8. 테스트 및 품질 관리

단위 테스트: JUnit 5, Mockito

✅ 1. JUnit 5 개요

항목	설명
표준 테스트 프레임워크	`org.junit.jupiter` 패키지
주요 애노테이션	`@Test`, `@BeforeEach`, `@AfterEach`, `@DisplayName`
실행 방식	메서드 단위 테스트 (Spring 없이도 사용 가능)

✅ 2. Mockito 개요

항목	설명
Mocking 라이브러리	가짜 객체 생성 후 행위 정의 및 검증
사용 목적	실제 의존 객체 없이 단위 테스트 가능
주요 기능	`@Mock`, `@InjectMocks`, `when().thenReturn()`, `verify()`

✅ 3. Gradle 의존성 추가

```
dependencies {
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    testImplementation 'org.mockito:mockito-core'
}
```

> `spring-boot-starter-test` 안에 JUnit 5, Mockito, AssertJ 포함

✅ 4. JUnit 5 기본 테스트 구조

```
class CalculatorTest {

    @BeforeEach
    void setUp() {
        System.out.println("테스트 시작");
    }

    @Test
    @DisplayName("더하기 테스트")
    void testAdd() {
        int result = 2 + 3;
        assertEquals(5, result);
    }

```

```
    @AfterEach
    void tearDown() {
        System.out.println("테스트 종료");
    }
}
```

✅ 5. Mockito 기본 구성

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

    @Mock
    private MemberRepository memberRepository;

    @InjectMocks
    private MemberService memberService;

    @Test
    void 회원_조회_성공() {
        Member mockMember = new Member("kim", "kim@email.com");
        when(memberRepository.findById(1L)).thenReturn(Optional.of(mockMember));

        Member result = memberService.findById(1L);

        assertEquals("kim", result.getName());
        verify(memberRepository).findById(1L); // 메서드 호출 검증
    }
}
```

✅ 6. 주요 애노테이션 정리

애노테이션	설명
`@Test`	테스트 메서드
`@DisplayName`	테스트 이름 커스터마이징
`@BeforeEach` / `@AfterEach`	매 테스트 전/후 실행
`@ExtendWith(MockitoExtension.class)`	Mockito와 JUnit5 연동
`@Mock`	가짜 객체 생성
`@InjectMocks`	테스트 대상 객체에 mock 주입

✅ 7. Mock 행위 정의 및 검증

1) 반환값 설정

```
when(mockObj.method()).thenReturn(value);
```

2) 예외 던지기

```
when(service.doSomething()).thenThrow(new IllegalArgumentException());
```

3) 메서드 호출 검증

```
verify(mockObj).save(any());
verify(mockObj, times(2)).deleteById(anyLong());
```

✅ 8. ArgumentCaptor 사용

```
ArgumentCaptor<Member> captor = ArgumentCaptor.forClass(Member.class);

verify(memberRepository).save(captor.capture());

Member saved = captor.getValue();
assertEquals("kim", saved.getName());
```

✅ 9. BDD 스타일 (권장)

```
given(memberRepository.findById(1L)).willReturn(Optional.of(member));
then(memberRepository).should().findById(1L);
```

→ `MockitoExtension` 사용 시 BDDMockito 지원

✅ 10. 실전 단위 테스트 예제

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

    @Mock private OrderRepository orderRepository;
    @Mock private ProductService productService;

    @InjectMocks private OrderService orderService;

    @Test
    void 주문_성공_테스트() {
        // given
        Product product = new Product("상품", 1000);
        given(productService.getById(1L)).willReturn(product);
```

```
14
15          // when
16          orderService.order(1L, 3); // 3개 주문
17
18          // then
19          then(orderRepository).should().save(any(Order.class));
20      }
21  }
```

✅ 결론 요약

항목	설명
단위 테스트	순수 로직만 검증, 외부 연동 없음
JUnit 5	표준 테스트 프레임워크
Mockito	외부 의존 객체를 가짜(mock)로 대체
핵심 구성	`@Mock`, `@InjectMocks`, `when`, `verify`
실무 권장	JUnit + Mockito + BDD 스타일 (given-when-then)

통합 테스트: `@SpringBootTest`

`@SpringBootTest`는 Spring Boot에서 제공하는 가장 강력한 테스트 도구로,
전체 애플리케이션 컨텍스트(ApplicationContext)를 로딩하여 실제 환경에 가까운 통합 테스트를 수행합니다.

