# 11. 테스트와 품질 관리

## 단위 테스트와 통합 테스트

Spring 기반 애플리케이션에서 **단위 테스트(Unit Test)**와 **통합 테스트(Integration Test)**는
코드의 품질과 안정성을 확보하기 위한 핵심 기법이다.
두 테스트는 **대상 범위, 실행 속도, 구성 복잡도, 검증 목적** 등이 서로 다르다.

---

### 📘 1. 개념 비교

| 구분 | 단위 테스트 (Unit Test) | 통합 테스트 (Integration Test) |
| --- | --- | --- |
| 테스트 대상 | **하나의 클래스나 메서드** | **여러 컴포넌트 간 실제 동작 흐름 전체** |
| 범위 | 작음 | 큼 |
| Spring Context 사용 | ❌ 없음 또는 최소한만 사용 | ✅ 전체 ApplicationContext 로드 |
| 속도 | 빠름 | 느림 (DB, 네트워크 등 I/O 포함) |
| 목적 | 로직 검증, 경계 조건 확인 | Bean 연결, DB 트랜잭션, REST, 보안 흐름 등 실제 동작 검증 |
| 격리도 | 높음 (Mocking 중심) | 낮음 (실제 환경 중심) |

---

### 🔧 2. 단위 테스트: 핵심 설정과 예시

#### ✅ 사용 어노테이션

| 어노테이션 | 설명 |
| --- | --- |
| `@ExtendWith(MockitoExtension.class)` | JUnit 5 + Mockito |
| `@Mock`, `@InjectMocks` | 의존성 모킹 |
| `@Test` | JUnit 테스트 메서드 |

#### ✅ 예제

```
@ExtendWith(MockitoExtension.class)
class UserServiceTest {

    @Mock
    private UserRepository userRepository;

    @InjectMocks
    private UserService userService;
```

```
9
10      @Test
11      void findUserById_success() {
12          User user = new User(1L, "kim");
13          given(userRepository.findById(1L)).willReturn(Optional.of(user));
14
15          User result = userService.findById(1L);
16
17          assertEquals("kim", result.getName());
18      }
19  }
```

→ 실제 DB 연결 없이 **UserService만 검증**

---

## 🔁 3. 통합 테스트: 전체 스프링 컨텍스트 포함

### ✅ 사용 어노테이션

| 어노테이션 | 설명 |
| --- | --- |
| `@SpringBootTest` | 전체 ApplicationContext 로드 |
| `@Transactional` | 테스트 후 자동 롤백 |
| `@AutoConfigureMockMvc` | `MockMvc` 자동 설정 |
| `@TestRestTemplate` | 실제 HTTP 요청 테스트 (서버 부팅 필요) |

### ✅ 예제

```
1   @SpringBootTest
2   @AutoConfigureMockMvc
3   class UserControllerIntegrationTest {
4
5       @Autowired
6       private MockMvc mockMvc;
7
8       @Test
9       void getUser_success() throws Exception {
10          mockMvc.perform(get("/users/1"))
11              .andExpect(status().isOk())
12              .andExpect(jsonPath("$.name").value("kim"));
13      }
14  }
```

→ `UserController` + `Service` + `Repository` + DB까지 통합된 흐름을 테스트

---

## 🧪 4. 테스트 환경 구성

### ✅ application.yml (테스트 전용)

```yaml
spring:
  datasource:
    url: jdbc:h2:mem:testdb
  jpa:
    hibernate:
      ddl-auto: create-drop
```

→ `@SpringBootTest`에서는 기본적으로 `application-test.yml`을 사용 가능
→ 메모리 DB를 통해 테스트 간 격리 및 속도 확보

---

## ⚙️ 5. MockMvc vs TestRestTemplate

| 도구 | 설명 |
|---|---|
| `MockMvc` | 컨트롤러를 서블릿 컨텍스트 없이 테스트 (빠름) |
| `TestRestTemplate` | 실제 포트에서 전체 서버 부팅 후 테스트 (느림) |

→ 단일 컨트롤러 테스트: `MockMvc`, E2E 테스트: `TestRestTemplate`

---

## 🧠 6. 실무 전략

| 전략 | 설명 |
|---|---|
| **단위 테스트로 기본 로직 보장** | Service, Util 등은 Mockito로 완전 격리 테스트 |
| **통합 테스트로 연결성 보장** | 컨트롤러 → 서비스 → DB까지 흐름 점검 |
| **배포 전 회귀 테스트로 전체 확인** | `@SpringBootTest` + `TestContainers` (실DB) |
| **Mock은 단위에서만 사용** | 통합에서는 실제 환경을 선호 |

---

## ⚠️ 7. 주의사항

| 항목 | 설명 |
|---|---|
| 단위 테스트에 DB 의존성 포함 금지 | → 통합 테스트로 분리할 것 |
| 통합 테스트에서 캐시/시큐리티 주의 | @WithMockUser 등 사용 필요 |
| 테스트 간 상태 공유 방지 | @DirtiesContext 또는 트랜잭션 롤백 사용 |
| 테스트와 운영 환경 분리 | `application-test.yml` 사용 + 프로필 설정 |

---

## 📌 8. 요약 비교

| 항목 | 단위 테스트 | 통합 테스트 |
|---|---|---|
| 속도 | ✅ 빠름 | ❌ 느림 |
| 대상 | 단일 클래스, 단일 메서드 | 컨트롤러~DB 전체 |
| 의존성 | 대부분 Mock | 실제 컴포넌트 |
| Spring Context | ❌ 없이도 가능 (`@ExtendWith`) | ✅ 전체 or 일부 컨텍스트 사용 |
| 어노테이션 | `@Mock`, `@InjectMocks` | `@SpringBootTest`, `@Transactional` |
| 검증 목적 | 로직 정확성 | 흐름/연결/환경 구성 검증 |

# JUnit 5 설정

Spring Boot와 함께 사용하는 **JUnit 5**(정식 명칭: *JUnit Jupiter*)는 **현대 자바 테스트 표준 프레임워크**이다.
JUnit 5는 이전 버전(JUnit 4)과는 아키텍처, 어노테이션, 확장성에서 많은 차이를 가지고 있으며,
Spring Boot 2.2+부터는 **기본 테스트 엔진**으로 JUnit 5가 채택되어 별도 설정 없이도 바로 사용할 수 있다.

