18. 모니터링 및 로깅

Actuator 사용법

Spring Boot Actuator 는 **운영 환경에서 애플리케이션 상태를 모니터링하고 관리하기 위한 기능**을 제공하는 라이브러리이다.

- ✅ Health Check
- ✅ Metrics 노출
- ✅ 환경 정보 확인
- ✅ Bean 목록 확인
- ✅ Thread dump
- ✅ Application Info 등

운영 중인 서비스의 **상태 가시성(Observability)** 확보를 위한 필수 도구로 활용된다.

1️⃣ 기본 구성

의존성 추가

Gradle:

```
implementation 'org.springframework.boot:spring-boot-starter-actuator'
```

Maven:

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

2️⃣ 기본 Endpoint 구성 🗂️

Spring Boot Actuator는 다양한 Endpoint 를 제공한다.
대표적인 Endpoint 예시:

Endpoint	설명
`/actuator/health`	Health check
`/actuator/metrics`	메트릭 데이터 노출
`/actuator/info`	Application Info 노출
`/actuator/env`	환경 변수 확인
`/actuator/beans`	Bean 목록 확인

Endpoint	설명
`/actuator/threaddump`	Thread dump
`/actuator/httptrace`	HTTP 요청 이력
`/actuator/mappings`	RequestMapping 정보 노출

3️⃣ application.yml 기본 설정 예시

```yaml
management:
  endpoints:
    web:
      exposure:
        include: '*'   # 모든 Endpoint 노출 (운영에서는 제한 권장)
  endpoint:
    health:
      show-details: always  # Health 상세 정보 노출
  info:
    env:
      enabled: true
```

운영 환경 Best Practice

```yaml
exposure:
  include: health, info, prometheus  # 필요한 Endpoint만 노출 권장
```

4️⃣ 주요 Endpoint 사용법 🚀

Health Check

```
GET /actuator/health
```

응답 예시:

```
{
  "status": "UP"
}
```

- DB 연결 문제 등 발생 시 `DOWN`, `OUT_OF_SERVICE` 상태 확인 가능.

Metrics

```
GET /actuator/metrics
```

응답 예시:

```
{
  "names": [
    "jvm.memory.used",
    "jvm.memory.max",
    "http.server.requests",
    "process.cpu.usage",
    "system.cpu.usage",
    "logback.events"
  ]
}
```

특정 Metrics 조회

```
GET /actuator/metrics/jvm.memory.used
```

Info

```
GET /actuator/info
```

- `application.yml` 에 info 값 추가 가능:

```
info:
  app:
    name: MyApp
    version: 1.0.0
```

Env

```
GET /actuator/env
```

- 전체 환경 변수 확인 가능 → 운영 환경에서는 주의 필요 ⚠️.

Beans

```
GET /actuator/beans
```

- 현재 ApplicationContext 내 Bean 목록과 의존성 정보 확인.

Thread Dump

```
GET /actuator/threaddump
```

- JVM Thread 상태 확인 → Deadlock, Blocked Thread 분석 시 유용.

5️⃣ 보안 구성 ⚠️

운영 환경에서는 반드시 Actuator Endpoint에 **접근 제한** 필요.

Spring Security 구성 예시:

```
management:
  endpoints:
    web:
      exposure:
        include: health, info, prometheus
  endpoint:
    health:
      show-details: when-authorized
  security:
    enabled: true
```

또는 Security Config 에서:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .requestMatcher(EndpointRequest.to("health", "info"))
        .authorizeRequests((requests) -> requests.anyRequest().permitAll());
}
```

6️⃣ Prometheus 연계 (Metrics Export) 📊

Prometheus scrape endpoint 활성화:

```
management:
  endpoints:
    web:
      exposure:
        include: prometheus
```

활성 후:

```
GET /actuator/prometheus
```

👉 Prometheus + Grafana 구성 시 → **Spring Boot Metrics 시각화 가능**.

7️⃣ 확장 구성 🚀

Custom Health Indicator

