9. 데이터베이스 연동 및 관리

MySQL, PostgreSQL, Oracle 연동

Spring Boot에서 **MySQL**, **PostgreSQL**, **Oracle**과 같은 **관계형 데이터베이스(RDBMS)**를 연동하려면,
각 DB에 맞는 **JDBC 드라이버**, **Datasource 설정**, **JPA/Hibernate 설정**이 필요합니다.
아래는 각 DB별 설정 예제와 공통 구조를 함께 정리한 **실무 중심 연동 가이드**입니다.

✅ 1. 공통 설정 구조

Spring Boot에서는 `application.yml` 또는 `application.properties`에 다음 항목들을 설정해야 합니다:

항목	설명
`spring.datasource.url`	JDBC 접속 URL
`spring.datasource.username`	DB 사용자 이름
`spring.datasource.password`	비밀번호
`spring.datasource.driver-class-name`	JDBC 드라이버 클래스
`spring.jpa.database-platform`	Hibernate Dialect
`spring.jpa.hibernate.ddl-auto`	테이블 자동 생성/갱신 옵션 (`none`, `validate`, `update`, `create`, `create-drop`)

✅ 2. MySQL 연동

📌 Gradle 의존성

```
implementation 'mysql:mysql-connector-java:8.0.33'
```

📌 application.yml

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/mydb?useSSL=false&serverTimezone=Asia/Seoul
    username: root
    password: root1234
    driver-class-name: com.mysql.cj.jdbc.Driver

  jpa:
    database-platform: org.hibernate.dialect.MySQL8Dialect
    hibernate:
      ddl-auto: update
    show-sql: true
```

```
    properties:
      hibernate.format_sql: true
```

✅ 3. PostgreSQL 연동

📌 Gradle 의존성

```
implementation 'org.postgresql:postgresql:42.6.0'
```

📌 application.yml

```
spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/mydb
    username: postgres
    password: postgres1234
    driver-class-name: org.postgresql.Driver

  jpa:
    database-platform: org.hibernate.dialect.PostgreSQLDialect
    hibernate:
      ddl-auto: update
    show-sql: true
    properties:
      hibernate.format_sql: true
```

✅ 4. Oracle 연동

> Oracle JDBC 드라이버는 Maven Central에 등록되지 않아 보통 수동 설치 또는 Oracle Maven Repository 사용 필요

📌 Gradle 의존성 (Oracle Maven 등록된 경우)

```
implementation 'com.oracle.database.jdbc:ojdbc8:19.3.0.0'
```

📌 application.yml

```
spring:
  datasource:
    url: jdbc:oracle:thin:@localhost:1521:xe
    username: oracle_user
    password: oracle_pass
    driver-class-name: oracle.jdbc.OracleDriver

  jpa:
    database-platform: org.hibernate.dialect.Oracle12cDialect
    hibernate:
      ddl-auto: update
    show-sql: true
```

```
    properties:
      hibernate.format_sql: true
```

✅ 5. Dialect 설정 참고

DB	Dialect
MySQL 8	`org.hibernate.dialect.MySQL8Dialect`
PostgreSQL	`org.hibernate.dialect.PostgreSQLDialect`
Oracle 12c	`org.hibernate.dialect.Oracle12cDialect`

> Dialect는 Hibernate가 DB별 SQL 문법 차이를 맞추기 위해 사용

✅ 6. DataSource 설정 정리표

항목	MySQL	PostgreSQL	Oracle
Driver	`com.mysql.cj.jdbc.Driver`	`org.postgresql.Driver`	`oracle.jdbc.OracleDriver`
URL 예	`jdbc:mysql://localhost:3306/db`	`jdbc:postgresql://localhost:5432/db`	`jdbc:oracle:thin:@localhost:1521:xe`
Dialect	`MySQL8Dialect`	`PostgreSQLDialect`	`Oracle12cDialect`

✅ 7. 유용한 설정 추가 (공통)

