

# 19. 운영 및 배포 환경

## Spring Boot Actuator

Actuator는 운영 환경에서 애플리케이션의 **헬스 상태, 메트릭, 환경 설정, 로그, 캐시, 쓰레드** 등 내부 정보를 외부에서 모니터링하고 제어할 수 있도록 해주는 강력한 기능이다.

### 1. Actuator란?

**Spring Boot Actuator**는 애플리케이션의 **운영 상태, 헬스 체크, 통계 메트릭, 환경 정보** 등을 외부에 노출하는 모듈이다.

주요 목적:

- 헬스 체크 (`/actuator/health`)
- 시스템 모니터링 (`/actuator/metrics`)
- 환경 속성 확인 (`/actuator/env`)
- HTTP 트래픽 추적 (`/actuator/httptrace`)
- 캐시 상태, Bean 목록 등 다양한 정보 확인

### 2. 의존성 추가

```
1 dependencies {
2     implementation 'org.springframework.boot:spring-boot-starter-actuator'
3 }
```

### 3. 기본 엔드포인트

엔드포인트	설명
<code>/actuator/health</code>	헬스 체크 (DB, Redis, MQ 등 연결 여부 포함 가능)
<code>/actuator/info</code>	빌드 정보, 애플리케이션 정보 노출
<code>/actuator/metrics</code>	CPU, 메모리, GC, 요청 수, 응답 시간 등의 메트릭
<code>/actuator/env</code>	<code>application.yml</code> 의 환경 변수 노출
<code>/actuator/beans</code>	등록된 모든 Bean 목록
<code>/actuator/mappings</code>	컨트롤러 매핑 정보
<code>/actuator/loggers</code>	로그 레벨 동적 변경
<code>/actuator/heapdump</code>	JVM Heap Dump 파일 다운로드
<code>/actuator/threaddump</code>	쓰레드 덤프 출력

엔드포인트	설명
<code>/actuator/caches</code>	Spring Cache 상태 확인 및 삭제

## 4. 설정 방법 (application.yml)

```
1 management:
2   endpoints:
3     web:
4       exposure:
5         include: "*"
6   endpoint:
7     health:
8       show-details: always
9   info:
10    env:
11      enabled: true
```

- `exposure.include`: 노출할 actuator 엔드포인트 목록 (\* = 전체)
- `show-details`: 헬스 정보 상세 출력 여부
- `info.env.enabled`: `info` 엔드포인트에 환경 정보 포함 여부

## 5. /actuator/info 설정

`info` 엔드포인트에 사용자 정의 정보 추가 가능:

```
1 info:
2   app:
3     name: demo-api
4     version: 1.2.3
5     description: Spring Boot REST API
```

```
1 GET /actuator/info
2 → {
3   "app": {
4     "name": "demo-api",
5     "version": "1.2.3",
6     "description": "Spring Boot REST API"
7   }
8 }
```

## 6. 헬스 체크 상세화

Spring Boot는 **DataSource, Redis, MongoDB, Kafka** 등이 자동으로 health 구성에 포함됨.

```
1 management:
2   endpoint:
3     health:
4       show-details: always
```

```
1 GET /actuator/health
2 → {
3   "status": "UP",
4   "components": {
5     "db": {
6       "status": "UP",
7       "details": {
8         "database": "MySQL",
9         "result": 1
10      }
11    },
12    "diskSpace": {
13      "status": "UP",
14      "details": { ... }
15    }
16  }
17 }
```

## 7. Micrometer 기반 메트릭 통합

Spring Boot Actuator는 내부적으로 **Micrometer**를 사용함.

→ Prometheus, Grafana 등 외부 시스템과 연결 가능

```
1 GET /actuator/metrics
2 GET /actuator/metrics/jvm.memory.used
3 GET /actuator/metrics/http.server.requests
```

## 8. Prometheus 연동

의존성 추가

```
1 implementation 'io.micrometer:micrometer-registry-prometheus'
```

## 설정

```
1 management:
2   metrics:
3     export:
4       prometheus:
5         enabled: true
6   endpoints:
7     web:
8       exposure:
9         include: prometheus
```

```
1 GET /actuator/prometheus
2 → Prometheus가 스크래핑 가능
```

## 9. 로그 레벨 실시간 변경

```
1 GET /actuator/loggers/com.example.demo
2 PUT /actuator/loggers/com.example.demo
3 {
4   "configuredLevel": "DEBUG"
5 }
```

→ 실시간으로 특정 패키지의 로그 레벨을 변경할 수 있음 (운영 중에도 디버깅 가능)

## 10. 캐시 상태 확인

```
1 GET /actuator/caches
2 → 등록된 캐시 목록 출력
3
4 POST /actuator/caches/myCache
5 → 특정 캐시 항목 삭제
```

## 11. 보안: actuator 접근 제한

기본적으로 `/actuator` 는 보안이 필요함. Spring Security 사용 시 다음처럼 설정 가능:

```
1 management:
2   endpoints:
3     web:
4       exposure:
5         include: health, info
6   security:
7     enabled: true
```

```

1 @Override
2 protected void configure(HttpSecurity http) throws Exception {
3     http
4         .authorizeRequests()
5             .requestMatchers("/actuator/**").hasRole("ADMIN")
6             .anyRequest().permitAll();
7 }

```

## 12. actuator + Kubernetes, 클라우드

- `/actuator/health/liveness`, `/actuator/health/readiness` 는 K8s liveness probe와 readiness probe로 사용 가능
- `/actuator/prometheus` 로 Grafana + Prometheus 연동
- Spring Cloud + Actuator 조합으로 Eureka, Gateway, Config Server 상태 점검도 가능

### ✅ 마무리 요약

항목	설명
핵심 목적	헬스 체크, 메트릭 수집, 운영 정보 노출
주요 엔드포인트	<code>/actuator/health</code> , <code>/metrics</code> , <code>/info</code> , <code>/prometheus</code> 등
설정	<code>management.endpoints.web.exposure.include=*</code>
확장	Prometheus, Grafana, Slack 알림 등과 통합 가능
보안	필요 시 Security 연동으로 인증 보호 가능

## Prometheus, Grafana 모니터링 연동

### 1. 구성도

```

1 [Spring Boot App]
2 |
3 | <-- /actuator/prometheus
4 |
5 [Prometheus] <-- scrape
6 |
7 |
8 [Grafana] <-- 시각화 대시보드

```

## 2. 사전 요구 사항

- Spring Boot 2.5+ 또는 3.x
- `spring-boot-starter-actuator`
- `micrometer-registry-prometheus`
- Prometheus 실행 환경
- Grafana 실행 환경

로컬에서 테스트할 땐 **Docker로 Prometheus + Grafana** 띄우는 게 편해.