```
@Component
public class CustomHealthIndicator implements HealthIndicator {

    @Override
    public Health health() {
        // Custom Health Logic
        return Health.up().withDetail("custom", "OK").build();
    }
}
```

Custom Info Contributor

```
@Component
public class CustomInfoContributor implements InfoContributor {

    @Override
    public void contribute(Info.Builder builder) {
        builder.withDetail("customInfo", "My Custom Value");
    }
}
```

8️⃣ 결론 🗂️

✅ Spring Boot Actuator는 **운영 상태 가시성 확보**에 매우 중요한 도구다.
✅ 다양한 Endpoint 를 통해 **Health**, **Metrics**, **Thread 상태**, **Bean 구성** 등을 확인 가능.
✅ Prometheus 등 외부 모니터링 시스템과 연계 시 → **강력한 실시간 관찰(Observability) 플랫폼 구성 가능**.
✅ 운영 환경에서는 반드시 **Endpoint 노출 정책 및 보안 설정**을 적용할 것.

헬스체크, 메트릭, 트레이싱

Observability(관찰 가능성) 를 확보하려면 서비스의:

✅ **헬스 상태 확인** (Health Check)
✅ **운영 지표 수집** (Metrics)
✅ **분산 추적/성능 분석** (Tracing)

이 세 가지 요소를 체계적으로 구성하는 것이 매우 중요하다.

Spring Boot에서는 **Actuator** 를 중심으로 기본 지원 기능이 잘 마련되어 있고, 이를 기반으로 **Prometheus + Grafana**, **Zipkin/Jaeger** 등과 쉽게 통합할 수 있다 🚀.

1️⃣ 헬스체크 (Health Check) 🩺

목적

- 서비스의 **가용성 확인** (UP/DOWN 상태)
- LB(로드 밸런서), Kubernetes 등에서 **자동으로 상태 확인 후 트래픽 조절**
- 주요 외부 시스템(DB, MQ 등)의 Health 상태도 확인 가능

기본 Endpoint

```
GET /actuator/health
```

응답 예시:

```
{
  "status": "UP",
  "components": {
    "db": {
      "status": "UP",
      "details": {
        "database": "PostgreSQL",
        "validationQuery": "isValid()"
      }
    },
    "diskSpace": {
      "status": "UP",
      "details": {
        "total": 512000000000,
        "free": 128000000000,
        "threshold": 10485760
      }
    }
  }
}
```

커스텀 Health Indicator

```
@Component
public class CustomHealthIndicator implements HealthIndicator {
    @Override
    public Health health() {
        // Custom logic here
        return Health.up().withDetail("custom", "OK").build();
    }
}
```

외부에서 활용

- AWS ALB, Nginx → **Health Check URL** 로 `/actuator/health` 사용 가능
- Kubernetes → Readiness/Liveness Probe로 활용

```
readinessProbe:
  httpGet:
    path: /actuator/health
    port: 8080
  initialDelaySeconds: 10
  periodSeconds: 5
```

2️⃣ 메트릭 (Metrics) 📊

목적

- JVM, 시스템 자원, 애플리케이션 성능 지표 수집
- DB 커넥션 풀 상태, GC 동작 상태, HTTP 요청 처리 시간, 에러 비율 등 모니터링
- Prometheus와 연계하여 **시각화 대시보드 구성 가능** 🚀

기본 Endpoint

```
GET /actuator/metrics
```

전체 항목 목록 확인 가능:

```
{
  "names": [
    "jvm.memory.used",
    "jvm.memory.max",
    "http.server.requests",
    "system.cpu.usage",
    "process.cpu.usage",
    "logback.events"
  ]
}
```

특정 메트릭 조회

