20. 스프링 클라우드 (Spring Cloud)

마이크로서비스 아키텍처 개요

1️⃣ 마이크로서비스 아키텍처란?

정의

- 하나의 **대규모 애플리케이션**을 작고 독립적으로 배포 가능한 서비스들의 모음으로 구성하는 아키텍처 스타일
- 각 서비스는 **자율적**으로 개발, 배포, 확장 가능
- 서비스들은 일반적으로 **REST API / 메시징** 기반으로 통신

2️⃣ MSA vs 모놀리식(Monolithic) 아키텍처

구분	모놀리식 (Monolithic)	마이크로서비스 (Microservice)
서비스 구성	단일 애플리케이션	여러 개의 독립적인 서비스
배포 단위	하나	서비스별 개별 배포
확장성	전체 애플리케이션 확장	서비스 단위 개별 확장
장애 전파	전체 서비스에 영향	특정 서비스만 영향
기술 스택	통일된 기술 스택 사용	서비스별 최적의 기술 스택 사용 가능
개발팀 조직	하나의 팀 중심 개발	서비스별 독립 팀 운영 가능

3️⃣ 주요 특징

1. 독립 배포 가능성

- 서비스는 독립적으로 배포됨 (CI/CD 파이프라인 구성 용이)

2. 서비스 경계 명확화

- 각 서비스는 **명확한 경계(Context Boundary)** 를 가짐 → 도메인 주도 설계(DDD)와 잘 결합됨

3. 자율성

- 각 서비스는 독립된 **DB / 기술 스택 / 개발팀**을 가질 수 있음

4. 고가용성 & 확장성

- 서비스별 **스케일 업 / 스케일 아웃** 가능

5. 장애 격리

- 하나의 서비스 장애가 전체 시스템으로 전파되지 않도록 설계 가능

4️⃣ MSA 기본 구성요소

구성 요소	설명
API Gateway	클라이언트 요청을 수집하고 라우팅
서비스 디스커버리 (Service Discovery)	서비스 인스턴스 동적 등록 및 탐색
Configuration Server	서비스 구성 정보 중앙 관리
분산 트랜잭션 관리	서비스 간 데이터 정합성 확보
서비스간 통신	REST, gRPC, 메시지 큐(Kafka, RabbitMQ) 사용
중앙 로깅 / 모니터링	분산 환경에서 로그, 메트릭 수집 및 모니터링
보안	OAuth2, JWT 기반 인증/인가 구현
테스트 전략	서비스 단위 테스트, 계약 기반 테스트(Contract Testing) 적용

5️⃣ 스프링 기반 MSA 구성 예시

역할	기술 스택 예시
API Gateway	Spring Cloud Gateway / Zuul
Service Discovery	Eureka / Consul
Config Server	Spring Cloud Config
Inter-Service Communication	REST (Feign), 메시징 (Kafka)
Circuit Breaker	Resilience4j
Distributed Tracing	Spring Cloud Sleuth + Zipkin/Jaeger
Centralized Logging	ELK Stack (Elasticsearch, Logstash, Kibana)
Security	Spring Security + OAuth2 + JWT

6️⃣ MSA 도입의 장점과 단점

장점

- 유연한 확장성
- 장애 격리 가능
- 빠른 독립 배포 가능
- 다양한 기술 스택 활용 가능
- 조직 구조와 잘 매핑 (팀 단위 서비스 소유)

단점

- 시스템 복잡성 증가
- 서비스 간 통신 및 데이터 정합성 관리 필요
- 운영/모니터링/배포 환경 고도화 필요
- 개발/테스트 난이도 상승
- 초기 설계 비용 증가

7️⃣ 전형적인 MSA 예시 아키텍처

```
+------------+
|   Client   |
+------------+
       |
       v
+----------------+                     +-------------------+
|  API Gateway   |-------------------->|  Auth Service     |
+----------------+                     +-------------------+
       |                               +-------------------+
       |------------------------------>|  User Service     |
       |                               +-------------------+
       |------------------------------>|  Order Service    |
       |                               +-------------------+
       |------------------------------>|  Payment Service  |
       |                               +-------------------+
       |------------------------------>|  Notification Svc |
                                       +-------------------+

Supporting Components:

+-----------------------+
| Configuration Server  |
+-----------------------+

+---------------------------+
| Eureka (Service Registry) |
+---------------------------+