---

## 📘 1. JUnit 5 구조 개요

JUnit 5는 3개의 모듈로 구성됨:

| 구성 요소 | 설명 |
|---|---|
| **JUnit Platform** | 테스트 실행 기반 (`IDE`, `빌드 툴`, `Console Launcher`) |
| **JUnit Jupiter** | 실제 테스트 API (`@Test`, `@BeforeEach`, `@DisplayName` 등) |
| **JUnit Vintage** | JUnit 4 이전 버전 호환 실행기 (선택사항) |

---

## ⚙️ 2. Gradle 설정

```
dependencies {
    testImplementation 'org.springframework.boot:spring-boot-starter-test' // 포함됨
    testImplementation 'org.junit.jupiter:junit-jupiter'
}
```

### 추가 설정 (build.gradle)

```
test {
    useJUnitPlatform()  // 필수: JUnit5 엔진 활성화
}
```

---

## ⚙️ 3. Maven 설정

```xml
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <exclusions>
      <exclusion>
        <groupId>org.junit.vintage</groupId>
        <artifactId>junit-vintage-engine</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
</dependencies>
```

→ Spring Boot 2.4 이상은 `junit-jupiter` 자동 설정됨

---

## 🧪 4. JUnit 5 주요 어노테이션

| 어노테이션 | 설명 |
|---|---|
| `@Test` | 테스트 메서드 지정 |
| `@BeforeEach` | 각 테스트 실행 전 수행 |
| `@AfterEach` | 각 테스트 후 정리 |
| `@BeforeAll` | 전체 테스트 시작 전 1회 수행 (static 필요) |
| `@AfterAll` | 전체 테스트 끝난 후 1회 수행 (static 필요) |
| `@DisplayName("...")` | 테스트 이름 설정 (한글 가능) |
| `@Nested` | 중첩 테스트 클래스 작성 |
| `@Disabled` | 테스트 일시 중단 |
| `@Tag("slow")` | 태그 분류로 실행 필터링 가능 |

---

## 🔁 5. Spring과 연동

### ✅ 기본 Spring 테스트 (통합)

```
@SpringBootTest
class MyServiceIntegrationTest {

    @Autowired
    MyService myService;

```

```
    @Test
    void 서비스_동작_테스트() {
        ...
    }
}
```

### ✅ 슬라이스 테스트

| 어노테이션 | 대상 |
| --- | --- |
| `@WebMvcTest` | Controller 계층만 로드 |
| `@DataJpaTest` | Repository + EntityManager만 로드 |
| `@MockBean` | Spring Bean을 Mock으로 교체 |

---

## 📂 6. 디렉토리 구조 예시

```
src/
 ├─ main/
 |    └─ java/...
 └─ test/
     └─ java/
         └─ com.example.MyServiceTest
```

---

## 🧠 7. 테스트 예제

```
@DisplayName("유저 서비스 테스트")
class UserServiceTest {

    private UserService userService;

    @BeforeEach
    void setUp() {
        userService = new UserService();
    }

    @Test
    @DisplayName("이름으로 유저 조회")
    void findByName() {
        User user = userService.findByName("kim");
        assertEquals("kim", user.getName());
    }
}
```

---

## ✅ 8. 실무 전략

| 전략 | 설명 |
|---|---|
| 테스트 명칭 한글 사용 | `@DisplayName("회원 가입 성공 테스트")` |
| 테스트 격리 | 모든 테스트는 상태 공유 없이 독립 실행 |
| JUnit 5 + Mockito + AssertJ 조합 | 가독성과 표현력 강화 |
| CI/CD 연동을 위해 **태그 전략** 사용 | `@Tag("integration")` 로 분류 실행 가능 |
| 반복 테스트 | `@RepeatedTest(n)` 활용 가능 |

---

## ⚠️ 9. 자주 발생하는 오류

| 문제 | 원인 |
|---|---|
| `No tests found` | `useJUnitPlatform()` 누락 |
| `NoSuchMethodError` | JUnit 4 + 5 혼용 시 버전 충돌 |
| `Method must be static` | `@BeforeAll`, `@AfterAll`는 static 필요 |
| `Spring Boot context 오류` | 테스트 시 `@SpringBootTest` → DB 설정 또는 Bean 누락 확인 |

---

## 📌 10. 요약

| 항목 | 설명 |
|---|---|
| 버전 구조 | Platform + Jupiter + Vintage |
| 기본 설정 | `spring-boot-starter-test` 포함 |
| 실행 방식 | JUnit5: `useJUnitPlatform()` 필수 |
| Spring 연동 | `@SpringBootTest`, `@WebMvcTest`, `@MockBean` 등 |
| 실무 전략 | `@DisplayName`, 태그 전략, 격리성, 반복성 보장 |

# Spring TestContext Framework

**Spring TestContext Framework**는 Spring에서 제공하는 **통합 테스트 지원 인프라**이다.
단순히 `@SpringBootTest`를 사용하는 것 그 이상으로,
**ApplicationContext 관리, 트랜잭션 지원, 의존성 주입, 컨텍스트 캐싱, 테스트 실행 제어** 등
테스트를 위한 전반적인 기능을 담당하는 **백그라운드 프레임워크**이다.

---

## 📘 1. Spring TestContext Framework란?

Spring이 **JUnit, TestNG, 기타 테스트 프레임워크와 무관하게**
**공통 테스트 기능을 제공하기 위해 만든 메타 프레임워크**이다.

> ❗ 즉, `@SpringBootTest`, `@WebMvcTest`, `@DataJpaTest` 등은 전부 이 TestContext Framework 위에서 작동한다.

---

## 🧱 2. 핵심 구성 요소

| 구성 요소 | 설명 |
|---|---|
| `TestContextManager` | 전체 테스트 실행 컨텍스트를 조율하는 관리자 |
| `TestContext` | ApplicationContext, test instance, method, class 등을 포함하는 컨텍스트 |
| `TestExecutionListener` | 테스트 실행 전/후 이벤트를 처리하는 리스너 인터페이스 |
| `ContextLoader` | ApplicationContext를 로드하는 전략 클래스 |
| `@ContextConfiguration` | 수동으로 컨텍스트 설정할 때 사용 |
| `@TestExecutionListeners` | 리스너를 확장하거나 대체할 때 사용 |

---

## 🔧 3. 지원 기능 요약

| 기능 | 설명 |
|---|---|
| ApplicationContext 자동 로딩 | `@SpringBootTest`, `@ContextConfiguration` 등을 통해 로드 |
| DI 지원 | `@Autowired`, `@Value`, `@MockBean` 등 자동 주입 |
| 트랜잭션 지원 | `@Transactional` + 롤백 |
| 컨텍스트 캐싱 | 테스트 속도 최적화 |
| 리스너 확장 | 실행 전후 Hook (`beforeTestMethod`, `afterTestExecution` 등) |