```
GET /actuator/metrics/http.server.requests
```

주요 메트릭 항목 예시

항목	설명
jvm.memory.used	JVM Heap/Non-Heap Memory 사용량
jvm.gc.pause	GC(Pause) 시간

항목	설명
process.cpu.usage	프로세스 CPU 사용률
system.cpu.usage	전체 시스템 CPU 사용률
http.server.requests	HTTP 요청 처리 관련 메트릭
datasource.connections.active	DB Connection Pool 사용량
logback.events	로그 이벤트 발생 수

Prometheus 연계 (Metrics Export)

```
management:
  endpoints:
    web:
      exposure:
        include: prometheus
```

→ `/actuator/prometheus` endpoint 제공 → Prometheus scrape → Grafana 시각화 가능.

3️⃣ 트레이싱 (Tracing) 🧭

목적

- **분산 시스템 환경**에서 개별 요청 흐름을 추적
- 하나의 요청이 여러 서비스에 걸쳐 처리될 때 **전체 경로(Trace)를 시각화**
- Performance Bottleneck 분석 및 문제 진단에 유용

기본 원리

```
HTTP 요청 → Trace ID 생성
    ↓
Trace ID가 서비스 간 호출 시 Propagate 됨
    ↓
각 구간의 Span 기록 (시작/종료 시각, 메타데이터)
    ↓
Collector(Zipkin, Jaeger 등)로 전송 → Trace 시각화
```

구현 예시 (Spring Cloud Sleuth + Zipkin 사용)

의존성 추가

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

application.yml 예시

```yaml
spring:
  zipkin:
    base-url: http://zipkin-server:9411
    sender:
      type: web
  sleuth:
    sampler:
      probability: 1.0  # 100% 샘플링 (운영에서는 보통 낮춤)
```

결과

- 요청마다 자동으로 Trace ID, Span ID 부여
- 로그 출력 시 자동 포함:

```
[traceId=4bf92f3577b34da6a3ce929d0e0e4736, spanId=00f067aa0ba902b7]
```

- Zipkin UI에서 전체 Trace 흐름 확인 가능:

```
Client → Gateway → Service A → Service B → DB
```

주요 장점

✅ 복잡한 **서비스 간 호출 흐름**을 직관적으로 분석 가능
✅ 성능 Bottleneck 지점 파악 가능
✅ 장애 원인 분석(전파 경로 확인)에 매우 유용

4️⃣ 통합 구성 Best Practice 🎁

영역	도구
Health Check	Spring Boot Actuator `/actuator/health`
Metrics	Actuator `/actuator/metrics` + Prometheus + Grafana
Tracing	Spring Cloud Sleuth + Zipkin (or Jaeger)

5️⃣ 결론 🗂️

✅ **헬스체크**는 LB, Kubernetes 등과 연계하여 **서비스 가용성 확보**에 활용된다.
✅ **메트릭**은 운영 상태 모니터링 및 성능 최적화를 위해 필수적이다 → Prometheus + Grafana 연계 시 매우 효과적.
✅ **트레이싱**은 복잡한 MSA 환경에서 **요청 흐름 분석, 성능 Bottleneck 파악, 장애 분석**에 강력한 도구다.

Observability 3대 요소(헬스체크 + 메트릭 + 트레이싱)를 유기적으로 구성하면 운영 안정성을 크게 향상시킬 수 있다 🚀.

Prometheus + Grafana 연동

Prometheus 는 시계열(Time Series) 데이터 수집/저장 시스템이며,
Grafana 는 데이터 시각화 도구이다.
둘을 연계하면 → **Spring Boot 애플리케이션의 Metrics를 실시간 시각화 및 모니터링**할 수 있다 🚀.

Spring Boot + Actuator + Micrometer + Prometheus + Grafana 조합은 현재 **Observability 스택의 표준**으로 많이 사용된다.

1️⃣ 전체 구성 흐름

