략 예시

```
public class MyCustomLoadBalancer implements ReactorServiceInstanceLoadBalancer {

    private final ObjectProvider<ServiceInstanceListSupplier>
serviceInstanceListSupplierProvider;
    private final String serviceId;

```

```
    public MyCustomLoadBalancer(ObjectProvider<ServiceInstanceListSupplier> provider,
String serviceId) {
        this.serviceInstanceListSupplierProvider = provider;
        this.serviceId = serviceId;
    }

    @Override
    public Mono<Response<ServiceInstance>> choose(Request request) {
        return serviceInstanceListSupplierProvider.getIfAvailable()
            .get()
            .next()
            .map(instances -> {
                // IP 기반 선택, 랜덤, Hash 기반 등 전략 가능
                return new DefaultResponse(instances.get(0));
            });
    }
}
```

🧠 10. 실무 설계 팁

항목	전략
기본 전략	`RoundRobin`으로 충분, 특정 요구시 `Weighted`, `IPHash`
로깅	로드밸런서 선택된 인스턴스 로깅 → 문제 추적 도움
장애 대응	HealthCheck 없이 Eureka 기반만 사용 시, 장애 인스턴스 선택 위험 있음
분산 정책	서비스 특성에 따라 다르게 구성 가능 (읽기 전용, 쓰기 전용 인스턴스 등)
확장성	`LoadBalancerClientFilter`, `ReactorLoadBalancer`로 고급 구성 가능

✅ 마무리 요약표

항목	설명
기본 전략	`RoundRobinLoadBalancer`
지원 대상	`RestTemplate`, `WebClient`, `FeignClient`
Eureka 연동	`lb://service-name` 주소 → IP:PORT 선택
커스텀 전략	`ReactorLoadBalancer` 구현으로 확장 가능
실무 중요 포인트	분산 처리 + 인스턴스 장애 대응 + 로깅 + 테스트

장애 대응: Resilience4j

🧩 1. Resilience4j란?

> 마이크로서비스 간 통신에서 발생할 수 있는 **지연, 실패, 장애 상황을 제어하고 보호**하기 위한 경량 라이브러리

📌 **Netflix Hystrix의 대체**로 개발됨
📌 **함수형(FP), 람다 기반 구조**, Spring Boot와 완벽 연동

🔍 2. 왜 필요한가?

문제 상황	해결책 (Resilience4j)
외부 서비스 응답 지연 → 전체 지연	⛔ Circuit Breaker
일시적 오류 → 전체 실패	🔁 Retry
트래픽 폭증 → 다운	⏱️ Rate Limiter
리소스 초과 → 타임아웃	⏳ Timeout
장애 발생 시 Fallback 없음	🔁 Fallback 처리

🧱 3. Resilience4j의 주요 모듈

모듈	설명
✅ CircuitBreaker	장애 감지 시 회로 차단 (Fail Fast)
✅ Retry	실패 요청 자동 재시도
✅ RateLimiter	초당 요청 제한 (Throttling)
✅ Bulkhead	동시 실행 수 제한 (격리)
✅ TimeLimiter	타임아웃 처리
✅ Cache	실패 시 이전 결과 재사용 (optional)

⚙️ 4. Spring Boot + Resilience4j 기본 설정

✅ 1) 의존성 추가

```
implementation 'io.github.resilience4j:resilience4j-spring-boot3'
implementation 'org.springframework.boot:spring-boot-starter-aop'
```

✅ 2) application.yml 설정

```
resilience4j:
  circuitbreaker:
    instances:
      myService:
        slidingWindowSize: 10
        failureRateThreshold: 50
        waitDurationInOpenState: 5s
  retry:
    instances:
      myService:
        maxAttempts: 3
        waitDuration: 1s
```