---

## 3. Spring Boot 설정

### ◆ 의존성 추가 (Gradle 기준)

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

---

### ◆ application.yml 설정

```
1 management:
2   endpoints:
3     web:
4       exposure:
5         include: health, metrics, prometheus
6   endpoint:
7     prometheus:
8       enabled: true
9   metrics:
10    export:
11      prometheus:
12        enabled: true
```

---

### ◆ 확인

애플리케이션 실행 후 브라우저에서 아래 주소 확인:

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

→ Prometheus가 인식할 수 있는 **텍스트 포맷의 메트릭 정보**가 출력되어야 해.

---

## 4. Prometheus 설정

### ◆ Docker로 Prometheus 실행

```
1 docker run -d \  
2   -p 9090:9090 \  
3   -v $(pwd)/prometheus.yml:/etc/prometheus/prometheus.yml \  
4   prom/prometheus
```

### ◆ prometheus.yml 내용

```
1 global:  
2   scrape_interval: 10s  
3  
4   scrape_configs:  
5     - job_name: 'spring-boot-app'  
6       metrics_path: '/actuator/prometheus'  
7       static_configs:  
8         - targets: ['host.docker.internal:8080']
```

`host.docker.internal`은 Docker에서 **호스트의 8080 포트에 접근**하는 공식적인 방법 (Windows/Mac 기준)

리눅스라면 `localhost` 대신 **실제 IP**를 써야 할 수도 있음.

## 5. Grafana 설정


### ◆ Docker로 Grafana 실행

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

### ◆ 로그인 정보

- URL: <http://localhost:3000>
- 기본 ID: `admin` / PW: `admin`

### ◆ Prometheus 데이터소스 연결


1. 좌측 메뉴 →  **Configuration > Data Sources**
2. Add data source → **Prometheus**
3. URL: `http://host.docker.internal:9090`  
(Linux는 `localhost` 또는 IP로)

## 6. 대시보드 구성

### ◆ 방법 1: 직접 생성

- Dashboards → New → Add Panel
- Metric 선택: `http_server_requests_seconds_count`, `jvm_memory_used_bytes` 등

### ◆ 방법 2: JSON Import

1. Grafana 왼쪽 메뉴 →  Dashboards → Import
2. Dashboard ID 입력: `4701` (Spring Boot Metrics)
3. 데이터소스 선택 → Import

ID 4701은 Micrometer + Prometheus 공식 템플릿이야

## 7. 대표 메트릭 항목

메트릭 이름	설명
<code>jvm.memory.used</code>	JVM 메모리 사용량
<code>jvm.gc.pause</code>	GC 시간
<code>process.cpu.usage</code>	CPU 사용률
<code>http.server.requests</code>	요청 수, 응답 시간
<code>logback.events</code>	로그 레벨별 이벤트 수
<code>tomcat.threads.active</code>	활성 쓰레드 수

## 8. 커스텀 메트릭 추가 (선택)

```
1 | @Autowired
2 | MeterRegistry registry;
3 |
4 | @PostConstruct
5 | public void init() {
6 |     registry.counter("custom.orders.processed").increment();
7 | }
```

→ `/actuator/prometheus` 에서 `custom_orders_processed` 확인 가능



## 9. 실전 팁

항목	팁
보안	<code>/actuator/prometheus</code> 는 인증 필요 시 Security 설정으로 제한
다중 인스턴스	Prometheus의 <code>job</code> , <code>instance</code> 레이블로 구분
응답지연 추적	<code>http_server_requests_seconds_count</code> , <code>sum(rate(...))</code> 으로 분석
알림	Grafana Alert 또는 Prometheus AlertManager 연동

### ✅ 마무리 요약

구성 요소	역할
Spring Boot	메트릭 제공 ( <code>/actuator/prometheus</code> )
Micrometer	측정 포맷 및 수집
Prometheus	수집기 (scraper)
Grafana	시각화 도구

## 환경 분리 설정

- `application-dev.yml`, `application-prod.yml`

### 📁 파일 구조

```
1 | src/
2 |   └─ main/
3 |     └─ resources/
4 |         └─ application.yml           # 공통 설정
5 |         └─ application-dev.yml      # 개발 환경 전용
6 |         └─ application-prod.yml     # 운영 환경 전용
```

- `application.yml`: 공통 설정 (모든 환경에 적용)
- `application-dev.yml`: 개발 환경 전용 설정 (로컬 개발 시 사용)
- `application-prod.yml`: 운영 환경 전용 설정 (실제 배포 시 사용)

## application.yml (공통 설정 예시)

```
1 spring:
2   application:
3     name: my-api
4   profiles:
5     active: dev # 기본은 dev로 시작 (운영 시 이걸 제거하고 prod로 실행)
6
7   server:
8     port: 8080
9
10  logging:
11    file:
12      name: logs/app.log
```

## application-dev.yml (개발 환경 설정 예시 + 상세 설명)

```
1 spring:
2   datasource:
3     url: jdbc:mysql://localhost:3306/devdb
4     username: devuser
5     password: devpass
6     driver-class-name: com.mysql.cj.jdbc.Driver
7   jpa:
8     hibernate:
9       ddl-auto: update # 개발용으로 테이블 자동 생성/업데이트
10      show-sql: true    # SQL 로그 출력
11      properties:
12        hibernate:
13          format_sql: true
14
15  logging:
16    level:
17      root: DEBUG
18      org.hibernate.SQL: DEBUG
19      org.hibernate.type.descriptor.sql.BasicBinder: TRACE
20
21  springdoc:
22    swagger-ui:
23      enabled: true # Swagger UI 개발 중에만 활성화
24
25  custom:
26    cache-ttl: 30s # 캐시 TTL을 짧게 (빠른 테스트 목적)
27
28  management:
29    endpoints:
30      web:
31        exposure:
32          include: "*"