```
Spring Boot (Actuator + Micrometer Prometheus Endpoint)
    ↓
/actuator/prometheus Endpoint 노출
    ↓
Prometheus → Scrape 주기적으로 Metrics 수집
    ↓
Prometheus Time Series DB 저장
    ↓
Grafana → Prometheus DataSource 연동 → Dashboard 구성
```

2️⃣ 기본 구성 준비물

✅ Spring Boot Actuator
✅ Micrometer (Spring Boot 기본 내장됨)
✅ Prometheus 설치
✅ Grafana 설치

3️⃣ Spring Boot 설정 🛠️

의존성 추가 (Gradle 예시)

```
implementation 'org.springframework.boot:spring-boot-starter-actuator'
implementation 'io.micrometer:micrometer-registry-prometheus'
```

application.yml 설정

```
management:
  endpoints:
    web:
      exposure:
        include: health, info, prometheus
  metrics:
    tags:
      application: my-spring-boot-app
```

👉 `prometheus` endpoint 를 노출해야 Prometheus가 scrape 가능.

확인

```
curl http://localhost:8080/actuator/prometheus
```

출력 예시:

```
# HELP jvm_memory_used_bytes Used bytes of a given JVM memory area.
# TYPE jvm_memory_used_bytes gauge
jvm_memory_used_bytes{area="heap",id="PS Eden Space",} 1.2345678E7
...
```

👉 Prometheus가 이 endpoint 를 **주기적으로 수집**.

4️⃣ Prometheus 구성 🗂️

설치 (Docker 예시)

```
docker run -d -p 9090:9090 --name prometheus \
  -v /path/to/prometheus.yml:/etc/prometheus/prometheus.yml \
  prom/prometheus
```

prometheus.yml 예시

```
global:
  scrape_interval: 15s  # 15초마다 수집

scrape_configs:
  - job_name: 'spring-boot-app'
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: ['host.docker.internal:8080']  # 또는 서버 IP:PORT
```

👉 `targets` 에 Spring Boot 애플리케이션의 `/actuator/prometheus` endpoint 를 설정.

Prometheus 웹 UI

```
http://localhost:9090
```

여기서 **Targets** → Spring Boot app의 endpoint 상태 확인 가능
Graph 탭 → 메트릭 쿼리 테스트 가능.

5️⃣ Grafana 구성 🎨

설치 (Docker 예시)

```
docker run -d -p 3000:3000 --name grafana grafana/grafana
```

초기 접속

```
http://localhost:3000
```

- 기본 ID/PW → admin / admin (초기 변경 요구됨)

DataSource 추가

1️⃣ Configuration → Data Sources → Add data source
2️⃣ Prometheus 선택
3️⃣ URL → `http://host.docker.internal:9090` (Prometheus 서버 URL 입력)
4️⃣ Save & Test → 연결 확인

Dashboard 구성

- Grafana는 **Prometheus DataSource 기반 Dashboard 템플릿** 제공
- Grafana.com 에서 **Spring Boot / JVM / Micrometer / Prometheus** 관련 Dashboard ID 검색 가능

예시 Dashboard:

- JVM Memory Usage
- HTTP Requests Per Second
- DB Connection Pool 상태
- CPU Usage
- GC Pause Time

Dashboard Import

1️⃣ + Create → Import → Dashboard ID 입력 (예: 4701, 6756 등)
2️⃣ Prometheus DataSource 선택 → Import 완료 🚀

6️⃣ 정리 구성 흐름 🗺️