🛠️ 5. 사용 예시 (Spring 방식)

```
@Service
public class RemoteUserService {

    @CircuitBreaker(name = "myService", fallbackMethod = "fallback")
    @Retry(name = "myService")
    public String getUser() {
        return restTemplate.getForObject("http://user-service/users", String.class);
    }

    public String fallback(Throwable t) {
        return "⚠ 사용자 정보 조회 실패 (fallback)";
    }
}
```

🔁 6. Circuit Breaker 동작 원리

```
CLOSED → 오류율 ↑ → OPEN → 일정 시간 후 HALF-OPEN → 성공 시 CLOSED
```

상태	설명
CLOSED	정상 통신 상태
OPEN	호출 차단 상태 (fallback 수행)
HALF-OPEN	일부 허용 → 성공 판단 후 CLOSED 복귀 or 재오픈

> 실시간 장애를 감지하고 **장애 전파 차단**이 핵심

🔁 7. Retry 동작 원리

```
요청 실패 → 자동 재시도 (n회) → fallback
```

- 네트워크 단절, 일시적 오류에 유효
- 주의: **DB 트랜잭션에서 재시도 시 중복 처리 발생 가능성** → `@Transactional` 조합 신중

⏱️ 8. RateLimiter / TimeLimiter / Bulkhead

모듈	설정 예시
RateLimiter	초당 요청 수 제한 → `limitForPeriod: 10`, `limitRefreshPeriod: 1s`
TimeLimiter	`timeoutDuration: 2s` → 응답이 2초 넘으면 강제 종료
Bulkhead	`maxConcurrentCalls: 5` → 동시에 5개만 실행 가능 (ThreadPool 격리)

➡️ 설정은 모두 `resilience4j.<module>.instances.<name>` 하위에 구성

📊 9. 실시간 상태 모니터링

Spring Boot Actuator와 연동하면 상태 확인 가능:

```
/actuator/circuitbreakerevents
/actuator/metrics/resilience4j.circuitbreaker.state
```

> Prometheus + Grafana로 대시보드 시각화 가능

🧠 10. 실무 설계 팁

항목	전략
Retry	비즈니스 무결성 유지가 중요한 작업엔 신중하게
CircuitBreaker	외부 API, DB, 결제 시스템 등에 꼭 설정
RateLimiter	인증/검색 서비스에 적용 추천
Fallback	단순 메시지 or 캐시 응답 등 전략적으로 설계
모니터링	actuator, log, slack 알림으로 실시간 추적

✅ 마무리 요약

모듈	핵심 기능
`@CircuitBreaker`	오류 감지 시 호출 차단 + fallback
`@Retry`	자동 재시도
`@RateLimiter`	초당 호출 수 제한
`@TimeLimiter`	응답 시간 초과 제어
`@Bulkhead`	격리 실행 (쓰레드 or semaphore)
fallbackMethod	실패 시 대체 응답 방식 지정

트레이싱: Sleuth, Zipkin

🧩 1. 개념: 분산 트레이싱이란?

> 마이크로서비스 환경에서 **하나의 요청이 여러 서비스로 전달**될 때,
> 그 **전체 요청 흐름을 추적(trace)** 하고
> **시각화하여 병목, 장애 원인을 분석**할 수 있도록 하는 구조.

✅ 단일 로그로는 흐름 파악이 어려움
✅ Sleuth가 **Trace ID/Span ID 부여**, Zipkin이 **시각화**

🔍 2. Sleuth란?

> Spring에서 **로그에 Trace ID/Span ID를 자동으로 삽입**하여
> 요청 간 관계를 추적할 수 있게 해주는 분산 트레이싱 라이브러리

✅ 핵심 용어

용어	설명
Trace ID	하나의 전체 요청 흐름을 식별하는 ID
Span ID	하나의 세부 작업 단위를 식별하는 ID
Parent Span	하위 작업의 부모 작업 ID
Tags/Logs	메타 정보 (URI, status, time 등)

🧩 3. Zipkin이란?