```

## ✓ dev 환경 특징

항목	설명
<code>ddl-auto: update</code>	DB 스키마 자동 변경 허용
<code>show-sql: true</code>	실행되는 쿼리 로그 출력
로그레벨	DEBUG 및 SQL 추적용 TRACE
Swagger UI	항상 활성화
캐시 TTL	짧게 설정 (30초 등)
Actuator	전체 endpoint 노출 ( <code>include: *</code> )

## 🔒 `application-prod.yml` (운영 환경 설정 예시 + 상세 설명)

```
1  spring:
2    datasource:
3      url: jdbc:mysql://prod-db.internal:3306/proddb
4      username: producer
5      password: ${PROD_DB_PASSWORD} # 운영 환경에선 환경변수로 관리
6      driver-class-name: com.mysql.cj.jdbc.Driver
7    jpa:
8      hibernate:
9        ddl-auto: none # 운영 DB는 자동 변경 ✗
10     show-sql: false
11     open-in-view: false # 성능과 안정성 고려
12    jackson:
13      serialization:
14        indent_output: false # 성능 고려 (pretty print 제거)
15
16    logging:
17      level:
18        root: WARN
19        com.example.app: INFO
20
21    custom:
22      cache-ttl: 1800s # 캐시 TTL을 길게 설정 (30분)
23
24    springdoc:
25      swagger-ui:
26        enabled: false # 운영환경에서는 Swagger UI 비활성화
27
28    management:
29      endpoints:
30        web:
31          exposure:
32            include: health, info, prometheus
33      endpoint:
34        health:
```

## ✓ prod 환경 특징

항목	설명
DB 계정	운영용 계정 사용, <code>환경변수</code> 로 비밀번호 외부화
<code>ddl-auto: none</code>	DB 구조는 수동 관리 (Flyway 등)
SQL 출력	비활성화 ( <code>show-sql: false</code> )
Swagger	비활성화 (보안 위험 방지)
로그레벨	WARN 기본, 패키지별로 INFO 설정
Cache TTL	더 길게 설정
Actuator	제한된 엔드포인트만 노출 ( <code>health</code> , <code>info</code> , <code>prometheus</code> )

## ⚙️ 프로파일 실행 방식

방식	명령어 예시
로컬 실행 시 dev	기본으로 dev가 활성화됨
운영 서버에서 prod 사용	<code>--spring.profiles.active=prod</code> 또는 환경변수로 설정

```

1 # 실행 예
2 java -jar app.jar --spring.profiles.active=prod

```

또는:

```

1 export SPRING_PROFILES_ACTIVE=prod
2 ./gradlew bootRun

```

## ✓ 보안 관련 주의사항

항목	설명
<code>application-prod.yml</code> 은 Git에 커밋 ❌	<code>.gitignore</code> 에 추가 또는 CI/CD에서 별도 주입
DB 암호, API Key 등은 환경 변수 또는 vault 사용	
Swagger, actuator 등은 운영에서는 제한 필요	

## ✳ 커스텀 설정 예시 가져오기

```
1 @Value("${custom.cache-ttl}")
2 private Duration cacheTtl;
```

또는 `@ConfigurationProperties(prefix = "custom")` 로 전체 바인딩도 가능해.

## ✅ 요약 비교표

항목	application-dev.yml	application-prod.yml
DB 주소	로컬 개발 DB	실제 운영 DB
Hibernate	ddl-auto: update	ddl-auto: none
로그 레벨	DEBUG, SQL 출력	WARN, SQL 숨김
Swagger	항상 활성화	비활성화
Actuator 노출	전체( *)	제한(health, info)
캐시 TTL	짧음 (30초~1분)	김 (10분~30분)
보안 민감 정보	포함되어도 됨(로컬)	반드시 외부화 (환경변수, vault)

## 로깅 설정: Logback, Log4j2

### 0. 로깅 시스템 비교

항목	Logback	Log4j2
Spring Boot 기본	✅ 예 (기본 포함)	❌ (별도 설정 필요)
비동기 로깅	지원 ( AsyncAppender )	고성능 비동기 로깅 ( AsyncLogger )
XML/JSON/YAML 설정	XML (기본), Groovy도 지원	XML, JSON, YAML 모두 지원
성능	좋음	더 우수함 (비동기 중심 설계)
설정 복잡도	중간	조금 더 복잡
Spring Boot 통합	기본 지원	spring-boot-starter-log4j2 로 명시적 설정 필요

### 1. ✅ Logback 설정 (Spring Boot 기본)

## 📁 기본 설정 파일 위치

1 | src/main/resources/logback-spring.xml

⚠ 반드시 `logback.xml` 이 아닌 `logback-**spring**.xml` 사용해야 **Spring profile 연동**이 가능함

### ◆ logback-spring.xml 예시

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <configuration>
3
4      <springProperty scope="context" name="LOG_PATH" source="logging.file.path"
5      defaultvalue="logs" />
6      <property name="LOG_PATTERN" value="%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level
7      %logger{36} - %msg%n"/>
8
9      <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
10         <file>${LOG_PATH}/app.log</file>
11         <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
12             <fileNamePattern>${LOG_PATH}/app.%d{yyyy-MM-dd}.log</fileNamePattern>
13             <maxHistory>7</maxHistory>
14         </rollingPolicy>
15         <encoder>
16             <pattern>${LOG_PATTERN}</pattern>
17         </encoder>
18     </appender>
19
20     <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
21         <encoder>
22             <pattern>${LOG_PATTERN}</pattern>
23         </encoder>
24     </appender>
25
26     <root level="INFO">
27         <appender-ref ref="CONSOLE" />
28         <appender-ref ref="FILE" />
29     </root>
30
31     <logger name="org.hibernate.SQL" level="DEBUG" />
32     <logger name="com.example.myapp" level="DEBUG" />
33
34 </configuration>
```

## ◆ application.yml에서 함께 설정 가능