> 단위 테스트가 `비즈니스 로직만 독립적으로 검증`한다면,
> 통합 테스트는 **스프링 구성, DI, DB, API 흐름, 트랜잭션, 보안**까지 **전체를 함께 테스트**합니다.

✅ 1. `@SpringBootTest`란?

항목	설명
정의	Spring Boot 전체 구성 요소를 포함한 통합 테스트
기능	컨트롤러, 서비스, 리포지토리, DB, 보안 등 전부 포함
내부 동작	`SpringApplication.run()` 방식으로 애플리케이션 전체 구동
테스트 범위	실제 실행과 가장 유사한 환경

✅ 2. 기본 예제

```
@SpringBootTest
class MemberServiceIntegrationTest {

    @Autowired
    private MemberService memberService;

    @Autowired
    private MemberRepository memberRepository;

    @Test
    void 회원가입_통합_테스트() {
        Member member = new Member("kim", "kim@example.com");

        memberService.register(member);

        Member saved = memberRepository.findByEmail("kim@example.com").orElseThrow();
        assertEquals("kim", saved.getName());
    }
}
```

✅ 3. 설정 옵션

속성	설명
`webEnvironment`	테스트 환경 설정 (기본은 `MOCK`)

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
```

값	설명
`MOCK`	서블릿 컨테이너 없이 MockMvc 사용 (기본값)
`RANDOM_PORT`	임의의 포트로 내장 톰캣 실행 → `TestRestTemplate` 사용
`DEFINED_PORT`	`application.yml`에 지정된 포트로 구동
`NONE`	웹 환경 비활성화 (일반 Bean만 테스트 시 사용)

✅ 4. 트랜잭션 롤백 설정

```
@SpringBootTest
@Transactional
class UserIntegrationTest {
    // 테스트 후 자동 롤백됨 (DB 초기화 유지)
}
```

- `@Transactional`을 테스트 클래스나 메서드에 붙이면
  각 테스트가 끝난 후 DB에 **자동으로 롤백 처리됨**
- 실무에서는 **DB 상태를 유지하지 않도록 기본 설정으로 사용함**

✅ 5. 테스트 전용 설정 파일 사용

```yaml
# src/test/resources/application-test.yml
spring:
  datasource:
    url: jdbc:h2:mem:testdb
  jpa:
    hibernate:
      ddl-auto: create-drop
```

```java
@SpringBootTest
@TestPropertySource(locations = "classpath:application-test.yml")
```

✅ 6. 테스트 시 전용 Profile 사용

```java
@SpringBootTest
@ActiveProfiles("test")
```

→ `application-test.yml`을 자동으로 인식하고 사용

✅ 7. 테스트 Rest API 호출 (`RANDOM_PORT`)

```java
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
class ApiIntegrationTest {