---

## 🧩 4. 대표 어노테이션 및 기능

| 어노테이션 | 역할 |
|---|---|
| `@SpringBootTest` | 전체 컨텍스트 로드 (통합 테스트용) |
| `@ContextConfiguration` | 수동 ApplicationContext 구성 지정 |
| `@TestExecutionListeners` | 리스너 커스터마이징 |
| `@DirtiesContext` | 테스트 종료 후 컨텍스트 폐기 |

| 어노테이션 | 역할 |
| --- | --- |
| `@Transactional` | 테스트 트랜잭션 + 자동 롤백 |
| `@Commit`, `@Rollback` | 롤백 여부 지정 |
| `@TestPropertySource` | 테스트 전용 프로퍼티 지정 |
| `@Sql` | DB 초기화 스크립트 실행 |

---

## ⚙️ 5. 동작 흐름

```
테스트 클래스 실행
  ↓
JUnit ↔ TestContextManager 연동
  ↓
ApplicationContext 로딩 (캐시 or 새로 생성)
  ↓
DI 주입 (@Autowired 등)
  ↓
@BeforeEach → @Test → @AfterEach
  ↓
필요 시 트랜잭션 롤백
```

---

## 🧠 6. ApplicationContext 캐싱 전략

- 같은 설정 클래스를 사용하는 테스트는 **ApplicationContext를 캐싱**하여 성능 최적화
- 단, `@DirtiesContext`가 붙으면 컨텍스트 재생성됨

```
@DirtiesContext
class UserServiceTest {
  // 이 테스트는 끝나고 컨텍스트 폐기됨
}
```

---

## 🧪 7. 리스너 확장 예시

```
@TestExecutionListeners(
  value = CustomTestExecutionListener.class,
  mergeMode = MERGE_WITH_DEFAULTS
)
public class MyTest { ... }
```

→ `TestExecutionListener` 구현체를 등록하여 `beforeTestClass`, `afterTestMethod` 등 제어 가능

---

## 🔁 8. JUnit 5와의 연동

JUnit 5에서는 `@ExtendWith(SpringExtension.class)`가 TestContextFramework를 활성화한다.

```
@ExtendWith(SpringExtension.class)
@SpringBootTest
public class MyTest { ... }
```

→ SpringExtension은 내부에서 TestContextManager를 초기화하고 실행 흐름을 Spring으로 넘긴다.

---

## 📦 9. 다양한 테스트 슬라이스 어노테이션들 (TestContext 기반)

| 어노테이션 | 설명 |
| --- | --- |
| `@WebMvcTest` | 컨트롤러 계층 테스트 (WebContext만 로딩) |
| `@DataJpaTest` | JPA Repository 테스트 |
| `@JdbcTest` | JDBC 기반 테스트 |
| `@RestClientTest` | REST Template 테스트 |
| `@JsonTest` | JSON 직렬화 테스트 |
| `@SpringBootTest` | 전체 애플리케이션 테스트 |

→ 전부 **TestContext Framework 위에서 실행**됨

---

## ✅ 10. 실무 전략

| 전략 | 설명 |
| --- | --- |
| 컨텍스트 재사용으로 속도 확보 | 설정 클래스 또는 Profile 분리로 캐싱 효과 극대화 |
| 필요한 기능만 슬라이스 테스트 | `@WebMvcTest`, `@DataJpaTest` 등 적극 활용 |
| DB 상태 유지하려면 `@Commit` 사용 | 기본은 자동 롤백이므로 주의 |
| 테스트 실행 흐름 제어 가능 | 커스텀 `TestExecutionListener` 등록 |
| 트랜잭션 테스트 실패 시 롤백 확인 | DB 변경 확인 위해 `@Transactional` 테스트 실행 후 검증 |

---

## 📌 요약

| 항목 | 설명 |
| --- | --- |
| 정의 | Spring 테스트 실행을 관리하는 기반 프레임워크 |

| 항목 | 설명 |
| --- | --- |
| 핵심 구성 | TestContextManager, TestExecutionListener |
| 주요 기능 | Context 로딩, DI, 트랜잭션, 캐시, 실행 훅 |
| 어노테이션 | `@SpringBootTest`, `@ContextConfiguration`, `@Transactional` 등 |
| 실무 팁 | 슬라이스 테스트, 캐싱 전략, 커스텀 리스너 활용 |

# `@SpringBootTest`, `@WebMvcTest`, `@DataJpaTest`

`@SpringBootTest`, `@WebMvcTest`, `@DataJpaTest`는 모두 Spring Boot에서 제공하는 **슬라이스 테스트 어노테이션**으로, **테스트의 목적과 범위에 맞게 필요한 Bean만 로딩**하여 성능과 테스트 품질을 향상시켜 준다.

---

## 📘 1. 어노테이션 개요 비교

| 어노테이션 | 목적 | 로딩 대상 Bean 범위 |
| --- | --- | --- |
| `@SpringBootTest` | 전체 애플리케이션 통합 테스트 | **ApplicationContext 전체 로딩** |
| `@WebMvcTest` | 컨트롤러(Web 계층) 단위 테스트 | **컨트롤러 + MVC 구성 요소만 로딩** |
| `@DataJpaTest` | JPA Repository 테스트 | **Entity + Repository + JPA 설정**만 로딩 |

---

## 🧩 2. @SpringBootTest

### ✅ 개요

- 전체 Spring Application Context를 로드하여 **완전한 통합 테스트 수행**
- Service, Repository, Security, Web 등 모든 Bean 사용 가능

### ✅ 예제

```
@SpringBootTest
class UserServiceIntegrationTest {

    @Autowired
    UserService userService;

    @Test
    void saveUser() {
        ...
    }
}
```

### ✅ 특징

- 기본적으로 내장 웹 서버를 시작하지 않음 (서버 테스트는 별도 설정 필요)
- `@Transactional` 사용 시 테스트 후 자동 롤백
- 속도 느림 (전체 컨텍스트 로딩)

---

## 🌐 3. @WebMvcTest

### ✅ 개요

- **Web 계층만 테스트 (Controller, @ControllerAdvice 등)**
- `@Service`, `@Repository` 등은 로딩되지 않음 → `@MockBean` 필요

### ✅ 예제

```
@WebMvcTest(UserController.class)
class UserControllerTest {

    @Autowired
    MockMvc mockMvc;

    @MockBean
    UserService userService;

    @Test
    void testGetUser() throws Exception {
        ...
    }
}
```