```
1 logging:
2   file:
3     path: logs
4   level:
5     root: INFO
6     com.example.myapp: DEBUG
7     org.hibernate.SQL: DEBUG
```

간단한 설정은 `application.yml` 로 처리하고, 세부 설정은 `logback-spring.xml` 로.

## ◆ 로그 프로파일링 (dev/prod 분기)

```
1 <springProfile name="dev">
2   <logger name="org.hibernate.SQL" level="DEBUG" />
3 </springProfile>
4
5 <springProfile name="prod">
6   <logger name="org.hibernate.SQL" level="WARN" />
7 </springProfile>
```

## 2. Log4j2 설정

### 의존성 교체 (Gradle)

```
1 dependencies {
2   implementation 'org.springframework.boot:spring-boot-starter-log4j2'
3
4   // 다음 2개 제거!
5   configurations {
6     all {
7       exclude group: 'org.springframework.boot', module: 'spring-boot-starter-
      logging'
8     }
9   }
10 }
```

### 설정 파일 이름

```
1 log4j2-spring.xml
```

### ◆ log4j2-spring.xml 예시

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Configuration status="WARN">
3   <Appenders>
4     <Console name="Console" target="SYSTEM_OUT">
5       <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %logger{36} -
6       %msg%n"/>
7     </Console>
8     <RollingFile name="File" fileName="logs/app.log"
9       filePattern="logs/app-%d{yyyy-MM-dd}.log.gz">
10      <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %logger{36} -
11      %msg%n"/>
12      <Policies>
13        <TimeBasedTriggeringPolicy />
14      </Policies>
15    </RollingFile>
16  </Appenders>
17  <Loggers>
18    <Root level="info">
19      <AppenderRef ref="Console"/>
20      <AppenderRef ref="File"/>
21    </Root>
22    <Logger name="com.example.myapp" level="debug" additivity="false">
23      <AppenderRef ref="File"/>
24    </Logger>
25  </Loggers>
26 </Configuration>
```

### 3. 공통 설정 항목 요약

설정 항목	설명
로그 파일 경로	logging.file.path 또는 .xml 직접 지정
로그 포맷	%d{} 날짜, %level, %logger, %msg
로그 레벨	TRACE, DEBUG, INFO, WARN, ERROR
파일 롤링	날짜 또는 크기 기준 회전 (RollingPolicy)
콘솔 출력	ConsoleAppender
비동기 로깅	AsyncAppender (Logback), AsyncLogger (Log4j2)








#### 4. 로그 레벨 동적 변경 (Actuator + Loggers)



```
1 management:
2   endpoints:
3     web:
4       exposure:
5         include: loggers

1 GET  /actuator/loggers/com.example.myapp
2 PUT  /actuator/loggers/com.example.myapp
3 {
4   "configuredLevel": "DEBUG"
5 }
```

#### 5. 언제 Logback vs Log4j2?

기준	Logback	Log4j2
Spring Boot 기본		
고성능 비동기 로깅	△ ( AsyncAppender )	 ( AsyncLogger )
설정 유연성	보통	뛰어남
JSON 로깅, Kafka Appender	 (외부 추가 필요)	 풍부한 공식 지원
간단한 설정	쉬움	약간 복잡함

#### 마무리 요약

항목	Logback	Log4j2
설정 파일	logback-spring.xml	log4j2-spring.xml
기본 포함		 (명시적 교체 필요)
Spring profile 연동	지원 ( <springProfile> )	지원
로그 레벨 제어	application.yml or XML	동일
고성능 비동기	제한적 ( AsyncAppender )	우수 ( AsyncLogger )

# 로그 수집기 연동: ELK, Loki

## 0. 구성 개요

시스템	구성 요소
ELK 스택	Elasticsearch, Logstash, Kibana
Grafana Loki	Loki + Promtail + Grafana

## 1. 🎯 공통 목표: Spring Boot 로그를 중앙 서버로 수집

```
1 [Spring Boot App]
2   ↓ logs (file or stdout)
3 [로그 수집기] ← logback 또는 log4j2
4   ↓
5 [Elasticsearch / Loki] ← 수집기 (Filebeat, Promtail 등)
6   ↓
7 [Kibana / Grafana] ← 시각화
```

## 2. ✅ ELK Stack 연동 (Elasticsearch + Logstash + Kibana)

### 2.1 로그 포맷 설정 (Logback 기반 JSON 로그)

📁 logback-spring.xml:

```
1 <configuration>
2   <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
3     <file>logs/app.log</file>
4     <encoder class="net.logstash.logback.encoder.LogstashEncoder" />
5   </appender>
6
7   <root level="INFO">
8     <appender-ref ref="FILE" />
9   </root>
10 </configuration>
```

### ◆ 의존성 추가

```
1 implementation 'net.logstash.logback:logstash-logback-encoder:7.4'
```

## 2.2 Filebeat 설정 (로그 수집기 → Logstash/ES)

filebeat.yml:

```
1 filebeat.inputs:
2   - type: log
3     paths:
4       - /app/logs/app.log
5     json.keys_under_root: true
6     json.add_error_key: true
7
8 output.elasticsearch:
9   hosts: ["http://localhost:9200"]
10
11 # 또는 Logstash 경유 시:
12 # output.logstash:
13 #   hosts: ["localhost:5044"]
```

Filebeat는 경량 log shipper로, 로그 파일을 실시간으로 읽어 Elasticsearch에 보냄

## 2.3 Kibana에서 시각화

- <http://localhost:5601> 접속
- "Index Pattern" 생성 → filebeat-\*
- Discover에서 JSON 로그를 실시간 검색

## 3. Grafana Loki 연동

### 3.1 로깅 포맷: stdout (혹은 JSON 파일)

```
1 logging:
2   pattern:
3     console: "%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger - %msg%n"
```