```

```
+-----------------------+
| Kafka (Event Bus)     |
+-----------------------+

+-----------------------+
| Redis (Caching)       |
+-----------------------+

+-----------------------+
| Centralized Logging   |
| (ELK Stack or similar)|
+-----------------------+

+-----------------------+
| Zipkin (Tracing)      |
+-----------------------+

```

---

8 MSA 도입 시 고려 사항

- ✅ 조직/팀 구성부터 MSA 친화적으로 운영 필요
- ✅ 배포 자동화 필수 (CI/CD Pipeline)
- ✅ 분산 환경에서 장애 탐지 / 복구 체계 구축 필요
- ✅ 데이터 분산/일관성 처리 전략 설계
- ✅ 서비스간 통신 방식 (동기 vs 비동기) 설계

---

9 적용 시기 가이드

상황	추천 아키텍처
작은 스타트업, 초기 MVP	모놀리식 우선
중견 이상 조직, 다수 개발팀 병렬 운영	MSA 적극 고려
높은 서비스 안정성과 확장성 요구	MSA 우선 고려
서비스가 명확한 경계로 나뉘는 경우	MSA 적합

---

10 결론

MSA는 **기술적 장점**뿐만 아니라
조직적 민첩성을 높이고 **서비스 품질과 확장성**을 확보하는 강력한 설계 패턴이다.

단, **운영 복잡성**과 **기술적 난이도**가 높아지므로
충분한 준비와 경험 축적 후 단계적 전환하는 접근이 바람직하다.

Eureka (Service Discovery)

1 Service Discovery란?

기본 개념

- 마이크로서비스 환경에서는 **서비스 인스턴스가 동적으로 늘고 줄고** 함
- 서비스 인스턴스가 어느 서버에 어떤 포트에서 동작하는지 **고정적으로 알 수 없음**
- 그래서 **서비스 레지스트리(Registry)** 라는 중앙 목록에 서비스들이 **자기 자신을 등록하고**, 다른 서비스는 거기서 **위치 정보를 조회**함 → Service Discovery

주요 역할

- **서비스 등록 (Service Registration)** → 서비스가 부팅 시 Registry 에 자신 등록
- **서비스 조회 (Service Lookup)** → 클라이언트가 해당 서비스의 위치(호스트/포트)를 Registry 에 질의
- **상태 체크 (Heartbeat)** → 서비스가 주기적으로 살아있음을 알림
- **레지스트리 갱신** → 더 이상 살아있지 않은 서비스는 제거됨

---

2 Eureka란?

- 넷플릭스(Netflix)가 개발한 **Service Discovery Server**
- Spring Cloud 에서 **Spring Cloud Netflix Eureka** 로 통합 제공
- REST 기반으로 동작
- 매우 가벼우며 **Spring Boot + Eureka**는 손쉽게 연동 가능

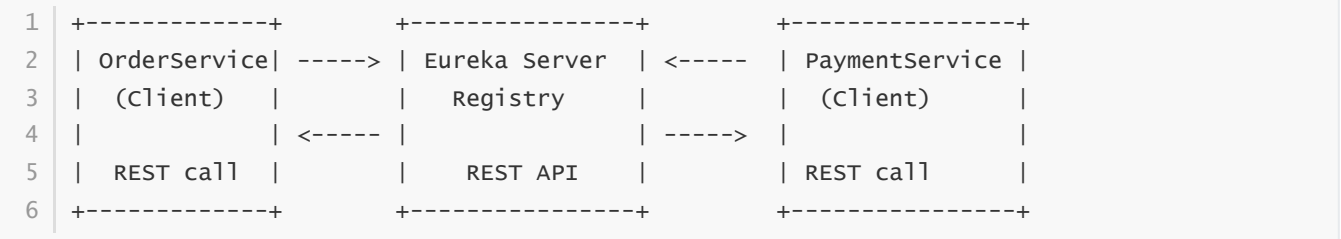
---

3 구성요소

구성 요소	역할
Eureka Server	Service Registry 역할 (중앙 서버)
Eureka Client	서비스 애플리케이션이 등록 및 조회 요청 (주로 마이크로서비스)
Eureka Dashboard	서비스 상태 확인 UI 제공 (웹 UI)