### ✅ 특징

- `MockMvc`를 사용하여 컨트롤러 요청을 가상으로 수행
- 컨트롤러와 관련된 **DispatcherServlet, Filter, MessageConverter** 등만 포함
- DB, Service, Component는 직접 주입 불가

---

## 🗄️ 4. @DataJpaTest

### ✅ 개요

- JPA 관련 컴포넌트 (Entity, Repository, EmbeddedDatabase 등)만 로드
- 실제 DB 동작을 빠르게 테스트할 수 있는 **슬라이스 테스트**

### ✅ 예제

```
@DataJpaTest
class UserRepositoryTest {

    @Autowired
    UserRepository userRepository;

    @Test
    void findByUsername() {
        ...
    }
}
```

### ✅ 특징

- 기본적으로 **임베디드 H2 DB**로 동작 (`spring.datasource.url` 오버라이드 가능)
- `@Transactional` 포함 → 테스트 종료 시 자동 롤백
- `@Import` 또는 `@ComponentScan` 없이 Service/Controller는 포함되지 않음

---

## 🧠 5. 비교 요약표

| 항목 | `@SpringBootTest` | `@WebMvcTest` | `@DataJpaTest` |
|---|---|---|---|
| 목적 | 전체 애플리케이션 테스트 | Web 계층 단위 테스트 | JPA 계층 단위 테스트 |
| 로딩 범위 | 전체 Bean | Controller + MVC 관련 | Entity + Repository |
| 테스트 속도 | 느림 | 빠름 | 빠름 |
| 기본 DB 설정 | 실제 DB or H2 | X (DB 없음) | 기본 H2 내장 |
| `@Transactional` 포함 | ❌ 명시해야 함 | ❌ | ✅ 자동 롤백 포함 |
| 내장 웹서버 | ❌ 기본 비활성 | ❌ | ❌ |
| 서비스 주입 | 가능 | 불가 (`@MockBean` 필요) | 불가 |

---

## ⚙️ 6. 실무 전략

| 상황 | 추천 어노테이션 |
|---|---|
| 전체 API 흐름 테스트 | `@SpringBootTest` |
| Controller만 검증 (MockMvc) | `@WebMvcTest` |
| Repository + Entity 테스트 | `@DataJpaTest` |
| Service + Repository 통합 테스트 | `@SpringBootTest + @Transactional` |

| 상황 | 추천 어노테이션 |
| --- | --- |
| Controller + 실제 DB 테스트 | `@SpringBootTest + TestContainers` |

---

## ⚠️ 7. 주의사항

| 항목 | 설명 |
| --- | --- |
| `@WebMvcTest`에는 DB 접근 불가 | Service, Repository 모두 제외됨 |
| `@DataJpaTest`에는 Controller 없음 | 순수 Repository만 테스트해야 함 |
| Mock 사용 필요 시 | `@MockBean`으로 수동 주입 |
| H2 대신 실제 DB 쓰려면 | `application-test.yml`에서 설정 변경 |
| WebMvcTest에서 JSON 변환 실패 | `ObjectMapper`, `MessageConverter` 포함 여부 확인 필요 |

---

## 📌 8. 요약

| 테스트 목적 | 추천 어노테이션 |
| --- | --- |
| 통합 테스트 (전체 흐름) | `@SpringBootTest` |
| 단일 Controller 테스트 | `@WebMvcTest` |
| JPA 단위 테스트 | `@DataJpaTest` |

# Mockito와 `@MockBean`

`Mockito`는 Java에서 가장 널리 쓰이는 **Mocking 프레임워크**이고,
`@MockBean`은 Spring Boot에서 Mockito를 **Spring Context 안에서 사용할 수 있도록 통합한 어노테이션**이다.
둘을 함께 사용하면 **테스트 대상 클래스의 의존성을 가짜(mock) 객체로 대체하고, 그 행위를 검증**할 수 있다.

---

## 📘 1. Mockito란?

- **테스트 대상 클래스의 의존 객체를 가짜(Mock)로 생성**하고
- 이 Mock 객체의 동작을 **사전에 정의**하거나, **호출 여부를 검증**할 수 있도록 지원

### ✅ 기본 용도

- 외부 시스템/API/DB 호출을 대체
- 서비스 로직 단위 테스트에서 Repository 등을 Mock 처리
- 테스트 속도, 독립성, 예측성 향상

---

## 🧩 2. @Mock vs @MockBean 차이

| 항목 | `@Mock` | `@MockBean` |
|---|---|---|
| 대상 | 순수 Mockito 사용 | Spring 테스트 컨텍스트와 함께 사용 |
| 등록 위치 | 테스트 클래스 내부 변수에만 존재 | **Spring ApplicationContext에 등록됨** |
| 동작 범위 | Spring 컨텍스트와 무관 | 실제 Bean을 대체 |
| 사용 가능 위치 | `@ExtendWith(MockitoExtension.class)` 필수 | `@SpringBootTest`, `@WebMvcTest` 등에서 사용 |

---

## ✅ 핵심 차이 요약

| 예시 | 설명 |
|---|---|
| `@Mock UserRepository` | 테스트 객체 안에서만 사용됨 |
| `@MockBean UserRepository` | 실제 애플리케이션의 Repository 빈을 **mock 객체로 교체함** |

---

## 🧪 3. @MockBean 예제 (@SpringBootTest 또는 @WebMvcTest 내부)

```
@SpringBootTest
class UserServiceTest {

    @Autowired
    UserService userService;

    @MockBean
    UserRepository userRepository;

    @Test
    void findUserById_success() {
        // given
        given(userRepository.findById(1L))
            .willReturn(Optional.of(new User(1L, "kim")));

        // when
        User result = userService.findById(1L);

        // then
        assertEquals("kim", result.getName());
        then(userRepository).should(times(1)).findById(1L);
    }
}
```

---

## 🧰 4. Mockito 핵심 메서드

| 메서드/도구 | 설명 |
|---|---|
| `given().willReturn()` | Stub 설정 (Mockito BDD 스타일) |
| `when().thenReturn()` | 일반 스타일 |
| `verify(mock).methodCall()` | 특정 호출이 있었는지 검증 |
| `verify(..., times(n))` | 호출 횟수 검증 |
| `doThrow().when(mock).method()` | 예외 시뮬레이션 |
| `ArgumentCaptor<T>` | 전달된 인자 검증 |