```
spring:
  jpa:
    open-in-view: false  # 권장: 트랜잭션 외 지연로딩 방지
    defer-datasource-initialization: true  # SQL 실행 시점 제어
    generate-ddl: true
  sql:
    init:
      mode: always  # data.sql 자동 실행 여부
```

✅ 8. 다중 DB 연결 (멀티 DataSource)

> 실무에서는 MySQL + Redis 또는 MySQL + Oracle 조합처럼
> 여러 DB를 동시에 사용하는 경우도 있음 → `@Primary`, `@Qualifier`, `JpaTransactionManager`로 분리 관리

✅ 9. 테스트 환경: Testcontainers 연동

Testcontainers를 사용하면 각 DB 환경을 Docker로 실행해서 테스트 가능:

```
@Container
static MySQLContainer<?> mysql = new MySQLContainer<>("mysql:8.0")
    .withDatabaseName("test")
    .withUsername("test")
    .withPassword("test");
```

```
@DynamicPropertySource
static void setDatasource(DynamicPropertyRegistry registry) {
    registry.add("spring.datasource.url", mysql::getJdbcUrl);
    registry.add("spring.datasource.username", mysql::getUsername);
    registry.add("spring.datasource.password", mysql::getPassword);
}
```

✅ 결론 요약

항목	설명
공통 설정	`spring.datasource.*`, `spring.jpa.*` 중심
DB별 설정	JDBC URL, Driver, Dialect 주의
테스트	`@DataJpaTest`, `Testcontainers`, `H2` 대체 또는 실제 DB
실무 권장	`application.yml` → Profile로 분리 (`application-dev.yml`, `application-test.yml`)

H2, SQLite 등 인메모리 DB 활용

Spring Boot에서 **H2, SQLite**와 같은 **인메모리(또는 경량형) 데이터베이스**를 활용하면
실제 운영용 DB(MySQL, PostgreSQL, Oracle 등) 없이도 빠르게 테스트나 개발 환경을 구성할 수 있습니다.
이들은 특히 **단위 테스트, 슬라이스 테스트, 로컬 개발 환경, CI 자동화 환경**에서 매우 유용합니다.

아래에 두 가지 인메모리/경량 DB의 특성과 설정 방법을 실무 중심으로 정리해드립니다.

✅ 1. H2 데이터베이스

📌 특징

항목	내용
용도	**순수 인메모리 모드** 또는 파일 모드 가능
장점	가볍고 빠르며, JPA와 잘 통합됨
실행 방식	웹 브라우저 콘솔 제공 (`/h2-console`)
SQL 호환성	MySQL과 유사하지만 완전 동일하진 않음

📌 Gradle 의존성

```
runtimeOnly 'com.h2database:h2'
```

📌 application.yml 예시

```yaml
spring:
  datasource:
    url: jdbc:h2:mem:testdb
    driver-class-name: org.h2.Driver
    username: sa
    password:
  h2:
    console:
      enabled: true
      path: /h2-console
  jpa:
    hibernate:
      ddl-auto: create-drop
    show-sql: true
    properties:
      hibernate.format_sql: true
```

> `jdbc:h2:mem:testdb`는 테스트가 끝나면 메모리에서 사라짐

📌 파일 모드 (디스크 저장형)

```
spring.datasource.url: jdbc:h2:file:./data/testdb
```

> 메모리 모드보다 느리지만 지속성 있음

📌 H2 콘솔 접속

- 주소: `http://localhost:8080/h2-console`
- JDBC URL: `jdbc:h2:mem:testdb`
- 사용자: `sa`, 비밀번호 없음

✅ 2. SQLite 데이터베이스

📌 특징

항목	내용
용도	**단일 파일 기반** 경량 DB (모바일, 데스크탑 앱에서도 사용)
장점	운영체제 의존 없음, 로컬 테스트에 적합

항목	내용
단점	JPA 통합 시 Dialect, 기능 제한 등 설정 주의 필요

📌 Gradle 의존성

```
implementation 'org.xerial:sqlite-jdbc:3.43.2.2'
```

📌 application.yml 예시

```
spring:
  datasource:
    url: jdbc:sqlite:db/test.sqlite
    driver-class-name: org.sqlite.JDBC
  jpa:
    database-platform: org.hibernate.community.dialect.SQLiteDialect
    hibernate:
      ddl-auto: update
    show-sql: true
```

> `SQLiteDialect`는 Hibernate 6 이상 또는 별도 커스텀 필요

📌 커스텀 Dialect 예시 (Hibernate 5.x 이하일 경우)

SQLite는 공식 Hibernate Dialect가 없을 수 있음 → 직접 구현:

```
public class SQLiteDialect extends Dialect {
    public SQLiteDialect() {
        super();
        registerColumnType(Types.VARCHAR, "text");
        registerColumnType(Types.INTEGER, "integer");
        ...
    }
    @Override
    public boolean supportsIdentityColumns() {
        return true;
    }
}
```

```
spring.jpa.database-platform: com.example.db.SQLiteDialect
```

✅ 3. 테스트에서 활용 (예: `@DataJpaTest`)

```
@DataJpaTest
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.ANY)
class RepositoryTest {
    ...
}
```

→ `H2` 또는 `application-test.yml`에 따라 SQLite 활용 가능

✅ 4. H2 vs SQLite 비교

항목	H2	SQLite
형태	인메모리 or 파일	파일 기반
속도	매우 빠름	빠름
SQL 호환성	MySQL 유사	독립적 SQL 구문
테스트	단위 테스트 최적	경량 단일 DB 테스트 적합
사용 추천	CI / 테스트 자동화	임베디드 환경, 로컬 DB

✅ 5. 실무 활용 전략

목적	권장 DB
테스트 자동화 (CI)	`H2` 인메모리
로컬 API 서버, 데모	`SQLite`
테스트 슬라이스 (`@DataJpaTest`)	기본 H2 (Spring 자동 구성됨)
TestContainers 불가 환경	SQLite 대안 가능

✅ 결론 요약

항목	설명
H2	인메모리로 빠르고 테스트에 최적화됨. Spring과 자연스럽게 통합
SQLite	파일 기반 경량 DB. 간편한 운영/테스트 DB로 활용 가능
공통점	별도 설치 필요 없음. 테스트 또는 로컬용으로 우수
차이점	H2는 자바에서 개발됨, SQLite는 C 기반. SQL 문법 차이 있음
주의점	운영용 DB(MySQL 등)과의 SQL 차이로 마이그레이션 전 테스트는 별도 필요

트랜잭션 처리 (`@Transactional`)

Spring Framework의 `@Transactional`은 데이터베이스 작업에 있어서 **트랜잭션의 경계를 선언적으로 지정**하는 핵심 기능입니다.
DB 일관성과 무결성을 유지하면서도, 코드의 간결성과 유지보수성을 높여주는 **스프링 기반 애플리케이션의 필수 개념**입니다.

✅ 1. 트랜잭션이란?

> 여러 DB 작업을 **하나의 논리적 단위로 묶어 일괄적으로 처리**하는 기법.
> 작업 중 하나라도 실패하면 전체 작업을 **롤백(Rollback)**하여 데이터 일관성을 유지한다.

✅ 2. `@Transactional`이란?

> 메서드 또는 클래스에 붙여, 그 범위 내의 DB 작업을 **트랜잭션으로 감싸고**,
> **성공 시 커밋, 예외 발생 시 롤백**되도록 하는 선언형 애노테이션이다.

✅ 3. 사용 예제

```
@Service
public class MemberService {

    @Transactional
    public void registerMember(Member member) {
        memberRepository.save(member);
        welcomeEmailService.send(member); // 예외 발생 시 롤백
    }
}
```

- 위 메서드는 두 작업 모두 성공해야 DB에 반영됨.
- 중간에 `send()`에서 예외가 발생하면 `save()`도 롤백됨.

✅ 4. 애노테이션 사용 위치

위치	의미
클래스	모든 public 메서드에 트랜잭션 적용
메서드	해당 메서드에만 트랜잭션 적용 (우선 순위 높음)

✅ 5. 롤백 대상

기본적으로 `RuntimeException`, `Error`가 발생하면 롤백됨.
`Checked Exception`은 **자동 롤백되지 않음**.

❗ 예외 전략 요약

예외 타입	기본 처리	수동 지정
RuntimeException	자동 롤백	-
Checked Exception	커밋됨	`rollbackFor =` 옵션 필요

```
@Transactional(rollbackFor = IOException.class)
public void uploadFile() throws IOException {
    ...
}
```

✅ 6. 주요 속성 정리