---

4 기본 동작 흐름



1 서비스 애플리케이션 (OrderService, PaymentService) → Eureka Server에 등록

2️⃣ 서비스들은 주기적으로 Heartbeat 전송 (lease 갱신)

3️⃣ 서비스들은 Eureka Server에 **다른 서비스 위치를 조회**해서 직접 REST 호출

5️⃣ Eureka 구성 방법 (Spring Boot 기반 예제)

5.1 Eureka Server 설정

1️⃣ 의존성 추가

```
<!-- build.gradle 예시 -->
implementation 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-server'
```

2️⃣ 서버 애플리케이션 설정

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

3️⃣ application.yml 설정

```
server:
  port: 8761

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

4️⃣ 결과

- http://localhost:8761 으로 Eureka Dashboard 확인 가능
- 등록된 서비스 목록 조회 가능

5.2 Eureka Client 설정

1️⃣ 의존성 추가

```
<!-- build.gradle 예시 -->
implementation 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client'
```

2️⃣ application.yml 설정

```yaml
spring:
  application:
    name: order-service

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

3️⃣ 결과

- OrderService 가 Eureka Server 에 등록됨
- 다른 서비스에서 order-service 를 Eureka 통해 조회 가능

6️⃣ Eureka Heartbeat 및 Registry 관리

- **Heartbeat 주기**: 기본 30초
- **Registry 갱신 주기**: 기본 30초
- **Lease 만료 시간**: 기본 90초
    - 서비스가 다운되어 Heartbeat 가 끊기면 90초 후 레지스트리에서 제거됨

7️⃣ Eureka의 장점

✅ 간단하고 가벼운 구성
✅ Spring Cloud 와 자연스럽게 통합
✅ REST 기반으로 작동 → HTTP API 친숙
✅ 서비스 상태 모니터링 Dashboard 제공

8️⃣ Eureka의 단점 및 한계

❌ Netflix 가 **2021년 이후로 메인테인 중단**
❌ 동적 확장성 측면에서 Kubernetes 서비스 디스커버리보다 유연성 떨어짐
❌ Strong Consistency 보장 안됨 (Eventually Consistent) → 복제 중간에 레지스트리 차이 가능성 존재
❌ Spring Cloud Netflix 전반적으로 **기능 축소 단계** (→ 대안: Consul, Nacos, Kubernetes 서비스 디스커버리 사용 증가 중)

9️⃣ Eureka 운영 시 고려사항

✅ **Eureka Server HA 구성 필요** → 최소 2대 구성 (Cluster)
✅ **Eureka Client retry 설정** 잘 구성 (네트워크 장애 대비)
✅ **서비스 탐색 API 호출 시 Cache 활용 가능**
✅ **서비스 명세(Contract) 관리 필요** → API 변경에 주의

1️⃣0️⃣ 대안 기술

기술	특징
Consul (HashiCorp)	Service Discovery + Key-Value Store + Health Check 기능
Nacos (Alibaba)	서비스 등록/조회 + 구성 관리
Kubernetes Service	Kubernetes 환경에서 기본 제공 (DNS 기반 서비스 디스커버리)
Zookeeper	Strong consistency 기반 Service Registry

→ 현재는 **Kubernetes + Istio/Linkerd + Kubernetes Service/Endpoint 기반 Service Discovery** 사용이 주류

1️⃣1️⃣ 결론

- **Eureka**는 MSA 환경에서 가장 쉽게 **Service Discovery**를 구축할 수 있는 훌륭한 도구였음
- **Spring Boot + Eureka** 조합은 빠른 학습과 구축에 적합
- 운영 환경에서는 **HA 구성**, **클러스터 설정** 필수
- Kubernetes 환경에서는 **Kubernetes Service 기반 Discovery**를 우선 고려하는 것이 최근 트렌드