---

## 🔧 5. Spring Boot에서의 `@MockBean` 작동 원리

- 테스트 실행 시 해당 타입의 **실제 Bean을 Spring Context에서 제거**
- 대신 Mockito가 만든 **Mock 객체를 해당 빈 자리에 주입**
- `@Autowired`나 `@InjectMocks`가 있는 클래스에는 **Mock이 자동 주입**

→ `@WebMvcTest`에서는 Service, Repository가 포함되지 않기 때문에 반드시 `@MockBean`으로 등록해야 사용 가능

---

## 🧠 6. 실무 테스트 구조

### ✅ 컨트롤러 테스트 (`@WebMvcTest`)

```
@WebMvcTest(UserController.class)
class UserControllerTest {

    @Autowired MockMvc mockMvc;
    @MockBean UserService userService;

    @Test
    void testGetUser() throws Exception {
        given(userService.findUser(anyLong()))
            .willReturn(new User(1L, "kim"));

        mockMvc.perform(get("/users/1"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.name").value("kim"));
    }
}
```

---

## ⚠️ 7. 주의사항

| 항목 | 설명 |
|---|---|
| `@MockBean`은 반드시 Spring 테스트에서만 작동 | 일반 JUnit 테스트에서는 `@Mock` 사용 |
| 같은 타입의 빈을 여러 개 MockBean으로 등록하면 | 중복 등록 오류 발생 |
| 테스트 간 상태 공유 방지 필요 | Mockito는 static 캐시 사용 안 함 → 안전 |
| `@SpyBean`은 실제 Bean + 일부만 Mock 가능 | 실전 로직 일부만 검증할 때 유용 |

---

## 📌 8. 요약 비교표

| 항목 | `@Mock` | `@MockBean` |
|---|---|---|
| 위치 | Mockito 내부만 | Spring Context 내의 Bean 대체 |
| 테스트 대상 | 일반 Java 테스트 | `@SpringBootTest`, `@WebMvcTest` 등 |
| 주입 방식 | `@InjectMocks`로 수동 주입 | Spring의 DI로 자동 주입 |
| 주 용도 | 서비스 단위 로직 테스트 | Spring 기반 계층 분리 테스트 |

# 테스트 더블

**테스트 더블(Test Double)**은 **단위 테스트에서 실제 의존 객체(real object)를 대체**하여 사용하는 가짜 객체이다.
"스턴트 더블(Stunt Double)"이 실제 배우 대신 위험한 장면을 수행하듯,
**테스트 대상 클래스가 의존하는 객체를 대체**하여 독립적이고 예측 가능한 테스트를 가능하게 한다.

---

## 📘 1. 테스트 더블이란?

> "테스트 대상이 의존하는 객체의 자리를 대신하여 테스트를 보조하는 모든 가짜 객체"

📌 용어 정리:

- "테스트 더블"은 통칭이며, 그 아래 여러 종류가 존재함 (Stub, Mock, Fake, Spy, Dummy)

---

## 🧩 2. 테스트 더블의 5가지 종류 (xUnit 패턴 기준)

| 이름 | 설명 |
|---|---|
| **Dummy** | 사용되지 않지만 메서드 시그니처를 채우기 위한 객체 (`null` 대용) |
| **Fake** | **실제 구현이 있지만 간단한 버전** (ex. InMemory DB) |
| **Stub** | **고정된 응답을 리턴**하는 객체 |
| **Mock** | **호출 여부, 순서, 인자 등을 검증**하는 객체 |

| 이름 | 설명 |
| --- | --- |
| Spy | **실제 객체를 감싸서 동작은 진짜로 하되**, 호출 여부 추적도 가능 |

---

## 🧪 3. 예제 비교 (UserService → UserRepository 의존)

### ✅ 1. Dummy

```
class DummyUserRepository implements UserRepository {
    // 아무 동작도 하지 않음
}
```

→ 단지 `UserRepository` 인터페이스를 만족시키기 위한 객체

---

### ✅ 2. Stub

```
class StubUserRepository implements UserRepository {
    public Optional<User> findById(Long id) {
        return Optional.of(new User(id, "kim"));
    }
}
```

→ 항상 같은 응답을 리턴하여 테스트 결과를 예측 가능하게 만듦

---

### ✅ 3. Fake

```
class FakeUserRepository implements UserRepository {
    private Map<Long, User> store = new HashMap<>();

    public void save(User user) {
        store.put(user.getId(), user);
    }

    public Optional<User> findById(Long id) {
        return Optional.ofNullable(store.get(id));
    }
}
```

→ 실제 DB 없이 작동하는 InMemory 구현 (예: FakeEmailSender, FakePaymentGateway 등)

---

### ✅ 4. Mock (Mockito 사용)

```
@Mock
UserRepository userRepository;

@Test
void shouldCallFindByIdOnce() {
    userService.findById(1L);
    verify(userRepository, times(1)).findById(1L);
}
```

→ 호출 횟수, 순서, 파라미터 등 **행위를 검증**

---

### ✅ 5. Spy (실제 객체 + 추적)

```
@Spy
UserRepository userRepository = new RealUserRepository();

@Test
void shouldCallRealMethodAndVerify() {
    userService.doSomething(); // 실제 저장 수행
    verify(userRepository).save(any());
}
```

→ 진짜 메서드를 호출하되, Mockito로 추적도 가능

---

## 🔧 4. 테스트 더블 선택 기준

| 목적 | 선택 |
|---|---|
| 단순 시그니처 채움 | Dummy |
| 고정된 결과 반환 | Stub |
| 실제 동작하되 단순화된 구조 | Fake |
| 행위(호출 여부, 순서 등) 검증 | Mock |
| 진짜 객체 + 검증 | Spy |

---

## 🧠 5. 실무에서의 적용 전략

| 계층 | 테스트 더블 추천 | 설명 |
|---|---|---|
| Repository (DB) | **Fake**, Stub | InMemory 저장소 자주 사용 |
| 외부 API Client | **Stub**, Mock | 호출 여부, 응답 예측 필요 |

| 계층 | 테스트 더블 추천 | 설명 |
| --- | --- | --- |
| EmailSender 등 IO | **Mock**, Spy | 실제 전송 방지 + 호출 검증 |
| 서비스 내부 | Stub or Mock | 비즈니스 로직 중심 테스트 |
| 테스트가 어려운 클래스 | 인터페이스로 분리 후 Mock 사용 | 테스트 용이성 확보 |

---

## 📌 6. 요약

| 더블 종류 | 목적 | 사용 예시 |
| --- | --- | --- |
| Dummy | 시그니처 채움만 필요 | `null`, 빈 객체 |
| Stub | 고정 응답 리턴 | `given(...).willReturn(...)` |
| Fake | 실제 구현이지만 단순 버전 | InMemoryRepository |
| Mock | 호출/순서/인자 검증 | `verify(...)`, `times(...)` |
| Spy | 진짜 동작 + 검증 | `@Spy`, `doReturn().when(...)` |

# 테스트 DB 설정

Spring Boot에서 **테스트용 데이터베이스 설정**은 매우 중요하다.
단위 테스트나 통합 테스트에서 **운영 DB에 영향을 주지 않고**,
**속도와 격리성, 신뢰성**을 보장하기 위해 테스트 전용 DB를 구성한다.