```
Spring Boot → /actuator/prometheus → Prometheus scrape → Prometheus DB → Grafana Dashboard 시각화
```

7️⃣ 결론

✅ Spring Boot는 **Micrometer + Prometheus Exporter** 를 통해 Metrics 수집에 최적화되어 있음.
✅ Prometheus + Grafana 를 연계하면 운영 환경에서 **시계열 모니터링 체계 구축** 가능.
✅ JVM, HTTP 요청 처리 시간, DB 상태, GC 상태 등 다양한 지표를 실시간으로 확인 가능.
✅ 구성도 단순 → **DevOps / SRE 팀에서 표준적인 Observability 스택으로 사용**.

8️⃣ 추가 Best Practice 🎁

영역	전략
Scrap interval	Critical 서비스는 5~15초 추천
Label 구성	metrics.tags.application 으로 App Tag 구성 권장
Custom Metrics	`@Timed`, `@Counted` 등으로 비즈니스 레이어 Metrics 추가 가능
Grafana Dashboard	Import 후 → **자체 커스터마이징** 적극 권장
Alert 구성	Grafana + Prometheus Alertmanager 사용 → 실시간 알람 가능

ELK Stack(Logstash, Kibana)

LK Stack 은 로그/이벤트 데이터 수집 → 저장 → 분석 → 시각화를 위한 **오픈소스 Observability 플랫폼**이다.

구성 요소	역할
Elasticsearch	로그 데이터 저장 및 검색 (Distributed Search Engine)
Logstash	로그 데이터 수집 → 변환 → Elasticsearch로 전송 (Pipeline 구성)
Kibana	시각화 및 분석 UI 제공 (Dashboard 구성)

Spring Boot 애플리케이션 로그를 ELK Stack으로 연동하면:

✅ 운영 환경 로그 수집 표준화
✅ 실시간 검색 및 분석 가능
✅ 장애 발생 시 **로그 기반 원인 분석(Forensics)** 가능 🚀.

1️⃣ 전체 구성 흐름

```
Spring Boot Application (Logback → File / Syslog / TCP)
    ↓
Logstash → 입력(Input) → 변환(Filter) → 출력(Output)
    ↓
Elasticsearch → 로그 저장 (Time Series Index 기반)
    ↓
Kibana → Dashboard 구성 → 시각화 + 검색 + Alert 구성
```

2️⃣ 구성 준비

✅ Elasticsearch 설치
✅ Logstash 설치
✅ Kibana 설치
✅ Spring Boot Logging 설정 구성

👉 보통 **Docker Compose 기반으로 전체 Stack 구성** 많이 사용.

3️⃣ Docker Compose 예시

```yaml
version: '3.7'
services:

  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:8.9.0
    environment:
      - discovery.type=single-node
      - xpack.security.enabled=false
    ports:
      - 9200:9200

  logstash:
    image: docker.elastic.co/logstash/logstash:8.9.0
    ports:
      - 5044:5044   # Beats input
      - 5000:5000   # TCP input
    volumes:
      - ./logstash/pipeline:/usr/share/logstash/pipeline

  kibana:
    image: docker.elastic.co/kibana/kibana:8.9.0
    ports:
      - 5601:5601
    environment:
      - ELASTICSEARCH_HOSTS=http://elasticsearch:9200
```

4️⃣ Logstash Pipeline 구성

Pipeline 구성 예시

```
./logstash/pipeline/logstash.conf
```

```
input {
  tcp {
    port => 5000
    codec => json_lines
  }
}
```

```
filter {
  date {
    match => ["timestamp", "ISO8601"]
  }
}

output {
  elasticsearch {
    hosts => ["http://elasticsearch:9200"]
    index => "spring-boot-logs-%{+YYYY.MM.dd}"
  }
}
```

👉 Logstash 가 **TCP 5000 포트** 에서 JSON 로그 수신 → 필터 적용 → Elasticsearch 저장.

5️⃣ Spring Boot Logging 구성

1️⃣ Logback → File Appender → Logstash FileBeat 방식

(실무에서 많이 사용 → 안정적)

2️⃣ Logback → Logstash TCP Appender

→ 직접 Logstash로 TCP 전송 가능 → 실시간 처리 가능 🚀.

build.gradle 추가

```
implementation 'net.logstash.logback:logstash-logback-encoder:7.2'
```

logback-spring.xml 구성 예시

```
<configuration>
    <appender name="LOGSTASH"
class="net.logstash.logback.appender.LogstashTcpSocketAppender">
        <destination>localhost:5000</destination>
        <encoder class="net.logstash.logback.encoder.LogstashEncoder" />
    </appender>