### 3.2 Promtail 설정

promtail-config.yaml:

```
1 server:
2   http_listen_port: 9080
3
4 positions:
5   filename: /tmp/positions.yaml
6
7 clients:
8   - url: http://localhost:3100/loki/api/v1/push
9
10 scrape_configs:
```

```
11 - job_name: spring-boot-app
12   static_configs:
13     - targets:
14       - localhost
15     labels:
16       job: springboot
17       __path__: /app/logs/*.log
```

Promtail은 Loki에 로그를 push하는 agent (Filebeat의 역할과 유사)

### 3.3 Grafana 설정

1. Grafana → Configuration → Data Sources
2. Add data source → Loki 선택
3. URL: `http://localhost:3100`

### 3.4 Grafana 쿼리 예시

```
1 | {job="springboot"} |= "ERROR"
```

→ 실시간 로그 스트림 + 필터링

## 4. 비교: ELK vs Loki

항목	ELK	Loki
스토리지	JSON 구조 (Elasticsearch)	텍스트 + 메타데이터 (Log lines + label)
쿼리	Kibana + Lucene	Grafana + LogQL
사용 메모리	높음 (Elasticsearch 무거움)	가벼움
수집기	Filebeat, Logstash	Promtail
시각화 도구	Kibana	Grafana
도입 난이도	높음	낮음
통합 용도	로그 분석, 검색, 필터링	로그 모니터링, 스트리밍
추천 환경	대규모 검색, 법적 감사용	실시간 DevOps/클라우드 환경

## 5. 실전 운영 팁

항목	팁
로그 포맷	JSON 구조가 수집기와 시각화에 유리
태그/Label	<code>application</code> , <code>env</code> , <code>instance</code> 등을 로그에 포함
보안	로그 서버에 인증 프록시 두기 (Nginx + Basic Auth)
문제 대응	로그 파싱 실패 시 <code>@timestamp</code> 유실, 필드명 확인
Spring profile	로그에 <code>[dev]</code> , <code>[prod]</code> 포함하면 구분 쉬움

### ✅ 마무리 요약

구성	ELK	Loki
로그 저장소	Elasticsearch	Loki
시각화	Kibana	Grafana
수집기	Filebeat / Logstash	Promtail
Spring 설정	JSON 로그 + File 출력	텍스트 로그 + stdout or 파일
성능	무거움	가벼움
로그 라인 단위 분석	제한적	매우 강력 (streaming)

## Docker 기반 배포

### 1. 목표

Spring Boot 프로젝트를 다음과 같이 Docker로 배포 가능한 형태로 만들기:

1	[Spring Boot Project]
2	↓ build
3	[.jar 파일 생성]
4	↓ Dockerfile 빌드
5	[Docker 이미지]
6	↓ 실행
7	[Docker 컨테이너에서 앱 실행]

## 2. 선행 조건

- Java 17 또는 21 (Spring Boot 3+ 권장)
- Gradle 또는 Maven 빌드 툴
- Docker 설치됨

## 3. JAR 생성

### Gradle 기준

```
1 | ./gradlew bootJar
```

생성 위치:

```
1 | build/libs/myapp-0.0.1-SNAPSHOT.jar
```

## 4. Dockerfile 작성

 프로젝트 루트에 `Dockerfile` 생성

```
1 | # 베이스 이미지
2 | FROM eclipse-temurin:17-jdk-jammy
3 |
4 | # 작업 디렉토리 설정
5 | WORKDIR /app
6 |
7 | # JAR 복사
8 | COPY build/libs/myapp-0.0.1-SNAPSHOT.jar app.jar
9 |
10 | # 포트 오픈
11 | EXPOSE 8080
12 |
13 | # 실행 명령
14 | ENTRYPOINT ["java", "-jar", "app.jar"]
```

## 5. `.dockerignore` 작성 (필수 아님)

```
1 | /build
2 | .gradle
3 | .DS_Store
4 | *.iml
```

→ Docker 빌드 컨텍스트에서 불필요한 파일 제외

## 6. ✅ 이미지 빌드

```
1 | docker build -t myapp:latest .
```

이미지 확인:

```
1 | docker images
```

## 7. ✅ 컨테이너 실행

```
1 | docker run -d -p 8080:8080 --name myapp myapp:latest
```

→ <http://localhost:8080> 에서 정상 동작 확인


## 8. ✅ 환경 변수/프로파일 설정

```
1 | docker run -d \  
2 |   -p 8080:8080 \  
3 |   -e SPRING_PROFILES_ACTIVE=prod \  
4 |   -e SPRING_DATASOURCE_URL=jdbc:mysql://host.docker.internal:3306/proddb \  
5 |   myapp:latest
```

## 9. ✅ volume/log 설정 (운영 추천)

```
1 | docker run -d \  
2 |   -v /var/log/myapp:/app/logs \  
3 |   -e LOG_PATH=/app/logs \  
4 |   myapp:latest
```

## 10. Docker Compose 예제 (멀티 컨테이너)

 docker-compose.yml 예시:

```
1 | version: '3.8'  
2 | services:  
3 |   myapp:  
4 |     image: myapp:latest  
5 |     ports:  
6 |       - "8080:8080"  
7 |     environment:  
8 |       - SPRING_PROFILES_ACTIVE=prod  
9 |     depends_on:  
10 |       - mysql  
11 | mysql:
```

```

12 image: mysql:8
13 environment:
14     MYSQL_ROOT_PASSWORD: root
15     MYSQL_DATABASE: proddb
16 ports:
17     - "3306:3306"

```

```
1 docker-compose up --build
```

## 11. 멀티 스테이지 빌드 (고급)

```

1 # 1단계: Build
2 FROM gradle:8.3-jdk17 AS builder
3 WORKDIR /workspace
4 COPY . .
5 RUN gradle bootJar
6
7 # 2단계: Run
8 FROM eclipse-temurin:17-jdk
9 WORKDIR /app
10 COPY --from=builder /workspace/build/libs/*.jar app.jar
11 ENTRYPOINT ["java", "-jar", "app.jar"]