---

## 📘 1. 테스트 DB의 필요성

| 이유 | 설명 |
| --- | --- |
| 운영 DB 오염 방지 | 테스트로 인한 실데이터 손상 예방 |
| 테스트 격리성 확보 | 테스트마다 독립 실행 가능 |
| 속도 개선 | 인메모리 DB로 빠른 수행 |
| 테스트 재현성 보장 | 상태 초기화 가능 |
| CI/CD 자동화 대응 | 외부 의존성 없는 테스트 가능 |

---

## 📜 2. 기본 설정: `application-test.yml`

```yaml
spring:
  profiles:
    active: test

  datasource:
```

```
    url: jdbc:h2:mem:testdb
    driver-class-name: org.h2.Driver
    username: sa
    password:

  h2:
    console:
      enabled: true

  jpa:
    hibernate:
      ddl-auto: create-drop
    show-sql: true
    properties:
      hibernate:
        format_sql: true
```

→ 테스트 환경에서만 로드되도록 `@ActiveProfiles("test")`를 테스트 클래스에 명시

```
@SpringBootTest
@ActiveProfiles("test")
class UserRepositoryTest { ... }
```

---

## 🧩 3. 인메모리 DB 종류

| DB | 특징 |
|---|---|
| **H2** | Java 기반, 매우 빠르고 Spring 기본값 |
| HSQLDB | H2보다 기능 적지만 대체 가능 |
| Derby | Java 내장 DB, Embedded 모드 지원 |
| SQLite | 파일 기반, Android에 특화 |
| **Testcontainers** | Docker 기반, 실제 DB 사용 가능 |

---

## 🔁 4. 테스트 트랜잭션과 롤백

Spring 테스트는 기본적으로 `@Transactional` + **자동 롤백** 전략 사용

```java
@DataJpaTest  // includes @Transactional
class MyRepositoryTest {

    @Autowired
    MyRepository repo;

    @Test
    void testInsert() {
        repo.save(new MyEntity(...));
        // 테스트 끝나면 DB 롤백됨
    }
}
```

### ✅ 수동 커밋이 필요할 경우

```java
@Commit // 테스트 후 커밋
@Rollback(false) // 롤백 방지
```

---

## 🧪 5. 데이터 초기화 전략

### ✅ 1. SQL 파일 자동 실행

```sql
-- src/test/resources/data.sql
INSERT INTO users (id, name) VALUES (1, 'test');
```

→ `spring.datasource.initialization-mode=always` 또는 Spring Boot 2.5+는 자동 실행

### ✅ 2. `@Sql` 어노테이션

```java
@Sql("/init/test-data.sql")
@Sql(scripts = "/init/cleanup.sql", executionPhase =
Sql.ExecutionPhase.AFTER_TEST_METHOD)
class UserRepositoryTest { ... }
```

---

## 🐬 6. Testcontainers를 활용한 실DB 테스트

**실제 DB (MySQL, PostgreSQL 등)를 Docker로 띄워 테스트**

```java
@Testcontainers
@SpringBootTest
@ActiveProfiles("test")
class IntegrationTest {

    @Container
    static PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>("postgres:14")
        .withDatabaseName("testdb")
        .withUsername("user")
        .withPassword("pass");
```

```
11
12      @DynamicPropertySource
13      static void overrideProps(DynamicPropertyRegistry registry) {
14          registry.add("spring.datasource.url", postgres::getJdbcUrl);
15          registry.add("spring.datasource.username", postgres::getUsername);
16          registry.add("spring.datasource.password", postgres::getPassword);
17      }
18  }
```

→ 운영 환경과 가장 유사한 조건에서 테스트 가능

---

## 📦 7. 테스트 DB 설정 전략 요약

| 항목 | 설정 전략 |
|---|---|
| 빠른 테스트 실행 | H2, HSQL 등 인메모리 사용 |
| 실제 환경과 유사 | Testcontainers로 실DB |
| 상태 초기화 | `@Sql`, `data.sql`, 트랜잭션 롤백 |
| 설정 분리 | `application-test.yml` + `@ActiveProfiles` |
| 독립성 보장 | 테스트마다 DB 초기화 or 트랜잭션 격리 |

---

## 📌 8. 요약

| 항목 | 설명 |
|---|---|
| 테스트용 DB | 기본: H2, 고급: Testcontainers |
| 프로필 분리 | `application-test.yml`, `@ActiveProfiles("test")` |
| 트랜잭션 관리 | `@Transactional` + 자동 롤백 |
| 데이터 초기화 | `data.sql`, `@Sql`, `@BeforeEach` |
| 실무 권장 | 빠른 테스트: H2 / 정확한 검증: Testcontainers |

# 테스트 데이터 삽입 (`@Sql`)

Spring에서 **테스트용 데이터 삽입**을 할 때 `@Sql` 어노테이션은 매우 강력하고 유연한 도구이다.
테스트 메서드나 클래스 실행 전후에 SQL 스크립트를 자동으로 실행하여
**테스트에 필요한 데이터를 미리 준비하거나 정리(cleanup)** 할 수 있다.