```
@Transactional(
  propagation = Propagation.REQUIRED,
  isolation = Isolation.DEFAULT,
  readOnly = false,
  rollbackFor = Exception.class
)
```

속성	설명
`propagation`	전파 방식 (기존 트랜잭션과 어떻게 결합할지)
`isolation`	트랜잭션 격리 수준 (READ_COMMITTED 등)
`readOnly`	읽기 전용 여부 (쿼리 최적화)
`rollbackFor`	롤백할 예외 지정

✅ 7. 전파 수준 (`propagation`)

옵션	설명
`REQUIRED`	기본값. 트랜잭션이 있으면 참여, 없으면 새로 시작
`REQUIRES_NEW`	항상 새로운 트랜잭션 시작 (기존 중단됨)
`NESTED`	기존 트랜잭션 내에서 중첩 트랜잭션 (Savepoint 사용)
`MANDATORY`	반드시 기존 트랜잭션이 있어야 함

옵션	설명
`NEVER`	트랜잭션 있으면 예외
`SUPPORTS`	트랜잭션 있으면 참여, 없으면 비트랜잭션
`NOT_SUPPORTED`	트랜잭션을 사용하지 않음 (중단 후 실행)

✅ 8. 격리 수준 (`isolation`)

수준	설명
`DEFAULT`	DB 설정 따름
`READ_UNCOMMITTED`	커밋되지 않은 데이터도 읽음 (dirty read)
`READ_COMMITTED`	커밋된 데이터만 읽음 (Oracle 기본)
`REPEATABLE_READ`	같은 쿼리 반복 시 같은 결과 보장
`SERIALIZABLE`	가장 엄격, 동시성 낮음 (테이블 락 수준)

✅ 9. readOnly = true

- 트랜잭션은 시작하지만 **쓰기 작업 금지**
- 특정 DB (Oracle 등)에서는 **쿼리 최적화 힌트로 작동**
- 주로 조회 전용 서비스에서 사용

```
@Transactional(readOnly = true)
public List<Member> getMembers() {
    return memberRepository.findAll();
}
```

✅ 10. 테스트에서 사용 시

```
@SpringBootTest
@Transactional
class ServiceIntegrationTest {
    // 각 테스트 메서드 종료 시 자동 롤백
}
```

✅ 11. 내부 호출 주의 (프록시 문제)

```
@Transactional
public void methodA() {
    methodB(); // ❌ 같은 클래스의 내부 호출 → 트랜잭션 적용 안 됨
}

@Transactional
public void methodB() { ... }
```

> 해결: 메서드를 외부 컴포넌트로 분리하거나 AOP Proxy로 호출되도록 구조 조정 필요

✅ 12. 결론 요약

항목	설명
`@Transactional`	DB 작업을 하나의 트랜잭션 단위로 묶음
기본 롤백 대상	`RuntimeException`, `Error`
Checked 예외	`rollbackFor` 명시 필요
readOnly	조회 최적화 시 사용
실무 전략	Service 계층 중심으로 선언, 테스트 클래스에서는 자동 롤백 활용

커넥션 풀(HikariCP) 설정

Spring Boot 2 이상에서는 **기본 커넥션 풀(Connection Pool)**로 `HikariCP`가 자동으로 설정됩니다.
HikariCP는 **가볍고 빠르며, 안정적인 JDBC 커넥션 풀 구현체**로
대부분의 Spring Boot 실무 프로젝트에서 표준으로 사용됩니다.

아래는 **HikariCP의 개념, 기본 작동 원리, 설정 항목, 성능 튜닝 전략**을 모두 포함한 실무 중심 가이드입니다.

✅ 1. 커넥션 풀이란?

> JDBC 커넥션은 생성/소멸에 높은 비용이 드는 자원이므로,
> 커넥션을 **미리 생성하여 풀(Pool)로 보관**하고 재사용하는 기술.

목적	설명
성능 개선	커넥션 재사용으로 생성/종료 오버헤드 제거
자원 절약	동시 접속 수 제어로 DB 부하 감소
안정성 향상	커넥션 유실, 누수, 과다 접속 방지

✅ 2. HikariCP란?

항목	설명
구현체	Java로 작성된 초경량 커넥션 풀
Spring Boot 기본 채택	Spring Boot 2.x+의 기본 `DataSource`
장점	빠름, 메모리 효율적, JMX 지원, 자동 회복
대체 후보	Apache DBCP2, Tomcat JDBC Pool 등 (현재는 거의 Hikari 사용)

✅ 3. 의존성