> Sleuth가 생성한 트레이스 데이터를 수집하여
> **시각적으로 요청 흐름을 보여주는 트레이싱 서버**

✅ 웹 UI에서 **요청 단위 시각화, 지연 구간 분석, 호출 순서 파악**

⚙️ 4. Sleuth + Zipkin 아키텍처 구조

```
[Client]
  ↓
[Gateway]
  ↓           ↓
TraceID   SpanID 부여
  ↓
[Service A] → [Service B]
      ↘
     [Zipkin Server] ← 로그 전송
```

➡️ 모든 서비스에 Sleuth 적용 시
➡️ Trace ID가 **모든 로그에 포함**되고
➡️ Zipkin은 전체 요청 트리 흐름을 보여줌

📦 5. 의존성 구성 (Gradle)

```
implementation 'org.springframework.cloud:spring-cloud-starter-sleuth'
implementation 'org.springframework.cloud:spring-cloud-starter-zipkin'
```

📘 6. 기본 설정 예시 (application.yml)

```
spring:
  zipkin:
    base-url: http://localhost:9411
    enabled: true
  sleuth:
    sampler:
      probability: 1.0  # 100% 추적 (운영에서는 0.1~0.5 권장)
```

🌐 7. Zipkin 서버 실행 방법

✅ 방법 1: 도커 실행

```
docker run -d -p 9411:9411 openzipkin/zipkin
```

➡️ http://localhost:9411 에서 웹 UI 확인 가능

✅ 방법 2: Spring Boot 기반 서버 직접 실행

```
implementation 'io.zipkin.java:zipkin-server'
implementation 'io.zipkin.java:zipkin-autoconfigure-ui'
```

```
@SpringBootApplication
@EnableZipkinServer
public class ZipkinServerApplication { ... }
```

📝 8. Sleuth가 로그에 추가하는 항목

```
[INFO] [user-service, 1234567890abcdef, abc12345] 요청 시작
```

항목	설명
application name	서비스 이름
traceId	전체 요청 식별자
spanId	현재 작업의 ID
parentId	상위 작업 ID (있을 경우)

🛠️ 9. 실전 예제 흐름 (FeignClient 포함)

✅ Service A (user-service)

```
@FeignClient(name = "order-service")
public interface OrderClient {
    @GetMapping("/orders")
    List<Order> getOrders();
}
```

➡️ Sleuth가 자동으로 **Trace ID/Span ID를 Feign 요청 헤더에 삽입**

✅ Service B (order-service)

```
GET /orders
→ 헤더에 traceId, spanId, parentSpanId 포함
→ Zipkin에 trace 정보 전송됨
```

🧠 10. 실무 설계 팁

항목	전략
추적 범위	운영 환경에선 `sampler.probability = 0.1` 정도로 설정
연동 대상	Gateway, Service, DB, Kafka 등 모두 추적 가능
보안	민감한 데이터는 Span tag에 직접 넣지 않기

항목	전략
지연 분석	Trace 시간 기준으로 병목 지점 시각적으로 확인
로그 연계	ELK (ElasticSearch + Kibana)와 함께 연동 시 검색도 가능

✅ 마무리 요약표

항목	설명
Sleuth	Trace ID/Span ID 부여 → 로그 자동 삽입
Zipkin	Trace 데이터를 수집하고 시각화
연동 방법	의존성 + base-url 설정
Feign/WebClient	자동으로 traceId 전달
실무 활용	장애 추적, 성능 병목 분석, 요청 흐름 추적

이벤트 기반 연동: Spring Cloud Stream

🧩 1. Spring Cloud Stream이란?