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    void 사용자_조회_API() {
        ResponseEntity<UserDto> response = restTemplate.getForEntity("/api/users/1",
UserDto.class);
        assertEquals(HttpStatus.OK, response.getStatusCode());
    }
}
```

✅ 8. 실무 활용 전략

목적	방법
전체 구성 통합 테스트	`@SpringBootTest + @Transactional`
API 레벨 테스트	`@SpringBootTest(webEnvironment = RANDOM_PORT)` + `TestRestTemplate`
컨트롤러만 테스트	`@WebMvcTest(Controller.class)`
DB 연동 테스트	`@DataJpaTest`
보안까지 포함한 전체 테스트	`@SpringBootTest + @WithMockUser`

✅ 9. 주의사항

항목	설명
속도	느림. 실제 앱 전체 컨텍스트를 로딩함
비용	단위 테스트보다 리소스 많이 소모
분리 권장	단위 테스트와 통합 테스트는 별도 패키지/구성으로 관리 권장

✅ 10. 결론 요약

항목	설명
핵심 목적	Spring 전체 구성 + Bean DI + DB + Security 등 **통합 테스트** 수행
Bean 주입 방식	`@Autowired` 사용 가능
트랜잭션 롤백	`@Transactional`로 자동 처리
속도	느리지만 가장 신뢰도 높은 테스트
추천 사용	복잡한 서비스 로직, API 흐름 테스트, 보안 검증 등

MockMvc, WebTestClient

Spring에서 API 테스트를 수행할 때, 실제 웹 서버를 구동하지 않고도 컨트롤러 계층을 테스트할 수 있는 강력한 도구가 있습니다.
대표적으로 `MockMvc` **(MVC 테스트)** 와 `WebTestClient` **(Reactive & WebFlux 또는 WebEnvironment 사용)** 가 있으며,
이들은 **테스트 목적, 구조, 환경 구성 방식이 다릅니다.**

✅ 1. MockMvc란?

> Spring MVC 기반의 API 테스트 도구로,
> **내장 톰캣을 구동하지 않고 DispatcherServlet을 모의로 실행**시켜 컨트롤러를 테스트한다.

- 주로 `@WebMvcTest`와 함께 사용
- 필터, 컨트롤러, 인터셉터, ArgumentResolver 등을 테스트 가능
- `MockHttpServletRequest` 기반으로 수행

🔹 MockMvc 기본 예시

```
@WebMvcTest(controllers = MemberController.class)
class MemberControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private MemberService memberService;

    @Test
    void 회원조회_성공() throws Exception {
        given(memberService.findById(1L)).willReturn(new MemberDto("kim",
"kim@example.com"));

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

🔹 MockMvc 특징 정리

항목	설명
실행 방식	Servlet 환경, DispatcherServlet 모의 실행
테스트 대상	Controller + Filter + ArgumentResolver
@WebMvcTest	컨트롤러 계층 테스트에 최적화
실제 톰캣 실행 여부	❌ 없음
장점	빠르고 가벼움, 필터와 인터셉터까지 검증 가능
단점	완전한 HTTP 환경은 아님, WebClient 테스트 불가

✅ 2. WebTestClient란?

> **Spring WebFlux용으로 개발된 비동기/논블로킹 테스트 클라이언트**
> 하지만 **Servlet 기반 Spring MVC 환경에서도 사용 가능** (Spring Boot 2.2 이상)

- **실제 HTTP 요청 방식에 가깝게 작동**
- `WebEnvironment.RANDOM_PORT` 또는 `WebFlux` 기반 컨텍스트 필요
- API 테스트, 응답 본문, 헤더, 쿠키까지 체계적 테스트 가능

🔹 WebTestClient 기본 예시

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@AutoConfigureWebTestClient
class MemberApiTest {

    @Autowired
    private WebTestClient webTestClient;