```
implementation 'com.zaxxer:HikariCP:5.1.0' // Spring Boot 사용 시 생략 가능
```

✅ 4. application.yml 설정 예시

```yaml
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/mydb
    username: user
    password: pass
    driver-class-name: com.mysql.cj.jdbc.Driver
    hikari:
      pool-name: MyHikariPool
      maximum-pool-size: 20
      minimum-idle: 5
      idle-timeout: 600000       # 10분
      max-lifetime: 1800000      # 30분
      connection-timeout: 30000  # 커넥션 요청 최대 대기 시간
      validation-timeout: 5000
      connection-test-query: SELECT 1
```

✅ 5. 주요 설정 옵션 설명

항목	기본값	설명
`maximum-pool-size`	10	풀에서 동시에 허용할 최대 커넥션 수
`minimum-idle`	same as max	최소 대기 커넥션 수
`idle-timeout`	600000ms	유휴 커넥션이 회수되는 시간
`max-lifetime`	1800000ms	커넥션의 최대 수명 (DB가 종료 전 종료 방지)
`connection-timeout`	30000ms	커넥션 요청 대기 최대 시간

항목	기본값	설명
`validation-timeout`	5000ms	커넥션 유효성 검사 최대 시간
`connection-test-query`	DB별로 다름	커넥션 상태 확인 쿼리 (Hikari는 `isValid()`로 대체 가능)

✅ 6. JMX, 메트릭 연동

Spring Actuator를 함께 사용할 경우 HikariCP의 **풀 상태 모니터링 가능**:

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

```
management:
  endpoints:
    web:
      exposure:
        include: health, metrics
```

접속 예:

```
/actuator/metrics/hikaricp.connections.active
/actuator/metrics/hikaricp.connections.max
```

✅ 7. 실시간 풀 상태 확인 (Actuator)

```
GET /actuator/metrics/hikaricp.connections.active
```

```
{
  "name": "hikaricp.connections.active",
  "measurements": [{"statistic": "VALUE", "value": 2}],
  "availableTags": [...]
}
```

✅ 8. 커스텀 HikariConfig 사용 (Java 기반)

```
@Bean
@ConfigurationProperties("spring.datasource.hikari")
public HikariDataSource dataSource() {
    return DataSourceBuilder.create()
        .type(HikariDataSource.class)
        .build();
}
```

또는

```java
@Bean
public DataSource customHikari() {
    HikariConfig config = new HikariConfig();
    config.setJdbcUrl("jdbc:mysql://localhost:3306/mydb");
    config.setUsername("user");
    config.setPassword("pass");
    config.setMaximumPoolSize(20);
    return new HikariDataSource(config);
}
```

✅ 9. 주의사항

항목	권장/주의
`max-lifetime` < DB 커넥션 timeout	반드시 조정 필요 (DB 커넥션보다 짧게)
JDBC Driver 자동 등록 확인	일부 DB에서 드라이버 설정 누락 주의
Spring Boot 2 이상	자동으로 Hikari 사용됨 (따로 설정하지 않아도 적용)

✅ 10. 결론 요약

항목	설명
기본 커넥션 풀	Spring Boot 2+에서는 **HikariCP**
설정 위치	`spring.datasource.hikari.*`
핵심 설정	`maximum-pool-size`, `idle-timeout`, `connection-timeout` 등
실무 팁	JMX 모니터링, `actuator` 연동으로 상태 확인 권장
장점	빠르고 안정적인 커넥션 풀, 테스트에서도 그대로 활용 가능

Flyway / Liquibase로 DB 마이그레이션 관리

Spring Boot에서 **DB 마이그레이션(스키마 버전 관리)**을 안전하고 체계적으로 수행하려면
Flyway 또는 **Liquibase**를 사용하는 것이 실무 표준입니다.
이 두 도구는 **버전 기반의 SQL 변경 이력을 코드로 관리**하며,
DB를 수동 수정하지 않고도 일관된 스키마 변경 및 배포를 가능하게 합니다.

✅ 1. Flyway vs Liquibase 간단 비교

항목	Flyway	Liquibase
언어 지원	SQL 위주 (Java도 가능)	SQL + XML + YAML + JSON
학습 난이도	쉽고 간단	유연하나 약간 복잡

항목	Flyway	Liquibase
통합성	Spring Boot와 매우 강력하게 통합	Spring Boot와 공식 연동 지원
실무 활용	단순 마이그레이션에 강함	대규모 팀, 조건 분기 있는 DB 작업에 적합

✅ 2. Flyway 설정

📌 Gradle 의존성