> Kafka, RabbitMQ 등 메시징 시스템을 기반으로 하는
> **이벤트 기반 마이크로서비스 연동을 위한 추상화 프레임워크**

📌 메시지 브로커에 종속되지 않고,
📌 생산자(Publisher)와 소비자(Consumer)를 **애플리케이션 단위로 선언**
📌 코드를 변경하지 않고도 **브로커를 교체**할 수 있음 (바인더 개념)

🧱 2. 구성 구조

```
[Service A] -- emits Event --> [Broker] --> [Service B] -- handles Event
```

➡️ 서비스 A, B는 서로 직접 알지 않음 (Loosely Coupled)

🧰 3. 핵심 구성 요소

구성 요소	설명
Binder	Kafka, RabbitMQ 등과 연결을 담당하는 어댑터
Binding	애플리케이션 → 바인더 간 메시지 채널 연결
Producer	이벤트 발행 주체 (`@Output`)
Consumer	이벤트 수신 주체 (`@Input`, `@StreamListener`)

구성 요소	설명
MessageChannel	메시지 전달 경로 (`Flux<Message<T>>` 등)

📦 4. 지원하는 바인더(Binder)

바인더	설명
`kafka`	Apache Kafka
`rabbit`	RabbitMQ
`kinesis`	AWS Kinesis (추가 설정 필요)
`solace`, `azure-event-hubs`, ...	다양한 바인더 확장 가능

⚙️ 5. 의존성 (예: Kafka 바인더 사용 시)

```
implementation 'org.springframework.cloud:spring-cloud-starter-stream-kafka'
```

RabbitMQ 사용 시:

```
implementation 'org.springframework.cloud:spring-cloud-starter-stream-rabbit'
```

📘 6. 설정 예시 (Kafka 기준)

```
spring:
  cloud:
    stream:
      default-binder: kafka
      bindings:
        user-events-out:
          destination: user.events
          content-type: application/json
        user-events-in:
          destination: user.events
          group: user-service
```

항목	의미
`destination`	Kafka 토픽 이름 또는 RabbitMQ 큐
`group`	Kafka consumer group
`content-type`	메시지 변환 형식 (JSON, AVRO 등)

🔁 7. 이벤트 발행 (Producer)

```
@RequiredArgsConstructor
@EnableBinding(UserEventSource.class)
public class UserEventPublisher {

    private final UserEventSource source;

    public void publish(UserEvent event) {
        source.output().send(MessageBuilder.withPayload(event).build());
    }
}
```

```
public interface UserEventSource {
    @Output("user-events-out")
    MessageChannel output();
}
```

➡️ `user-events-out` → Kafka 토픽 `user.events`로 바인딩됨

📥 8. 이벤트 수신 (Consumer)

```
@EnableBinding(UserEventSink.class)
public class UserEventHandler {

    @StreamListener("user-events-in")
    public void handle(UserEvent event) {
        System.out.println("📥 사용자 이벤트 수신: " + event.getUsername());
    }
}
```

```
public interface UserEventSink {
    @Input("user-events-in")
    SubscribableChannel input();
}
```

⚡ 9. Spring Cloud Stream의 장점

기능	설명
브로커 추상화	코드 변경 없이 Kafka ↔ Rabbit 교체 가능
이벤트 중심	진짜 비동기, 느슨한 결합 구조
확장성	확장성 좋은 Kafka 기반으로 손쉽게 분산 처리
테스트 편의	Embedded Kafka, Test Binder 제공

🧠 10. 실무 설계 팁

항목	전략
이벤트 설계	`Event Sourcing`과는 다르게 **명확한 의미(Event DTO)**를 설계
메시지 신뢰성	Kafka의 경우 `acks`, Rabbit의 경우 `manual ack` 고려
재처리	DLQ 설정 필요 (Kafka: retry topic / Rabbit: DLX)
멱등성	메시지 중복 수신 대비 → `idempotent` 처리 필요
관측성	Zipkin + Sleuth 연동 가능 (TraceID 전파)

✅ 마무리 요약표

항목	설명
바인더(Binder)	Kafka, RabbitMQ 등 메시징 연동 어댑터
Producer	`@Output`, `MessageChannel`을 통해 이벤트 발행
Consumer	`@Input`, `@StreamListener`로 이벤트 수신
설정 중심	`bindings.*` 통해 토픽/큐/컨슈머 그룹 설정
추상화 수준	고수준 → 브로커 종속성 제거 + 통일된 구조 제공

인증과 보안: OAuth2 Resource Server

1. 개념 요약

OAuth2 Resource Server는 외부에서 발급한 액세스 토큰(Access Token)을 검증하고, **인증된 사용자로부터 보호된 자원(Resource)**에 대한 접근을 제어하는 역할을 한다.