    <root level="INFO">
        <appender-ref ref="LOGSTASH" />
    </root>
</configuration>
```

👉 이렇게 구성하면 모든 로그가 TCP(5000) 으로 Logstash로 실시간 전송됨.

6️⃣ Kibana 구성

```
http://localhost:5601
```

첫 구성 시

1️⃣ Management → Index Patterns → `spring-boot-logs-*` 생성
2️⃣ Discover 탭 → 실시간 로그 검색 가능
3️⃣ Dashboard → 다양한 Widget 구성 가능
4️⃣ Alerting 기능 활용 → 특정 Error 발생 시 알람 가능

7️⃣ 주요 활용 예시

- 실시간 **Error 추적** 및 통계 분석
- **Slow Query, OutOfMemory 등 JVM 로그 분석**
- 사용자 요청 추적 → User Journey 분석
- Application KPI 모니터링 → 성공/실패 비율 모니터링
- 운영팀 장애 대응 속도 향상 🚀

8️⃣ Best Practice 🎁

영역	전략
Index Management	Index Lifecycle 설정 → 오래된 Index 자동 삭제
Log Format	JSON 기반 통일 → Kibana 필터링 용이
Security	Production 환경에서는 반드시 TLS 적용 + 인증 적용
Resource Tuning	Logstash JVM Heap, Elasticsearch JVM Heap 최적화
Alert 구성	Kibana Alert 구성 → Slack / Email 알림 연계

9️⃣ 결론 🗂️

✅ **ELK Stack** 은 Spring Boot 애플리케이션의 운영 로그를 수집/저장/분석하는 데 매우 강력한 도구다.
✅ **Logstash TCP Appender 구성** → 실시간 로그 수집 가능 → 운영 장애 분석에 유리.
✅ **Kibana 대시보드**를 통해 실시간 모니터링 및 Alert 구성 가능.
✅ 시스템 가시성(Observability) 확보에 필수적인 구성으로 매우 추천.

Zipkin, Jaeger 기반 분산 트레이싱

분산 트레이싱(Distributed Tracing) 은 MSA 환경에서 필수적인 관찰(Observability) 요소이다.
하나의 사용자 요청이 여러 서비스에 걸쳐 흐를 때:

✅ 전체 호출 경로(Trace)를 시각화
✅ 각 서비스 간 응답 시간 파악
✅ 병목 구간 및 장애 분석 가능 🚀

Zipkin 과 **Jaeger** 는 대표적인 오픈소스 분산 트레이싱 시스템이다.

항목	Zipkin	Jaeger
개발 주체	Twitter (초기), OpenZipkin	CNCF (Cloud Native Computing Foundation)
설치 용이성	매우 간단	Kubernetes 친화적, 엔터프라이즈급 기능 우수
UI 기능	기본적	고급 (Trace Graph, Service Dependency 시각화)
사용 사례	경량 MSA, 빠른 구성	대규모 MSA, 고가용성 트레이싱 구성

1️⃣ 전체 구성 흐름

```
Client Request → Gateway → Service A → Service B → DB
    ↓
각 서비스에서 TraceID + SpanID 자동 전파
    ↓
Zipkin or Jaeger Collector → Trace 데이터 저장
    ↓
UI (Zipkin or Jaeger Web UI) → Trace 시각화
```

👉 Spring Boot에서는 **Spring Cloud Sleuth** 를 통해 **자동으로 Trace 정보 관리 및 전파** 가능.

2️⃣ 기본 구성

의존성 추가 (Gradle)

Zipkin 사용 시

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

Jaeger 사용 시

```
implementation 'io.opentelemetry:opentelemetry-api:1.17.0'
implementation 'io.opentelemetry:opentelemetry-sdk:1.17.0'
implementation 'io.opentelemetry:opentelemetry-exporter-jaeger:1.17.0'
```

👉 최신 트렌드는 **OpenTelemetry 기반으로 Jaeger 연동**하는 경우가 많음.

3️⃣ Spring Boot + Sleuth + Zipkin 구성

application.yml 예시