    @Test
    void 회원조회_API_테스트() {
        webTestClient.get().uri("/api/members/1")
            .exchange()
            .expectStatus().isOk()
            .expectBody()
            .jsonPath("$.name").isEqualTo("kim");
    }
}
```

🔹 WebTestClient 특징 정리

항목	설명
실행 방식	HTTP 요청 시뮬레이션, Netty 또는 톰캣
테스트 대상	전체 API 흐름 (Controller + Filter + RestTemplate + 보안 등)
@SpringBootTest	`WebEnvironment.RANDOM_PORT` 필수
실제 톰캣 실행 여부	✅ 있음 (`TestRestTemplate`과 유사)
장점	완전한 HTTP 흐름 테스트 가능
단점	MockMvc보다 느림, 의존성 더 큼

✅ 3. MockMvc vs WebTestClient 비교표

항목	MockMvc	WebTestClient
대상	Spring MVC 전용	WebFlux + MVC
HTTP 처리	Servlet Mock 처리	실제 HTTP 흐름에 가까움
요청 전송	`mockMvc.perform(...)`	`webTestClient.get().uri(...).exchange()`
응답 검증	`andExpect(...)`	`expectStatus()`, `expectBody()`
실행 속도	빠름 (내장 톰캣 X)	다소 느림 (톰캣 or Netty 실행)
API 테스트 정확성	비교적 제한적	실운영에 가깝게 테스트
보안 필터 테스트	가능	가능
추천 사용 시점	컨트롤러 계층 단위 테스트	통합 테스트 or WebClient 기반

✅ 4. 함께 쓰면 좋은 어노테이션

어노테이션	설명
`@WebMvcTest`	MockMvc 전용, 컨트롤러 계층만 로딩
`@SpringBootTest`	WebTestClient, 전체 애플리케이션 로딩
`@AutoConfigureMockMvc`	`MockMvc` 자동 설정 (SpringBootTest에서도 사용 가능)
`@AutoConfigureWebTestClient`	`WebTestClient` 자동 주입

✅ 결론

전략	설명
MockMvc	빠르고 컨트롤러 위주 테스트에 최적화. `@WebMvcTest`와 조합
WebTestClient	실제 HTTP 흐름을 검증하고 싶을 때. `@SpringBootTest(webEnvironment = RANDOM_PORT)`에서 사용
실무에서는?	MockMvc로 단위/계층 테스트 → WebTestClient로 API 통합 테스트 분리

테스트 슬라이스 (`@WebMvcTest`, `@DataJpaTest`)

Spring Boot 테스트 슬라이스(Slice Test)는 애플리케이션의 **특정 계층만 테스트하기 위해 최소한의 Bean만 로딩**하는 방식입니다.
이를 통해 **빠르고 경량화된 단위 또는 계층별 테스트**가 가능하며,
`@SpringBootTest`보다 훨씬 빠른 테스트를 작성할 수 있습니다.

대표적인 슬라이스 테스트 애노테이션으로는 `@WebMvcTest`, `@DataJpaTest`, `@JsonTest`, `@MockBean`, `@RestClientTest` 등이 있습니다.

✅ 1. 테스트 슬라이스란?

개념	설명
목적	전체 컨텍스트를 로딩하지 않고 **일부 계층만 테스트**
이점	**테스트 속도 빠름, 구성 간결, 의존성 문제 조기 발견 가능**
방법	Spring이 계층별 Bean만 로딩하고 나머지는 제외
구성	테스트 대상 계층 + `@MockBean`으로 외부 의존성 대체

✅ 2. `@WebMvcTest`

📌 목적

- **Controller 계층만 테스트**
- `@Controller`, `@RestController` 관련 Bean만 로딩
- `Service`, `Repository`는 로딩되지 않음 → `@MockBean`으로 주입 필요

📌 예시

```
@WebMvcTest(controllers = MemberController.class)
class MemberControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private MemberService memberService;

    @Test
    void 회원조회_성공() throws Exception {
        given(memberService.findById(1L)).willReturn(new MemberDto("kim",
"kim@ex.com"));

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

📌 특징

항목	내용
포함 대상	`@Controller`, `@ControllerAdvice`, `@JsonComponent` 등
제외 대상	`@Service`, `@Repository`, `@Component` 등
실무 활용	REST API 테스트, 유효성 검증, URL 매핑 검증 등

✅ 3. `@DataJpaTest`

📌 목적

- **JPA Repository 및 DB 연동 테스트**
- Spring Data JPA 관련 Bean만 로딩 (`@Entity`, `JpaRepository`, `EntityManager` 등)
- 기본적으로 **내장 H2 DB** 사용 (`spring-boot-starter-data-jpa` 필요)