- **Resource Server**는 인증 기능을 하지 않음
- 토큰은 외부 **Authorization Server**(예: Keycloak, Auth0, Okta 등)에서 발급
- Resource Server는 주어진 토큰의 유효성을 검증하고, 해당 사용자의 권한(claims)을 기반으로 접근을 제어

시스템 구성

```
[Client] -- Access Token --> [Resource Server (Spring Boot API)]
                                |
                [Authorization Server (Keycloak, Auth0...)]
```

---

2. Spring Boot 구성 개요

필수 의존성 (Gradle 예시)

```
dependencies {
  implementation 'org.springframework.boot:spring-boot-starter-oauth2-resource-server'
  implementation 'org.springframework.boot:spring-boot-starter-security'
}
```

---

3. 기본 설정 방법 (`application.yml`)

✅ JWT 기반 토큰 사용 예시

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: https://auth.example.com/realms/myrealm
```

> `issuer-uri`는 JWT 토큰의 **iss (issuer)** 클레임 값과 매칭되는 URL로, `.well-known/openid-configuration`을 통해 메타 정보를 받아와 공개 키를 획득하고 토큰을 검증한다.

---

4. SecurityFilterChain 설정

```
@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception
{
        http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/public/**").permitAll()
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2
                .jwt() // JWT 방식 명시
            );
        return http.build();
    }
}
```

---

5. JWT Claim 매핑

기본적으로 `sub`, `scope`, `authorities` 등의 claim이 Spring Security의 **Principal 정보**로 매핑된다. 더 자세히 설정하려면 **Converter**를 커스터마이징해야 한다.

```
@Bean
public JwtAuthenticationConverter jwtAuthenticationConverter() {
    JwtGrantedAuthoritiesConverter authoritiesConverter = new
JwtGrantedAuthoritiesConverter();
    authoritiesConverter.setAuthorityPrefix("ROLE_");
    authoritiesConverter.setAuthoritiesClaimName("roles");

    JwtAuthenticationConverter converter = new JwtAuthenticationConverter();
    converter.setJwtGrantedAuthoritiesConverter(authoritiesConverter);
    return converter;
}
```

그리고 `SecurityFilterChain`에 추가:

```
.oauth2ResourceServer(oauth2 -> oauth2
    .jwt(jwt -> jwt
        .jwtAuthenticationConverter(jwtAuthenticationConverter())
    )
);
```

---

6. 토큰 검증 흐름

1. 클라이언트가 Authorization Server로부터 Access Token을 발급받음
2. 해당 토큰을 Resource Server에 요청 헤더에 포함하여 전송

   ```
   Authorization: Bearer eyJhbGciOiJIUzI1...
   ```

3. Resource Server는 `.well-known/openid-configuration`에서 가져온 공개 키로 JWT를 검증
4. 검증이 성공하면 SecurityContext에 인증 정보를 설정
5. 컨트롤러에서 `@AuthenticationPrincipal`이나 `SecurityContextHolder`를 통해 사용자 정보 사용 가능

---

7. 사용자 정보 접근 예시