```

→ 빌드 툴 없이 가벼운 이미지 생성 가능

## 12. 이미지 최적화 팁

항목	전략
이미지 크기	<code>eclipse-temurin</code> , <code>alpine</code> , <code>distroless</code> 사용
보안	root 사용자 금지, <code>USER 1000</code>
설정 파일 분리	<code>config/</code> → volume mount
JAR 위치	<code>/app/app.jar</code> 등 표준화 권장
healthcheck	Dockerfile 또는 Compose에서 설정 가능

### ✅ 마무리 요약

단계	명령
JAR 생성	<code>./gradlew bootJar</code>
Dockerfile 작성	<code>FROM eclipse-temurin</code> 등

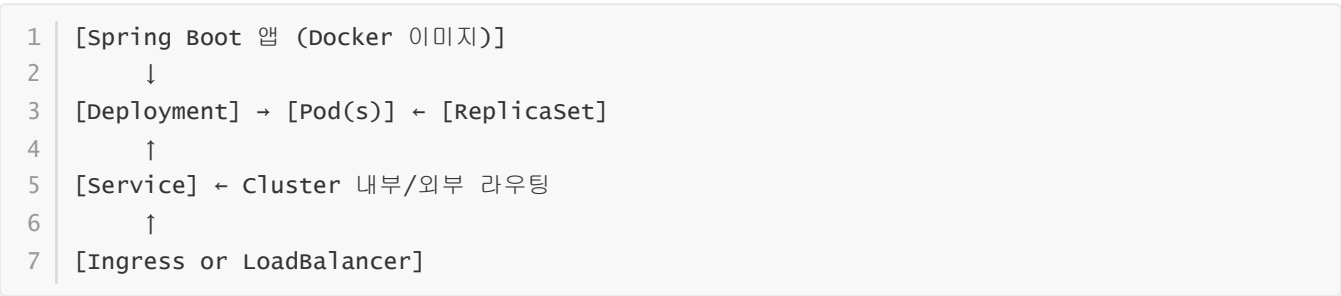


단계	명령
이미지 빌드	<code>docker build -t myapp .</code>
컨테이너 실행	<code>docker run -p 8080:8080 myapp</code>
프로파일 설정	<code>-e SPRING_PROFILES_ACTIVE=prod</code>
로그 볼륨 설정	<code>-v /logs:/app/logs</code>
Compose 구성	<code>docker-compose.yml</code> 사용

## Kubernetes 배포

### 목표

Spring Boot 애플리케이션을 Kubernetes에 다음과 같이 배포:



### 1. 준비 사항

항목	필요
Docker	이미지 빌드용
Kubernetes 클러스터	Minikube, Docker Desktop, EKS/GKE/AKS
kubectl	클러스터 제어
(선택) Helm	배포 템플릿 도구

### 2. Docker 이미지 빌드

```
1  ./gradlew bootJar
2  docker build -t myapp:latest .
```

로컬 Minikube에서 사용 시:

```
1  eval $(minikube docker-env) # Minikube 내부 도커 사용
2  docker build -t myapp:latest .
```

### 3. Kubernetes 리소스 정의

#### 1. Deployment (myapp-deployment.yaml)

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: myapp
5  spec:
6    replicas: 2
7    selector:
8      matchLabels:
9        app: myapp
10   template:
11     metadata:
12       labels:
13         app: myapp
14     spec:
15       containers:
16         - name: myapp-container
17           image: myapp:latest
18           ports:
19             - containerPort: 8080
20           env:
21             - name: SPRING_PROFILES_ACTIVE
22               value: prod
```

#### 2. Service (myapp-service.yaml)

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: myapp-service
5  spec:
6    type: ClusterIP
7    selector:
8      app: myapp
9    ports:
10     - port: 80
11       targetPort: 8080
```

외부에서 접속하려면 `type: LoadBalancer` 또는 Ingress 필요

#### 3. Ingress (선택: myapp-ingress.yaml)

```
1  apiVersion: networking.k8s.io/v1
2  kind: Ingress
3  metadata:
4    name: myapp-ingress
```

```
5 annotations:
6   nginx.ingress.kubernetes.io/rewrite-target: /
7 spec:
8   rules:
9     - host: myapp.local
10      http:
11        paths:
12          - path: /
13            pathType: Prefix
14            backend:
15              service:
16                name: myapp-service
17                port:
18                  number: 80
```

`minikube addons enable ingress`로 Nginx Ingress 활성화 필요

---

## 4. 리소스 배포

```
1 kubectl apply -f myapp-deployment.yaml
2 kubectl apply -f myapp-service.yaml
3 kubectl apply -f myapp-ingress.yaml # 선택
```

---

## 5. 서비스 접근

- **ClusterIP**: 클러스터 내부에서만 접근
- **LoadBalancer**: 외부 IP 자동 할당 (클라우드 환경)
- **Ingress + DNS**: 호스트 기반 라우팅 (ex: `myapp.local`)

```
1 minikube service myapp-service # LoadBalancer 대체
```

---

## 6. 상태 확인

```
1 kubectl get pods
2 kubectl get deployments
3 kubectl get services
4 kubectl describe pod myapp-xxxxx
5 kubectl logs myapp-xxxxx
```

---

## 7. 설정 ConfigMap / Secret 분리

### ConfigMap

```
1 apiVersion: v1
2 kind: ConfigMap
3 metadata:
4   name: myapp-config
5 data:
6   SPRING_PROFILES_ACTIVE: prod
```

### Deployment 적용

```
1 envFrom:
2   - configMapRef:
3     name: myapp-config
```

---

## 8. Health Check

```
1 livenessProbe:
2   httpGet:
3     path: /actuator/health/liveness
4     port: 8080
5   initialDelaySeconds: 15
6   periodSeconds: 10
7
8 readinessProbe:
9   httpGet:
10    path: /actuator/health/readiness
11    port: 8080
12   initialDelaySeconds: 5
13   periodSeconds: 5
```