```
implementation 'org.flywaydb:flyway-core'
```

> Spring Boot 3.1+에서는 자동으로 인식되며, **의존성만 추가하면 작동**

📌 기본 구조

- **디폴트 경로**: `src/main/resources/db/migration`
- **파일명 규칙**: `V<버전>__<설명>.sql`

```
V1__create_member_table.sql
V2__add_column_email.sql
```

📌 application.yml 설정

```
spring:
  flyway:
    enabled: true
    locations: classpath:db/migration
    baseline-on-migrate: true
    validate-on-migrate: true
    out-of-order: false
```

📌 예시 SQL

```
-- V1__create_member_table.sql
CREATE TABLE member (
  id BIGINT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(100),
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

📌 실행 시점

- **애플리케이션 시작 시점에 자동 실행됨**
- DB에 `flyway_schema_history` 테이블이 생성되어 **적용 이력 관리**

✅ 3. Liquibase 설정

📌 Gradle 의존성

```
implementation 'org.liquibase:liquibase-core'
```

📌 application.yml 설정

```
spring:
  liquibase:
    change-log: classpath:db/changelog/db.changelog-master.yaml
    enabled: true
```

📌 마이그레이션 파일 구조 (YAML 예시)

```
databaseChangeLog:
  - changeSet:
      id: 1
      author: kim
      changes:
        - createTable:
            tableName: member
            columns:
              - column:
                  name: id
                  type: BIGINT
                  autoIncrement: true
                  constraints:
                    primaryKey: true
              - column:
                  name: name
                  type: VARCHAR(100)
              - column:
                  name: created_at
                  type: timestamp
                  defaultValueComputed: CURRENT_TIMESTAMP
```

📌 파일명 구조

- 중앙 진입 파일: `db.changelog-master.yaml`
- 다른 파일을 import하거나 여러 버전으로 분할 가능

📌 실행 시점

- **애플리케이션 부트 시점에 자동 실행**
- Liquibase는 `DATABASECHANGELOG` 테이블을 사용하여 이력 관리

✅ 4. 공통 기능 비교

기능	Flyway	Liquibase
자동 버전 추적	✅	✅
Rollback	❌ (지원 안 함, 수동 작성 필요)	✅ (명시 가능)
SQL + Java Script 지원	✅	✅
YAML, XML 지원	❌	✅
CLI 지원	✅	✅
DB 타입 자동 감지	✅	✅
Spring Boot 자동 통합	✅	✅

✅ 5. 실무 활용 전략

전략	설명
개발 초반	Flyway + SQL 기반으로 간단히 시작
복잡한 마이그레이션	Liquibase로 롤백, 분기 처리 등 유연하게 관리
팀 협업	Git에 마이그레이션 파일 버전 관리 필수
CI/CD 환경	시작 시 자동 반영 or `migrateOnly=true` 설정
H2 테스트	테스트 시에도 동일하게 적용되도록 설정 유지 필요

✅ 6. Spring Boot 자동 실행 동작 요약

항목	Flyway	Liquibase
기본 경로	`classpath:db/migration`	`classpath:db/changelog/db.changelog-master.yaml`

항목	Flyway	Liquibase
히스토리 테이블	`flyway_schema_history`	`DATABASECHANGELOG`
적용 시점	`SpringApplication.run()` 직전	

✅ 결론 요약

항목	설명
Flyway	단순하고 빠르게 마이그레이션 수행. SQL 기반에 최적
Liquibase	XML/YAML 기반, 롤백, 조건 분기 등 유연한 기능 지원
공통	DB 스키마 버전을 코드로 안전하게 관리 가능
실무 팁	Git에서 `v1`, `v2` 등으로 버전 관리하고, **모든 배포 시점에 자동 적용**되도록 설정