```
@GetMapping("/user/info")
public ResponseEntity<?> getUserInfo(@AuthenticationPrincipal Jwt jwt) {
    String username = jwt.getSubject(); // "sub" claim
    List<String> roles = jwt.getClaimAsStringList("roles");

    return ResponseEntity.ok(Map.of("username", username, "roles", roles));
}
```

---

8. 실전 주의 사항

항목	설명
시간 동기화	JWT 토큰의 `exp`, `iat` 검증 시 시스템 시간이 중요함 (서버 시간 오차 허용 범위 내 유지 필요)
토큰 무효화	JWT는 상태가 없기 때문에, 무효화된 토큰 관리가 어렵다. Redis 또는 블랙리스트를 함께 사용하기도 함
HTTPS 필수	액세스 토큰은 민감 정보. 반드시 HTTPS 환경에서만 사용해야 함
토큰 크기	JWT는 헤더에 담기기 때문에 크기 제한 유의. 너무 많은 Claim은 X

---

9. 토큰 유형 비교

항목	Bearer Token (JWT)	Opaque Token
내용 해석	가능 (Base64 디코딩)	불가능
검증 방식	서명 확인 (자체 검증)	introspection endpoint 요청
Spring 지원	✅ `jwt` 방식	✅ `opaque-token` 방식
성능	빠름 (자체 검증)	느림 (원격 호출 필요)

---

10. OAuth2 Resource Server vs OAuth2 Client

항목	Resource Server	OAuth2 Client
목적	보호된 리소스를 제공	외부 리소스를 소비
대상	API 서버	브라우저, 웹 앱
주요 클래스	`spring-security-oauth2-resource-server`	`spring-security-oauth2-client`
인증 주체	토큰 기반	로그인 세션 기반

분산 환경에서의 트랜잭션 처리 (Saga, Eventual Consistency)

1. 배경: 왜 분산 트랜잭션이 필요한가?

단일 시스템 내 트랜잭션

- JPA, JDBC 등을 통한 **로컬 트랜잭션**은 `@Transactional`로 쉽게 처리 가능
- 동일 DB, 동일 서비스 내부에서만 유효함

문제 발생 지점

- 여러 **마이크로서비스**가 각자 **자신의 DB**를 가지는 구조에서
- 서로 다른 서비스 간에 **동시에 성공해야 하는 작업**이 필요할 때 발생

> 예: 주문 서비스 → 결제 서비스 → 배송 서비스
> 하나라도 실패하면 전체를 되돌려야 하는데, **DB 간 롤백은 기본적으로 불가능**

---

2. 전통적인 해결 방식: 분산 트랜잭션 (XA, 2PC)

특징	설명
XA 프로토콜	Two-Phase Commit (2PC) 기반의 분산 트랜잭션
장점	원자성 보장 (모두 성공 or 모두 실패)
단점	느림, 복잡함, 확장성 저하, 장애 시 위험

> 따라서 마이크로서비스에서는 보통 **Saga 패턴**이나 **이벤트 기반의 Eventually Consistent 패턴**으로 대체함

---

3. ✅ Saga 패턴 (Choreography & Orchestration)

개요

- **Saga**는 각 서비스가 **자신의 로컬 트랜잭션을 수행**하고,
- 이후 다른 서비스에 **이벤트 또는 명령(Command)**를 전달해 전체 흐름을 조율하는 패턴

2가지 모델

방식	설명
Choreography	이벤트 기반, 서비스들끼리 서로 이벤트를 구독하고 반응
Orchestration	중앙 컨트롤러(Saga Coordinator)가 전체 흐름을 제어

Choreography Saga 구조

```
[Order Service]
  ↓ OrderCreatedEvent
[Payment Service]
  ↓ PaymentCompletedEvent
[Inventory Service]
  ↓ InventoryReservedEvent
[Shipping Service]
  ↓ ProductShippedEvent
```

- 장점: 탈중앙화, 느슨한 결합
- 단점: 서비스 간 이벤트 관계가 많아지면 **스파게티 구조**로 발전 가능

Orchestration Saga 구조