Config Server / Gateway / Zuul

1️⃣ Config Server

역할

- **중앙 집중형 구성 관리 서버**
- 각 서비스의 `application.yml` 또는 `application.properties` 를 중앙에서 관리
- 서비스가 실행될 때 Config Server 에서 설정을 **자동으로 읽어옴**
- **프로파일(profile)** 별로 구성 관리 가능 → `dev`, `test`, `prod` 환경 분리

Spring Cloud Config Server 구성

기능	설명
중앙 설정 관리	Git / SVN / File System 기반 설정 저장소 지원
동적 갱신	Spring Cloud Bus + Kafka/RabbitMQ 를 이용한 설정 동적 반영 지원
Profile 지원	`application-dev.yml`, `application-prod.yml` 등 환경별 구성 지원
보안 관리	구성 정보에 보안 정보 포함 가능 (암호화 지원)

동작 흐름

```
+-----------------+         +-------------------------+
|   Application   |  --->   | Spring Cloud Config Server |
+-----------------+         +-------------------------+
                                    |
                                    v
                        Git / File / SVN Repository
```

간단한 설정 예시

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/my-org/config-repo
          clone-on-start: true
```

- 클라이언트 서비스에서는:

```
spring:
  config:
    import: "configserver:"
```

2️⃣ API Gateway

역할

- 클라이언트 요청을 내부 서비스로 **라우팅**
- 서비스의 위치를 클라이언트가 알 필요 없음 → Gateway가 대신 통신
- 공통 기능을 Gateway에 위임:
  - 인증/인가
  - 로깅
  - 트래픽 제어 (Rate Limiting)
  - CORS 처리
  - Monitoring / Metrics 수집
  - 요청/응답 변환

Spring Cloud Gateway

- Spring Cloud 에서 **Zuul의 후속 기술**
- **Netty 기반** → 성능 우수
- **비동기 / Reactive 지원**

주요 기능

기능	설명
Path Routing	URI 경로 기반으로 서비스 라우팅
Load Balancing	Spring Cloud LoadBalancer / Eureka 연동
Filter 적용	Pre / Post / Global Filter 지원
Rate Limiting	API 호출 제한 기능
Circuit Breaker	Resilience4j / Hystrix 연계 가능
Security	OAuth2 / JWT 기반 보안 필터 적용 가능

기본 흐름

```
+---------+      +----------------+      +------------------+
| Client  | ---> | API Gateway    | ---> | Internal Services|
+---------+      +----------------+      +------------------+
```

설정 예시

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: user-service
          uri: lb://user-service
          predicates:
            - Path=/users/**
```

- `lb://` → Eureka 와 연동된 서비스명 사용

3️⃣ Zuul

Zuul이란?

- 넷플릭스에서 개발한 **API Gateway/Edge Service**
- 주로 **Zuul 1** (Servlet 기반) 사용됨 → 동기 기반
- 현재는 **Zuul 2**가 개발되었지만 Spring Cloud 는 Gateway 로 전환 중

Zuul 1 vs Spring Cloud Gateway 비교

항목	Zuul 1	Spring Cloud Gateway
아키텍처	Servlet 기반	Netty 기반 (비동기)
성능	비교적 낮음 (블로킹 I/O)	매우 우수 (Non-blocking)

항목	Zuul 1	Spring Cloud Gateway
지원	넷플릭스 Zuul 1는 메인테인 중단	Spring에서 Gateway 적극 지원
필터	Pre / Post / Route / Error Filter	Global / Pre / Post / Custom Filter
호환성	Spring Boot 1.x / 2.x	Spring Boot 2.x / 3.x 이후 추천

Zuul 사용 시 장점

- 구축이 매우 쉬움 (Spring Cloud Netflix Zuul)
- Zuul 자체에 필터 체인 개념 제공

Zuul 간단한 설정 예시

```
spring:
  application:
    name: gateway-service

zuul:
  routes:
    user-service:
      path: /users/**
      serviceId: user-service
```