📌 예시

```
@DataJpaTest
class MemberRepositoryTest {

    @Autowired
    private MemberRepository memberRepository;

    @Test
    void 이메일로_회원_조회() {
        Member member = new Member("kim", "kim@email.com");
        memberRepository.save(member);

        Optional<Member> result = memberRepository.findByEmail("kim@email.com");

        assertTrue(result.isPresent());
        assertEquals("kim", result.get().getName());
    }
}
```

📌 특징

항목	내용
포함 대상	`@Entity`, `Repository`, `JdbcTemplate`, `TestEntityManager` 등
기본 설정	`H2`, `spring.jpa.hibernate.ddl-auto=create`, 트랜잭션 자동 롤백
설정 파일	`application-test.yml`, 또는 `@TestPropertySource` 사용 가능

✅ 4. 기타 테스트 슬라이스

애노테이션	설명
`@JsonTest`	Jackson, Gson 기반 JSON 직렬화/역직렬화 테스트
`@RestClientTest`	`RestTemplate`, `WebClient` 등 외부 HTTP 클라이언트 테스트
`@JdbcTest`	JDBC 기반 SQL 쿼리 테스트
`@DataMongoTest`	MongoDB Repository 테스트
`@WebFluxTest`	WebFlux 기반 Reactive Controller 테스트

✅ 5. 슬라이스 테스트 vs 통합 테스트

항목	슬라이스 테스트 (`@WebMvcTest`, `@DataJpaTest`)	통합 테스트 (`@SpringBootTest`)
대상	특정 계층만 로딩	전체 ApplicationContext
속도	빠름 (초 단위)	느림 (수 초~수 십 초)
테스트 목적	계층 기능 검증	기능 전체 연동 테스트
Bean 주입	명시적 `@MockBean` 필요	자동 주입 (`@Autowired`) 가능
DB 연동	필요에 따라 포함	포함 가능 (`RANDOM_PORT` 등 설정)

✅ 6. 실무 권장 전략

목적	권장 테스트
컨트롤러 기능 테스트	`@WebMvcTest + MockMvc`
비즈니스 로직 테스트	JUnit5 + Mockito 단위 테스트
Repository 쿼리 검증	`@DataJpaTest`
전체 API 연동	`@SpringBootTest + WebTestClient`
REST 외부 연동	`@RestClientTest + WireMock`

✅ 결론 요약

항목	설명
테스트 슬라이스	특정 계층만 테스트하여 빠른 피드백 제공
`@WebMvcTest`	컨트롤러 테스트에 최적화 (서비스는 Mock)

항목	설명
`@DataJpaTest`	DB 쿼리/엔티티 테스트에 최적화 (트랜잭션 롤백 포함)
테스트 전략	단위 테스트 → 슬라이스 테스트 → 통합 테스트 순으로 계층별 분리 권장

TestContainers 도입

Testcontainers는 테스트 실행 시 실제 외부 의존 환경(예: DB, Redis, Kafka 등)을 **Docker 컨테이너로 자동 실행하여 테스트 환경을 구성**하는 강력한 도구입니다.
이를 통해 테스트는 더 이상 "목업"이 아니라 **진짜와 가장 가까운 환경에서 검증**할 수 있습니다.

Spring Boot + JUnit5 + Testcontainers를 함께 사용하면
개발, QA, CI 환경에서도 일관된 통합 테스트를 구현할 수 있습니다.

✅ 1. Testcontainers란?