```
[Saga Coordinator]
  → Start Order
  ← Order Success / Fail
  → Trigger Payment
  ← Payment Success / Fail
  ...
```

- 장점: 전체 흐름이 명확, 디버깅 쉬움
- 단점: 중앙 집중형이므로 **Coordinator 장애가 치명적**

---

4. 보상 트랜잭션 (Compensating Transaction)

Saga에서는 **롤백 대신 보상**을 사용함.

예시:

1. 주문 생성 성공
2. 결제 성공
3. 배송 실패 → "결제 취소"라는 보상 작업 수행
4. 상태를 "주문 실패"로 변경

```
public void compensatePayment(String orderId) {
    paymentService.cancelPayment(orderId);
}
```

---

5. Eventually Consistent (최종 일관성)

개념

- 트랜잭션의 일관성을 강제로 맞추는 대신,
- 시간이 좀 지나더라도 **결국 상태가 일관되게 수렴되도록 설계**

예시

- Kafka 등 **메시지 브로커**를 통해 이벤트를 비동기 전달
- 각 서비스는 자신이 받은 메시지 기반으로 상태를 맞춰감

```
@KafkaListener(topics = "order-created")
public void handleOrderCreated(OrderEvent event) {
    // 상태 맞춤, 로컬 DB 반영
}
```

---

6. Spring Cloud 기반 구현

핵심 구성 요소

- **Spring Cloud Stream**: Kafka, RabbitMQ 기반 이벤트 발행/구독
- **Spring State Machine**: Saga Coordinator의 상태 관리
- **Spring Boot + @Transactional**: 각 로컬 트랜잭션 관리
- **Kafka**: Saga 이벤트 전달 수단
- **Dead Letter Topic (DLT)**: 실패한 이벤트 재처리 또는 보상용

---

7. Saga 상태 관리 예시 (Orchestration)

```
enum OrderStatus {
  PENDING, PAID, INVENTORY_RESERVED, SHIPPED, FAILED
}
```

상태 전이 흐름

```
PENDING
  ↓ (PaymentSuccess)
PAID
  ↓ (InventoryReserved)
INVENTORY_RESERVED
  ↓ (ShippingDone)
SHIPPED
```

실패 발생 시 → **보상 트랜잭션 수행 후 상태를 `FAILED`로 변경**

---

8. 트랜잭션 처리 전략 요약

전략	장점	단점	적용 상황
로컬 트랜잭션	빠름, 간단함	분산 불가	단일 DB
XA (2PC)	원자성 보장	성능 저하	레거시 시스템, 강한 일관성
Saga	확장성, 장애 견딤	복잡함, 보상 설계 필요	마이크로서비스
Eventually Consistent	확장성, 유연함	일시적 불일치 허용	비동기 이벤트 기반 시스템

---

9. 실전 적용 시 고려 사항

- 실패 시 **재시도/리트라이 전략** 명확히
- **Idempotency 보장**: 중복 메시지에 대비해야 함
- 상태 추적을 위한 **State Store** 도입 권장

- 메시지 순서 보장 필요 시 Kafka의 **파티션 키** 활용
- 장애 분석을 위한 **분산 트레이싱** 도입 (예: Sleuth + Zipkin)

---

10. 연관 도구

도구	역할
Kafka / RabbitMQ	이벤트 전달
Debezium	CDC 기반 이벤트 트리거
Axon Framework	Saga 구현 지원
Temporal.io / Camunda	워크플로우 기반 Orchestration
Spring Cloud Stream	이벤트 발행/소비

정리

- `@Transactional` 은 마이크로서비스에서는 **완전한 트랜잭션 제어 불가**
- Saga 패턴을 사용하면 **서비스 간 협력 기반의 트랜잭션 관리** 가능
- Eventually Consistent 구조는 **성능과 유연성은 뛰어나지만**, **불일치 관리 책임**이 서비스에 있음
- Spring에서는 `Spring Cloud Stream`, `Kafka`, `State Machine`, `Resilience4j` 등을 통해 robust한 구현 가능