- `serviceId` 를 통해 Eureka 에 등록된 서비스로 라우팅

---

4 통합 아키텍처 그림

```
+------------+          +-----------------+           +----------------------+
|   Client   |  ----->  | API Gateway     |  ----->   | Auth Service         |
+------------+          | (Zuul or SCG)   |  ----->   | User Service         |
                        +-----------------+           | Order Service        |
                        |  Config Server  |  ----->   | Payment Service      |
                        +-----------------+           | Notification Service |
                                                      +----------------------+

+----------------------------+
| Eureka (Service Registry)  |
+----------------------------+

+----------------------------+
| Kafka / RabbitMQ (optional)|
+----------------------------+

+----------------------------+
| Centralized Logging (ELK)  |
+----------------------------+

+----------------------------+
```

```
| Zipkin / Jaeger (Tracing)  |
+----------------------------+
```

5️⃣ 결론

구성 요소	역할
Config Server	구성 관리 중앙화 (Git 기반 설정 관리)
API Gateway (SCG or Zuul)	API 라우팅, 공통 기능 처리 (Auth, Logging 등)
Zuul	Servlet 기반의 API Gateway (레거시에 적합)
Spring Cloud Gateway	차세대 Gateway, 비동기 성능 우수, 신규 서비스에 권장

👉 **현재 권장 방향**:

- 새로운 시스템 구축 시에는 **Spring Cloud Gateway** 우선 사용
- Zuul 은 **레거시 시스템 유지 보수**에서만 선택

👉 Config Server 는 **MSA 환경에서 거의 필수 인프라 구성 요소**로 사용

API Gateway 인증 통합

1️⃣ 왜 API Gateway에서 인증을 처리할까?

전통적 구조 문제점

- 서비스마다 인증 기능을 중복 구현해야 함
- 동일한 인증/인가 로직이 각 서비스에 중복됨
- 변경 시 각 서비스 모두 수정 필요

Gateway 인증 처리 장점

- 클라이언트 요청이 API Gateway를 반드시 거치므로 **공통 인증 적용 가능**
- 인증 성공 여부에 따라 내부 서비스 접근 여부 결정
- 서비스는 **순수 비즈니스 로직에 집중** 가능 → 인증 부담 제거
- JWT 기반 사용 시 **Stateless 인증 구조 구현 용이**

---

2 API Gateway 인증 통합 구성

```
+------------+    HTTP + TOKEN    +-----------------+    INTERNAL CALL    +--------------
--+
|   Client   | ---------------->  |   API Gateway   | -----------------> | Internal
Service|
+------------+                    +-----------------+                     +---------------
-+


                                    |
                                    v
                          +---------------------+
                          | Auth Service (Token) |
                          +---------------------+
```

흐름 설명

1. Client → API Gateway 에 요청 시 **Access Token (ex: JWT)** 포함
2. API Gateway 는 토큰 검증 수행 (자체 검증 or Auth Service 호출)
3. 검증 성공 시 Internal Service 로 요청 전달
4. Internal Service 에서는 별도의 인증 처리 생략 가능 (단, 인가 검증은 필요할 수 있음)

---

3 인증 통합 시 기술적 구성 요소

요소	주요 기술 예시
API Gateway	Spring Cloud Gateway (SCG), Zuul, Kong, Nginx
인증 프로토콜	OAuth2, OpenID Connect, JWT 기반
토큰 유형	JWT (Access Token), Refresh Token
Auth Provider	OAuth2 Provider (Keycloak, Auth0, Cognito, 자체 구현)
Token 검증 방법	Gateway에서 검증 or Auth Service에 검증 위임

---

4 인증 처리 방법 패턴

방법 1: API Gateway에서 직접 JWT 검증

- Gateway 에 **Public Key (RSA/EC)** 설정
- Gateway 필터에서 JWT Signature 검증
- 검증 후 유효한 요청만 Internal Service 에 전달
- 매우 빠르고 확장성 높음