---

## 📘 1. `@Sql` 이란?

- **Spring TestContext Framework**에서 제공하는 어노테이션
- 테스트 **실행 전 또는 후에 SQL 스크립트를 실행**할 수 있음
- DB 초기화, 샘플 데이터 삽입, 정리용으로 사용

```
@Sql("/init/data.sql")
```

→ `src/test/resources/init/data.sql` 파일을 테스트 시작 전에 실행

---

## 🧪 2. 기본 예제

### ✅ 테스트 클래스 전체에 적용

```
@Sql("/sql/init.sql")
class UserRepositoryTest {

    @Test
    void testUserExists() {
        ...
    }
}
```

→ 모든 테스트 메서드 실행 전에 `init.sql` 이 실행됨

---

### ✅ 개별 테스트 메서드에 적용

```
@Sql("/sql/init-user.sql")
@Test
void testUser() {
    ...
}
```

---

## 🕰️ 3. 실행 시점 지정

```
@Sql(
  scripts = "/sql/init.sql",
  executionPhase = Sql.ExecutionPhase.BEFORE_TEST_METHOD
)
```

| 옵션 | 설명 |
| --- | --- |
| `BEFORE_TEST_METHOD` (기본값) | 각 테스트 메서드 실행 전에 실행 |
| `AFTER_TEST_METHOD` | 각 테스트 메서드 실행 후 실행 |

| 옵션 | 설명 |
| --- | --- |
| `BEFORE_TEST_CLASS` | 테스트 클래스 시작 전에 1회 실행 |
| `AFTER_TEST_CLASS` | 테스트 클래스 종료 후 1회 실행 |

## 📜 4. 정리 SQL 함께 사용

```
@Sql(scripts = "/sql/init.sql", executionPhase = Sql.ExecutionPhase.BEFORE_TEST_METHOD)
@Sql(scripts = "/sql/cleanup.sql", executionPhase =
Sql.ExecutionPhase.AFTER_TEST_METHOD)
```

→ 테스트 전 초기화, 테스트 후 정리

## 🔁 5. 다중 스크립트 지정

```
@Sql({
    "/sql/init-users.sql",
    "/sql/init-orders.sql"
})
```

→ 순차적으로 실행됨

## 🔒 6. 트랜잭션과의 관계

| 상황 | 설명 |
| --- | --- |
| `@Transactional`과 함께 사용 시 | SQL도 같은 트랜잭션 안에서 실행됨 |
| `@Rollback`이 기본 적용되어 있음 | 테스트 후 SQL도 롤백됨 |
| `@Commit` 추가 시 DB에 반영됨 | 테스트 종료 후 실제 적용 가능 |

## 🧠 7. 실무 팁

| 전략 | 설명 |
| --- | --- |
| 테스트용 데이터는 `test/resources/sql/` 폴더에 관리 | `@Sql("/sql/user-init.sql")` 구조 |
| 클래스 단위로 공통 데이터 구성 | `@Sql(scripts = "common.sql", executionPhase = BEFORE_TEST_CLASS)` |
| 외부에서 DB 초기화 관리 시 | Flyway/Migration 도구와 함께 사용 가능 |

| 전략 | 설명 |
| --- | --- |
| 정리(cleanup.sql)은 주로 `TRUNCATE`, `DELETE` 사용 | 데이터 충돌 방지 |

---

## 🧩 8. 자주 쓰는 SQL 예시

```
-- init.sql
INSERT INTO users (id, name) VALUES (1, 'kim');
INSERT INTO users (id, name) VALUES (2, 'lee');

-- cleanup.sql
DELETE FROM users;
```

---

## ✅ 9. 요약

| 항목 | 설명 |
| --- | --- |
| 목적 | 테스트용 데이터 준비 및 정리 |
| 위치 | `src/test/resources/**/*.sql` |
| 실행 시점 | BEFORE / AFTER (METHOD / CLASS) |
| 트랜잭션 연동 | `@Transactional`에 묶이며 롤백 가능 |
| 여러 스크립트 사용 | `@Sql({ "a.sql", "b.sql" })` 가능 |

# 테스트 자동화

**테스트 자동화(Test Automation)**는 코드 변경이 발생할 때마다 **사람의 개입 없이 테스트를 실행**하고,
그 결과를 자동으로 확인할 수 있도록 만드는 일련의 체계를 의미한다.
Spring Boot 기반의 백엔드 시스템에서 테스트 자동화는 **빌드, 테스트 실행, 리포트, 배포 전 검증**까지 연결되는 핵심 요소이다.

---

## 📘 1. 테스트 자동화란?

> "테스트를 수동으로 일일이 실행하지 않고, 코드 변경 시 자동으로 검증이 수행되도록 하는 일련의 시스템화된 프로세스"

📌 목적:

- 개발자의 실수 방지
- 릴리즈 시점에 신뢰성 확보
- 코드 변경으로 인한 회귀(regression) 방지
- 품질 관리 효율성 향상

---

## 🧩 2. 테스트 자동화의 구성 요소

| 구성 요소 | 역할 |
|---|---|
| **단위 테스트** | 기능 단위 검증 |
| **통합 테스트** | 컴포넌트 간 상호작용 검증 |
| **E2E 테스트** | 실제 사용자 흐름 시뮬레이션 |
| **CI/CD 파이프라인** | 변경 사항 푸시 시 테스트 자동 실행 |
| **테스트 커버리지** | 테스트가 코드를 얼마나 검증하는지 확인 |
| **리포트 시스템** | 실패 시 로그, 성공 시 리포트 자동 출력 |

---

## ⚙️ 3. Spring Boot에서 자동화 환경 구성

### ✅ 테스트 도구 스택

| 범위 | 도구 |
|---|---|
| 단위 테스트 | JUnit 5 + Mockito |
| 통합 테스트 | SpringBootTest + TestContainers |
| REST API 테스트 | MockMvc, RestAssured |
| 커버리지 분석 | JaCoCo |
| 빌드 자동화 | Gradle, Maven |
| CI/CD 자동화 | GitHub Actions, Jenkins, GitLab CI, CircleCI |

---

### ✅ 테스트 실행 명령어

#### Gradle

```
./gradlew test
```

#### Maven

```
mvn test
```

→ `src/test/java/**` 내 모든 테스트 실행

---

## 🧪 4. 자동화 구성 예시: GitHub Actions

```yaml
# .github/workflows/test.yml
name: Test and Build

on:
  push:
    branches: [ main ]
  pull_request:

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up JDK
        uses: actions/setup-java@v3
        with:
          java-version: '17'

      - name: Build and test
        run: ./gradlew clean test

      - name: Upload test report
        uses: actions/upload-artifact@v3
        with:
          name: junit-report
          path: build/test-results/test/
```

---

## 📦 5. 테스트 자동화 + 커버리지 측정 (JaCoCo)

### Gradle 설정

```
plugins {
    id 'jacoco'
}

jacocoTestReport {
    reports {
        xml.required = true
        html.required = true
    }
}
```