항목	설명
목적	테스트 실행 중 Docker 컨테이너로 DB, 서비스 환경 자동 구성
기술	JUnit + Docker 컨테이너 연동
장점	실제 환경과 거의 동일한 조건에서 테스트 수행 가능
대상	PostgreSQL, MySQL, Redis, MongoDB, Kafka, Elasticsearch 등

✅ 2. 사용 전 준비

- Docker가 로컬 혹은 CI/CD 서버에 설치되어 있어야 함
- JDK 11 이상 권장
- `testImplementation`으로만 의존성 추가하면 됨 (빌드 시 실행되지 않음)

✅ 3. Gradle 의존성 추가 (Spring Boot + JUnit5)

```
testImplementation 'org.testcontainers:junit-jupiter:1.19.3'
testImplementation 'org.testcontainers:mysql:1.19.3' // 사용할 DB에 맞게
```

> 버전은 현재 최신인 1.19.x 권장

✅ 4. MySQL 기반 테스트 예시

🔹 1) 컨테이너 정의

```
@Testcontainers
@SpringBootTest
@TestPropertySource(properties = {
    "spring.datasource.url=jdbc:mysql://localhost:3306/test", // 임시값 (실제는 동적 주입)
    "spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver",
    "spring.jpa.hibernate.ddl-auto=create-drop"
})
class MemberRepositoryTest {

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

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

    @Autowired
    private MemberRepository memberRepository;

    @Test
    void 회원_저장_테스트() {
        Member m = new Member("kim", "kim@test.com");
        memberRepository.save(m);

        Member result = memberRepository.findByEmail("kim@test.com").orElseThrow();
        assertEquals("kim", result.getName());
    }
}
```

✅ 5. 핵심 애노테이션 설명

애노테이션	설명
`@Testcontainers`	테스트 클래스에서 컨테이너 사용 선언
`@Container`	테스트 중 사용할 Docker 컨테이너 정의
`@DynamicPropertySource`	컨테이너 실행 결과를 Spring 환경 변수로 주입
`@SpringBootTest`	Spring 전체 컨텍스트 로딩

애노테이션	설명
`@TestPropertySource`	Spring 설정 오버라이드 (기본값 방지)

✅ 6. 컨테이너 재사용/속도 개선 전략

- `@Container`를 `static`으로 선언하여 테스트 클래스 간 재사용
- Gradle 캐시 활용 (`testcontainers.reuse.enable=true`)
- 로컬 테스트 시 `~/.testcontainers.properties`에 다음 설정 추가:

```
testcontainers.reuse.enable=true
```

- CI 환경에서는 Docker Layer Caching(DLC) 적용 권장

✅ 7. 다양한 컨테이너 지원

컨테이너 종류	라이브러리
PostgreSQL	`org.testcontainers:postgresql`
Redis	`org.testcontainers:redis`
MongoDB	`org.testcontainers:mongodb`
Kafka	`org.testcontainers:kafka`
Elasticsearch	`org.testcontainers:elasticsearch`
MinIO / Localstack	AWS 모의 환경

✅ 8. 장점 요약

항목	설명
신뢰성	실제 환경과 유사한 조건에서 테스트 가능
독립성	외부 의존 없이 실행 가능 (CI/CD 포함)
유지보수	DB 초기화, 포트 충돌 없이 자동 격리
확장성	여러 컨테이너 조합 가능 (DB + Redis + Kafka 등)

✅ 9. 실무 적용 전략

용도	추천 도구
Repository 테스트	`@DataJpaTest + Testcontainers`
통합 테스트	`@SpringBootTest + Testcontainers`
DB 마이그레이션 검증	`Flyway + Testcontainers`
API + Kafka 연동 테스트	`KafkaContainer` 연동
CI/CD 테스트	GitHub Actions, GitLab CI에 Docker 지원 필요

✅ 10. 결론

항목	설명
목적	Docker를 통해 **실제 환경 기반 테스트** 자동 구성
장점	신뢰성 + 격리성 + 무상태성
핵심 구성	`@Testcontainers`, `@Container`, `@DynamicPropertySource`
실무 효과	단위 테스트 수준에서 통합 수준까지 검증 가능
추천 사용처	Repository, DB, 외부 시스템 연동 테스트 전반