방법 2: API Gateway가 Auth Service에 검증 요청

- Gateway가 Auth Service API 를 호출해 Token 유효성 검증
- 장점: 중앙화된 검증 로직 사용
- 단점: Gateway <-> Auth Service 호출 시 오버헤드 발생

방법 3: Hybrid 패턴

- Gateway는 **Signature 검증** + 일부 Claim 확인 → 빠른 필터링
- 필요한 경우 Auth Service 로 추가 검증 요청 (ex: Token Blacklist 여부 확인 등)

5️⃣ Spring Cloud Gateway + 인증 통합 예시

필터 구성 흐름

```
@Component
public class AuthenticationFilter implements GlobalFilter, Ordered {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {

        ServerHttpRequest request = exchange.getRequest();

        // 1. 토큰 추출
        String token = extractToken(request);

        // 2. 토큰 검증
        if (isValid(token)) {
            // 3. 인증 정보 Context 에 주입 (필요 시)
            return chain.filter(exchange);
        } else {
            exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
            return exchange.getResponse().setComplete();
        }
    }

    // 구현 예시
    private String extractToken(ServerHttpRequest request) {
        return request.getHeaders().getFirst(HttpHeaders.AUTHORIZATION)
                .replace("Bearer ", "");
    }

    private boolean isValid(String token) {
        // JWT signature 검증 or Auth Service 호출
        return JwtUtils.verify(token); // 예시
    }

    @Override
    public int getOrder() {
        return -1; // 높은 우선순위
    }
```

```
}
```

6 실전 구성 시 고려사항

필수 고려 요소

- ✅ **Token Expiration 관리**
- ✅ **Refresh Token 전략 필요** (주기적 재발급)
- ✅ **Token Revocation 관리** (Logout / Token Blacklist 등)
- ✅ **Public Key 교체 (Key Rotation)** 관리 필요
- ✅ **Authorization 정보 전달 방식** (ex: role/authority Claim 전달 여부)
- ✅ **Internal Service 에 전달 시 최소한의 정보만 포함 권장**