```yaml
spring:
  zipkin:
    base-url: http://zipkin-server:9411
    sender:
      type: web
  sleuth:
    sampler:
      probability: 1.0  # 100% 샘플링 (운영에서는 0.1~0.3 추천)
```

👉 `/actuator/trace` 는 더 이상 기본 지원되지 않음 → Zipkin UI로 확인.

Trace 자동 전파 예시

- HTTP 요청 시 Header 에 자동으로 다음 정보 포함:

```
X-B3-TraceId: 4bf92f3577b34da6a3ce929d0e0e4736
X-B3-SpanId: 00f067aa0ba902b7
X-B3-Sampled: 1
```

👉 Feign Client, RestTemplate, WebClient 사용 시 자동으로 Header 전파됨.

4️⃣ Zipkin 구성

Docker Compose 예시

```yaml
services:
  zipkin:
    image: openzipkin/zipkin
    ports:
      - 9411:9411
```

👉 Web UI: `http://localhost:9411`
→ Trace → Service 선택 → 호출 경로 시각화 가능.

5️⃣ Spring Boot + OpenTelemetry + Jaeger 구성

최신 표준: OpenTelemetry → Jaeger Exporter 구성

의존성 추가

```
implementation 'io.opentelemetry.instrumentation:opentelemetry-spring-boot-starter:1.17.0-alpha'
implementation 'io.opentelemetry:opentelemetry-exporter-jaeger:1.17.0'
```

application.yml 예시

```
otel:
  traces:
    exporter: jaeger
  exporter:
    jaeger:
      endpoint: http://jaeger-collector:14250
```

👉 최신 OpenTelemetry 기반 → Zipkin/Jaeger 둘 다 Exporter 구성 가능 (플러그블 구조).

6️⃣ Jaeger 구성

Docker Compose 예시

```
services:
  jaeger:
    image: jaegertracing/all-in-one:1.41
    ports:
      - "16686:16686"  # UI
      - "14250:14250"  # OTLP gRPC
      - "6831:6831/udp"  # Jaeger Agent (UDP)
```

👉 Web UI: `http://localhost:16686`
→ Traces → Service 선택 → 호출 경로, Span 분석 가능.

7️⃣ Trace 정보 활용 예시

```
Client → Gateway → Order Service → Payment Service → Notification Service
```

Zipkin or Jaeger 에서:

✅ 전체 호출 시간 분석
✅ 각 구간 처리 시간 확인 (Span 단위로 시각화)
✅ 오류 발생 Span 확인
✅ TraceID 기반으로 **로그/Trace 연계 조회 가능** → 운영 장애 분석 매우 유리.

8️⃣ Best Practice

영역	전략
Sampling Probability	운영에서는 10~30% 수준으로 설정 (1.0은 성능 부하 주의)
Trace Header 전파	Feign / RestTemplate / WebClient 전부 지원
Trace Logging	MDC 기반으로 로그에 TraceID 추가 → `[%X{traceId}]`
Storage	Zipkin / Jaeger 모두 ES 연동 가능 → 장기 보관 시 Elasticsearch 활용

영역	전략
Trace 분석	장애 대응, 성능 Bottleneck 분석, 서비스 Dependency 시각화 활용

9️⃣ 결론

✅ Zipkin / Jaeger는 Spring Boot MSA 환경에서 **분산 트레이싱 표준 도구**이다.
✅ **Spring Cloud Sleuth** 기반으로 간편하게 Zipkin 연동 가능.
✅ 최신 트렌드는 **OpenTelemetry + Jaeger Exporter** 구조를 많이 사용한다.
✅ 분산 트레이싱 구축 시 **서비스 호출 흐름 이해, 장애 원인 분석, 성능 최적화**에 매우 강력한 효과.