Spring Boot 2.3+ 이상에서 actuator에서 `/liveness`, `/readiness` 지원

---

## 9. 롤링 업데이트 및 스케일링

```
1 kubectl scale deployment myapp --replicas=4
2 kubectl rollout restart deployment myapp
```

---

## 10. (선택) Helm 패키지화

```
1 helm create myapp-chart
2 # templates/deployment.yaml, service.yaml 설정
3 helm install myapp ./myapp-chart
```

---

✅ 마무리 요약

구성 요소	역할
Deployment	애플리케이션 정의, Replica 관리
Service	Pod 라우팅, 내부 LoadBalancer
Ingress	외부 라우팅 (도메인 → 서비스)
ConfigMap/Secret	설정 분리
Probes	헬스 체크 (liveness/readiness)
Helm	배포 템플릿화

# Helm Chart

## 1. Helm이란?

항목	설명
Helm	Kubernetes용 패키지 매니저
Chart	Helm의 패키지 단위 (YAML 템플릿 + values 설정)
Release	Chart를 실행한 결과 (인스턴스)
Repository	Chart를 저장하는 곳 (로컬/공식/사설 등)

## 2. 설치 및 초기화

```
1 | brew install helm # macOS
2 | choco install kubernetes-helm # windows
```

버전 확인:

```
1 | helm version
```

## 3. Helm Chart 생성

```
1 | helm create spring-boot-app
```

구조:

```
1 spring-boot-app/
2 |─ charts/           # 하위 차트
3 |─ templates/       # k8s 리소스 템플릿
4 |   |─ deployment.yaml
5 |   |─ service.yaml
6 |   |─ ingress.yaml
7 |   |─ _helpers.tpl  # 변수 처리 템플릿
8 |─ values.yaml       # 사용자 정의 변수 파일
9 |─ Chart.yaml        # Chart 메타데이터
```

---

## 4. Chart.yaml 편집

```
1 apiVersion: v2
2 name: spring-boot-app
3 description: Spring Boot Helm Chart
4 version: 0.1.0
5 appVersion: "1.0.0"
```

---

## 5. values.yaml 정의 (사용자 커스터마이징)

```
1 replicaCount: 2
2
3 image:
4   repository: myapp
5   tag: latest
6   pullPolicy: IfNotPresent
7
8 service:
9   type: ClusterIP
10  port: 80
11
12 env:
13   SPRING_PROFILES_ACTIVE: prod
14
15 resources:
16   limits:
17     cpu: 500m
18     memory: 512Mi
19
20 ingress:
21   enabled: true
22   className: nginx
23   hosts:
24     - host: myapp.local
25     paths:
26       - path: /
27       pathType: Prefix
```

---

## 6. templates/deployment.yaml 작성

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: {{ include "spring-boot-app.fullname" . }}
5  spec:
6    replicas: {{ .Values.replicaCount }}
7    selector:
8      matchLabels:
9        app: {{ include "spring-boot-app.name" . }}
10   template:
11     metadata:
12       labels:
13         app: {{ include "spring-boot-app.name" . }}
14     spec:
15       containers:
16         - name: app
17           image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
18           imagePullPolicy: {{ .Values.image.pullPolicy }}
19           ports:
20             - containerPort: 8080
21           env:
22             - name: SPRING_PROFILES_ACTIVE
23               value: {{ .Values.env.SPRING_PROFILES_ACTIVE | quote }}
```

## 7. templates/service.yaml

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: {{ include "spring-boot-app.fullname" . }}
5  spec:
6    type: {{ .Values.service.type }}
7    selector:
8      app: {{ include "spring-boot-app.name" . }}
9    ports:
10     - port: {{ .Values.service.port }}
11       targetPort: 8080
```

## 8. templates/ingress.yaml (옵션)

```
1  {{- if .Values.ingress.enabled }}
2  apiVersion: networking.k8s.io/v1
3  kind: Ingress
4  metadata:
5    name: {{ include "spring-boot-app.fullname" . }}
6  spec:
```

```

7 ingressClassName: {{ .Values.ingress.className }}
8 rules:
9   {{- range .Values.ingress.hosts }}
10   - host: {{ .host }}
11     http:
12       paths:
13         {{- range .paths }}
14         - path: {{ .path }}
15           pathType: {{ .pathType }}
16         backend:
17           service:
18             name: {{ include "spring-boot-app.fullname" $ }}
19             port:
20               number: {{ $.Values.service.port }}
21         {{- end }}
22     {{- end }}
23 {{- end }}

```

## 9. 배포 명령어

```
1 helm install springboot-release ./spring-boot-app
```

→ 이 명령은 `Release` 객체를 생성하고 전체 리소스를 K8s에 적용해.

## 10. 값 오버라이드 방식

```

1 helm install springboot-release ./spring-boot-app \
2   --set image.tag=1.2.3 \
3   --set env.SPRING_PROFILES_ACTIVE=dev

```

또는:

```
1 helm upgrade springboot-release ./spring-boot-app -f custom-values.yml
```

## 11. 상태 확인

```

1 helm list
2 helm status springboot-release
3 helm uninstall springboot-release

```



## 12. 배포 결과 확인

```
1 kubectl get all
2 kubectl describe deployment spring-boot-app
3 kubectl logs -f [pod-name]
```

### ✅ 마무리 요약

구성 요소	역할
<code>Chart.yaml</code>	Helm 차트 정보 (이름, 버전)
<code>values.yaml</code>	사용자가 덮어쓸 수 있는 기본값
<code>templates/*.yaml</code>	Kubernetes 리소스 템플릿
<code>helm install</code>	Helm 배포 실행
<code>helm upgrade</code>	설정 변경 후 롤링 업데이트
<code>helm uninstall</code>	배포 제거 (리소스 자동 삭제)

## CI/CD 구성 (GitHub Actions, Jenkins, GitLab CI)

### 📌 공통 파이프라인 흐름

```
1 [Git Push]
2   ↓
3 [CI: Build]
4   - Gradle/Maven 빌드
5   - 테스트 수행
6   - Docker 이미지 빌드
7   - 이미지 Registry에 푸시
8   ↓
9 [CD: Deploy]
10  - K8s 클러스터 접근
11  - Helm upgrade 또는 kubectl apply
```