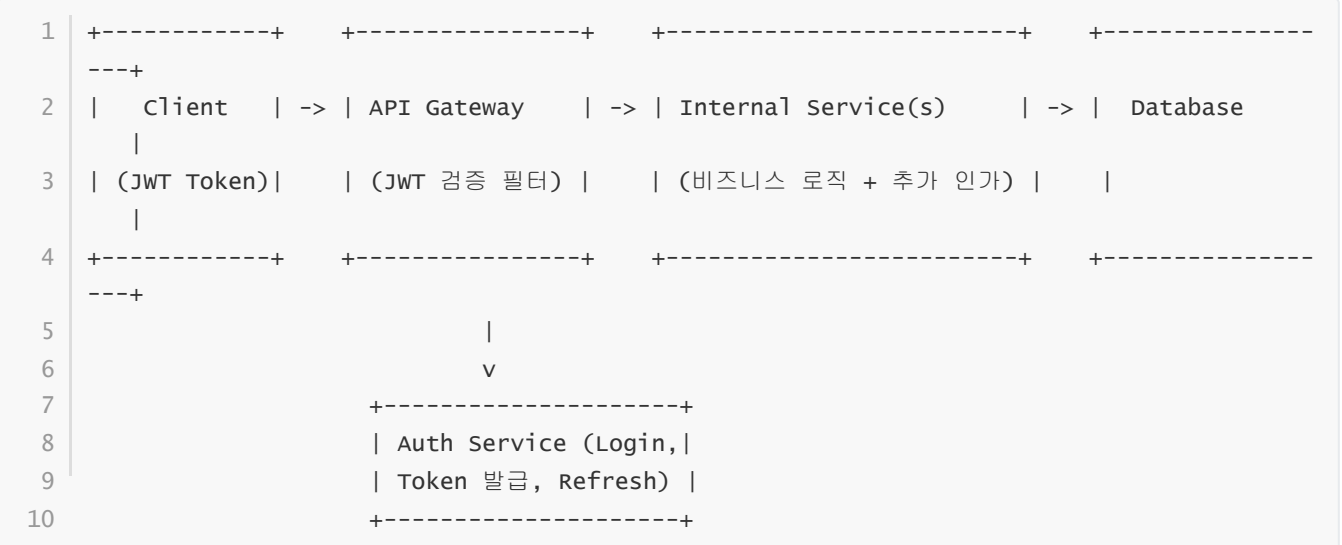
7 인가(Authorization) 처리 패턴

위치	처리 방법
API Gateway	최소 Role 기반 인가 (ex: ADMIN만 특정 서비스 접근 허용)
Internal Service	상세 비즈니스 인가 (ex: 특정 사용자만 자신의 리소스 접근 허용)

Best Practice

- **인증은 Gateway에서 담당**
- **인가의 세부 비즈니스 로직은 Service 내부에서 추가 확인**

8 API Gateway 인증 통합 전체 흐름 예시 (JWT 기반)



9️⃣ 대표 구성 예시

구성 요소	추천 기술
API Gateway	Spring Cloud Gateway (최신 시스템)
인증 프로토콜	OAuth2 + JWT
Token Provider	Keycloak / Auth0 / Cognito / 자체 구축 Auth Service
Token 검증 방법	SCG 필터에서 직접 JWT 검증 + 필요시 Auth Service 호출
Token 전달	Authorization: Bearer {AccessToken} 헤더 사용
Refresh Token 처리	Auth Service 전담 처리
내부 서비스 Role 전파	JWT Claim 통해 전달 or Request Header 에서 필요한 Claim 추출 후 전달

1️⃣0️⃣ 결론

API Gateway 인증 통합 설계 시 핵심 원칙:

✅ 인증은 API Gateway 단에서 통합 처리 → 내부 서비스 인증 부담 제거
✅ 인가는 서비스별로 적절히 분리 설계 (Gateway 단 coarse-grained, Service 단 fine-grained)
✅ JWT 기반 인증으로 **Stateless 구조** 설계 권장
✅ Token 관리 전략 (Expiration, Revocation, Rotation) 명확하게 수립 필요
✅ Gateway 성능 고려 시 **Signature 기반 Local 검증 + 필요 시 Auth Service 호출 Hybrid 구성 추천**

Spring Cloud Bus, Sleuth, Zipkin

1️⃣ Spring Cloud Bus

개념

- MSA 환경에서 **구성 변경이나 이벤트 메시지**를 **서비스 전체에 브로드캐스트**하는 역할
- **Kafka / RabbitMQ** 같은 메시지 브로커를 통해 서비스 간 메시지 전파

주 용도

용도	설명
Config 변경 반영	Config Server 변경 사항을 전체 서비스에 실시간으로 반영
서비스 간 이벤트 전파	상태 변경, 캐시 무효화, 사용자 이벤트 등 서비스 간 전파
확장성	서비스 수가 많아도 Bus 기반으로 효율적으로 메시지 전파 가능

흐름 예시 (Config 변경 시)

```
Developer pushes config change → Config Server Git Repo
      ↓
Spring Cloud Config Server 감지
      ↓
Spring Cloud Bus 메시지 발행 (Kafka / RabbitMQ)
      ↓
각 서비스 수신 → /actuator/bus-refresh 호출 → config reload
```

주요 구성

- Spring Cloud Config Server + Spring Cloud Bus + Kafka/RabbitMQ
- 서비스는 `@RefreshScope` 사용 → 동적 갱신 가능

간단한 의존성 예시

```
implementation 'org.springframework.cloud:spring-cloud-starter-bus-amqp'   // RabbitMQ 기반
implementation 'org.springframework.cloud:spring-cloud-starter-bus-kafka'  // Kafka 기반
```

2️⃣ Spring Cloud Sleuth

개념

- 분산 시스템(Distributed System)에서 **서비스 간 호출 추적(Tracing)** 기능 제공
- 각 서비스의 로그에 **Trace Id / Span Id** 를 자동 삽입
- 전체 서비스 호출 경로를 추적할 수 있음