### 실행

```
./gradlew test jacocoTestReport
```

→ `build/reports/jacoco/index.html`에서 커버리지 시각화

---

## 🧠 6. 실무 자동화 전략

| 전략 | 설명 |
| --- | --- |
| **단위/통합 테스트 분리 실행** | `@Tag("integration")`, `test filtering` |
| **슬로우 테스트 따로 관리** | `@Disabled`, `@Tag("slow")`로 분리 |
| **PR 시 자동 실행** | GitHub Actions, Jenkins CI로 연결 |
| **실DB 검증은 주기적만 실행** | TestContainers는 무겁기 때문에 선택 실행 |
| **빌드 실패 = 배포 실패** | 테스트 실패 시 배포 파이프라인 종료 처리 |

---

## 📌 7. 요약

| 항목 | 설명 |
| --- | --- |
| 목적 | 테스트 반복 자동화 → 품질 향상 |
| 도구 | JUnit, Mockito, JaCoCo, CI 서버 |
| 실행 방식 | Git 이벤트 기반 (Push, PR 등) |
| 보고서 | 테스트 리포트 + 커버리지 HTML 출력 |
| 실무 적용 전략 | 태깅/분리/캐싱/조건부 실행 조합 |

# 커버리지 측정 도구

- # JaCoCo, SonarQube

  **JaCoCo**와 **SonarQube**는 **테스트 커버리지 측정 및 코드 품질 분석**을 위한 대표적인 도구이다.
  둘은 각자 역할이 다르며, 함께 연동할 경우 **정적 분석 + 테스트 커버리지 + 유지보수성 분석**을 모두 자동화할 수 있다.

  ---

  ## 📘 1. JaCoCo란?

  **Ja**va **Co**de **Co**verage의 약자이며,
  JUnit 테스트 실행 시 **테스트가 실제 코드의 몇 %를 실행했는지를 측정하는 도구**이다.

| 항목 | 설명 |
| --- | --- |
| 타입 | **커버리지 측정 도구** (오픈소스, JVM 기반) |
| 분석 대상 | `.class` 파일 (바이트코드 단위) |
| 통계 지표 | Line, Branch, Instruction, Method, Class 커버리지 |
| 결과 리포트 | XML, HTML, CSV 등 지원 |
| 통합 가능 도구 | Gradle, Maven, SonarQube, Jenkins 등과 연동 가능 |

---

## ✅ JaCoCo 커버리지 종류

| 항목 | 설명 |
| --- | --- |
| Line | 몇 줄의 코드가 실행되었는가 |
| Branch | if/else 등 분기 조건이 테스트되었는가 |
| Method | 메서드가 실행되었는가 |
| Instruction | JVM 명령어 기준 실행 여부 |
| Class | 클래스 단위로 커버되었는가 |

---

## ✅ Gradle 설정 예시

```
plugins {
    id 'jacoco'
}

jacoco {
    toolVersion = "0.8.10" // 최신 버전
}

jacocoTestReport {
    dependsOn test
    reports {
        xml.required = true
        html.required = true
    }
}
```

### 실행 명령어

```
./gradlew test jacocoTestReport
```

### 결과 위치

```
build/reports/jacoco/test/html/index.html
```

---

## 📦 2. SonarQube란?

**코드 품질 관리 플랫폼**으로 다음 기능을 제공함:

| 기능 | 설명 |
|---|---|
| 정적 코드 분석 | 버그, 취약점, 코드 냄새 탐지 |
| 커버리지 리포트 | JaCoCo 등 외부 도구로부터 커버리지 수집 |
| 유지보수성 및 기술부채 계산 | "코드를 얼마나 고쳐야 하는가" 수치화 |
| CI/CD 연동 | GitHub Actions, Jenkins 등 |
| 다양한 언어 지원 | Java, Kotlin, JavaScript, Python 등 |

---

### ✅ 핵심 분석 항목

| 항목 | 설명 |
|---|---|
| **Bugs** | 실제 실행 시 오류를 유발할 수 있는 코드 |
| **Vulnerabilities** | 보안 결함 유발 가능성 |
| **Code Smells** | 유지보수 어려운 비효율적 구조 |
| **Coverage** | 테스트 커버리지 (JaCoCo 연동) |
| **Duplications** | 중복 코드 비율 |
| **Debt** | 개선에 필요한 시간 (기술 부채 지표) |

---

## 🔗 3. JaCoCo + SonarQube 연동

### ✅ Gradle 설정 예시

```
plugins {
    id 'jacoco'
    id 'org.sonarqube' version '4.4.1.3373' // 최신 버전 확인
}

sonar {
    properties {
        property "sonar.projectKey", "myproject"
        property "sonar.organization", "myorg"
```

```
        property "sonar.host.url", "http://localhost:9000"
        property "sonar.login", "your-token"
        property "sonar.coverage.jacoco.xmlReportPaths",
"build/reports/jacoco/test/jacocoTestReport.xml"
    }
}
```

### ✅ 실행 순서

```
./gradlew test jacocoTestReport sonar
```

→ JaCoCo가 커버리지를 수집하고, SonarQube가 분석 및 시각화

---

## 🌐 4. SonarQube 서버 설치

### Docker 방식 (로컬 테스트용)

```
docker run -d --name sonar \
  -p 9000:9000 \
  sonarqube:latest
```

- 접속 주소: http://localhost:9000
- 기본 계정: `admin` / `admin`

---

## 🧠 5. 실무 자동화 전략

| 단계 | 내용 |
|---|---|
| 로컬 테스트 | `JaCoCo`로 커버리지 확인 (`./gradlew test jacocoTestReport`) |
| CI 환경 | GitHub Actions 또는 Jenkins에서 `sonar` 태스크 실행 |
| 품질 게이트 | SonarQube에서 "커버리지 80% 이상" 같은 기준 설정 가능 |
| PR 분석 자동화 | GitHub PR에서 SonarQube 결과 코멘트 남기기 (Developer Edition 이상) |
| 코드 리뷰 연계 | Bug/Vulnerability/Smell 있는 코드 자동 표시 |

---

## ✅ 6. 요약

| 항목 | JaCoCo | SonarQube |
|---|---|---|
| 용도 | 테스트 커버리지 측정 도구 | 코드 품질 종합 분석 플랫폼 |
| 방식 | 바이트코드 실행 분석 | 정적 분석 + 커버리지 리포트 수집 |
| 주요 산출물 | HTML/XML 커버리지 리포트 | 웹 대시보드, 품질 게이트 |

| 항목 | JaCoCo | SonarQube |
| --- | --- | --- |
| 연동 | Gradle/Maven과 연동 | JaCoCo 리포트와 연동 가능 |
| CI 통합 | `./gradlew test jacocoTestReport` | `./gradlew sonar` |