### ✅ 1. GitHub Actions 구성

📄 `.github/workflows/deploy.yaml`

```
1 name: CI/CD Pipeline
2
3 on:
4   push:
5     branches: [ main ]
6
```

```
7  env:
8    IMAGE_NAME: ghcr.io/your-org/springboot-app
9
10 jobs:
11   build:
12     runs-on: ubuntu-latest
13     steps:
14       - name: Checkout
15         uses: actions/checkout@v3
16
17       - name: Set up JDK
18         uses: actions/setup-java@v3
19         with:
20           java-version: '17'
21           distribution: 'temurin'
22
23       - name: Build with Gradle
24         run: ./gradlew bootJar
25
26       - name: Log in to GitHub Container Registry
27         run: echo "${{ secrets.GITHUB_TOKEN }}" | docker login ghcr.io -u ${\
github.actor }} --password-stdin
28
29       - name: Build and push Docker image
30         run: |
31           docker build -t $IMAGE_NAME:${{ github.sha }} .
32           docker push $IMAGE_NAME:${{ github.sha }}
33
34   deploy:
35     needs: build
36     runs-on: ubuntu-latest
37     steps:
38       - name: Checkout
39         uses: actions/checkout@v3
40
41       - name: Set up kubectl
42         uses: azure/setup-kubectl@v3
43         with:
44           version: 'v1.27.3'
45
46       - name: Set KUBECONFIG
47         run: |
48           echo "${{ secrets.KUBECONFIG }}" | base64 --decode > kubeconfig.yaml
49           export KUBECONFIG=$(pwd)/kubeconfig.yaml
50
51       - name: Helm upgrade
52         run: |
53           helm upgrade --install springboot ./spring-boot-app \
54             --set image.repository=$IMAGE_NAME \
55             --set image.tag=${{ github.sha }}
```

## Secret 목록

- `GITHUB_TOKEN`: 자동으로 설정됨
- `KUBECONFIG`: base64 인코딩된 kubeconfig 파일

## 2. Jenkins 구성

### Jenkinsfile

```
1 pipeline {
2   agent any
3
4   environment {
5     IMAGE = "myregistry.io/springboot:${env.BUILD_NUMBER}"
6     KUBECONFIG_CRED = credentials('kubeconfig')
7   }
8
9   stages {
10    stage('Checkout') {
11      steps {
12        git 'https://github.com/your-org/springboot-app.git'
13      }
14    }
15
16    stage('Build') {
17      steps {
18        sh './gradlew bootJar'
19      }
20    }
21
22    stage('Docker Build & Push') {
23      steps {
24        sh """
25          docker build -t $IMAGE .
26          docker push $IMAGE
27        """
28      }
29    }
30
31    stage('Deploy to K8s') {
32      steps {
33        withCredentials([file(credentialsId: 'kubeconfig', variable: 'KUBECONFIG')]) {
34          sh """
35            helm upgrade --install springboot ./spring-boot-app \
36              --set image.repository=myregistry.io/springboot \
37              --set image.tag=${env.BUILD_NUMBER}
38          """
39        }
40      }
41    }
42  }
```

`credentialsId: 'kubeconfig'` 는 Jenkins에 등록한 kubeconfig 파일 크리덴셜

### ✓ 3. GitLab CI 구성

#### .gitlab-ci.yml

```

1 stages:
2   - build
3   - deploy
4
5 variables:
6   IMAGE_TAG: $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA
7
8 build:
9   stage: build
10  image: gradle:8.3-jdk17
11  script:
12    - ./gradlew bootJar
13    - docker build -t $IMAGE_TAG .
14    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
15    - docker push $IMAGE_TAG
16
17 deploy:
18   stage: deploy
19   image: bitnami/kubectl:1.27
20   script:
21     - echo "$KUBECONFIG_DATA" | base64 -d > kubeconfig
22     - export KUBECONFIG=$PWD/kubeconfig
23     - helm upgrade --install springboot ./spring-boot-app \
24       --set image.repository=$CI_REGISTRY_IMAGE \
25       --set image.tag=$CI_COMMIT_SHORT_SHA

```

#### GitLab Secret 목록

- `KUBECONFIG_DATA`: base64로 인코딩된 kubeconfig
- `CI_REGISTRY_USER`, `CI_REGISTRY_PASSWORD`: GitLab 컨테이너 레지스트리 인증용

### 공통 Helm 값 파일 (values.yaml)

```

1 image:
2   repository: myregistry.io/springboot
3   tag: latest
4
5 env:
6   SPRING_PROFILES_ACTIVE: prod

```

→ `--set image.tag=...` 으로 덮어쓰기 가능

## ✅ CI/CD 시스템 비교

항목	GitHub Actions	Jenkins	GitLab CI
러너 제공	✅ (GitHub Hosted)	❌ (직접 설치)	✅
빌드 속도	빠름	자유 설정	빠름
Helm/K8s 연동	간단 (action 많음)	자유도 높음	비교적 쉬움
보안/Secret 관리	GitHub Secrets	Jenkins Credentials	GitLab Variables
강점	오픈소스 통합에 강함	유연한 파이프라인	레지스트리 + CI 통합

## 🔒 보안 팁

- kubeconfig는 base64 인코딩해서 환경변수로 등록하고 `echo | decode` 방식으로 사용
- DOCKER\_AUTH, CI 토큰은 절대 Git에 커밋하지 않기
- Helm 값 중 민감정보는 Secret으로 분리 (`--set-string env.JWT_SECRET=...`)

## ✅ 마무리 요약

단계	작업
CI	빌드, 테스트, Docker 이미지 생성 및 Push
CD	Helm upgrade 또는 <code>kubectl apply</code>
Secret	kubeconfig, registry 인증정보, 앱 설정 값 등
툴	GitHub Actions / Jenkins / GitLab CI 모두 가능
운영환경	GKE, EKS, AKS 등 클라우드 클러스터 연동 가능