주 용도

용도	설명
서비스 간 호출 경로 추적	HTTP/gRPC 호출 흐름 시각화
문제 디버깅	어디에서 지연 발생하는지 파악
성능 분석	서비스별 응답 시간 분석

핵심 개념

개념	설명
Trace Id	하나의 전체 요청 흐름 식별자 (end-to-end)
Span Id	개별 작업(Span) 식별자 (HTTP 호출, DB 쿼리 등)
Parent Span Id	상위 Span 식별자 (호출 관계 구성)

흐름 예시

```
Client Request (Trace Id: X) → API Gateway → Order Service → Payment Service → Inventory Service

→ 각 단계의 로그에 동일한 Trace Id 가 찍힘
→ Span Id 는 각 단계마다 다름
```

자동 연동

- Sleuth 를 사용하면 **로그 MDC (Mapped Diagnostic Context)** 에 TraceId, SpanId 가 자동 삽입됨
- ex:

```
2025-06-12 10:15:30 [X-TraceId abc12345] [SpanId 789xyz] GET /orders/1
```

의존성 예시

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

---

3 Zipkin

개념

- **분산 추적 데이터를 수집/저장/시각화** 하는 오픈소스 플랫폼
- Sleuth 와 함께 사용 → Sleuth 가 Zipkin 에 데이터를 전송

주요 기능

기능	설명
Trace 시각화	서비스 호출 흐름 트리 형태로 시각화
호출 지연 구간 분석	서비스별 응답 시간 분석
병목 탐지	느린 서비스 또는 호출 구간 탐색
문제 진단	장애 시 호출 흐름 확인

구성 흐름

```
Service A → Sleuth → Zipkin Collector
Service B → Sleuth → Zipkin Collector
...
Zipkin UI → http://localhost:9411 → Trace 조회 가능
```

구성 요소

구성 요소	역할
Zipkin Collector	Trace 데이터 수집
Storage (ex: MySQL, Elasticsearch, InMemory)	Trace 데이터 저장
Zipkin UI	Trace 시각화 제공 (웹 UI)

Sleuth + Zipkin 연동 예시

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

application.yml:

```
spring:
  zipkin:
    base-url: http://localhost:9411
  sleuth:
    sampler:
      probability: 1.0   # 100% 수집 (운영환경에서는 낮게 설정 추천)
```

→ Zipkin 서버는 Docker 로도 쉽게 실행 가능:

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

→ 웹 UI: http://localhost:9411

4️⃣ 세 가지 통합 흐름 (전체 아키텍처)

```
+------------+       +-----------------+       +-----------------+
|   Client   | --->  | API Gateway     | --->  | Service A       |
+------------+       +-----------------+       +-----------------+
                          |                         |
                          |                         |
                          v                         v
                     Spring Cloud Sleuth         Spring Cloud Sleuth
                     + Trace Id / Span Id        + Trace Id / Span Id
                     + Logs                      + Logs
                          |                         |
                          +------------> Zipkin <----+

Config Change Flow:
Developer --> Git Repo --> Spring Cloud Config Server --> Spring Cloud Bus
(Kafka/RabbitMQ) --> All Services

→ Config 변경시 서비스 자동 갱신
→ 서비스 호출 흐름은 Sleuth + Zipkin 으로 추적
```

```
→ Logs 에는 TraceId/SpanId 자동 포함
```

5️⃣ 결론

구성 요소	역할	주 효과
Spring Cloud Bus	서비스 간 이벤트/Config 변경 전파	운영 효율성
Spring Cloud Sleuth	Trace Id/Span Id 기반 서비스 호출 추적	장애 분석, 병목 탐지
Zipkin	Trace 데이터 수집/시각화	호출 흐름 가시화

실전 Best Practice

✅ Config 변경 자동 반영 → **Config Server + Bus 필수**
✅ 서비스 호출 흐름 추적 → **Sleuth + Zipkin 연계 필수**
✅ Zipkin 에서 **100% Sampling 은 개발환경에서만 사용**
✅ 운영환경에서는 Sampling 비율을 조정 (ex: 0.1 ~ 0.01)