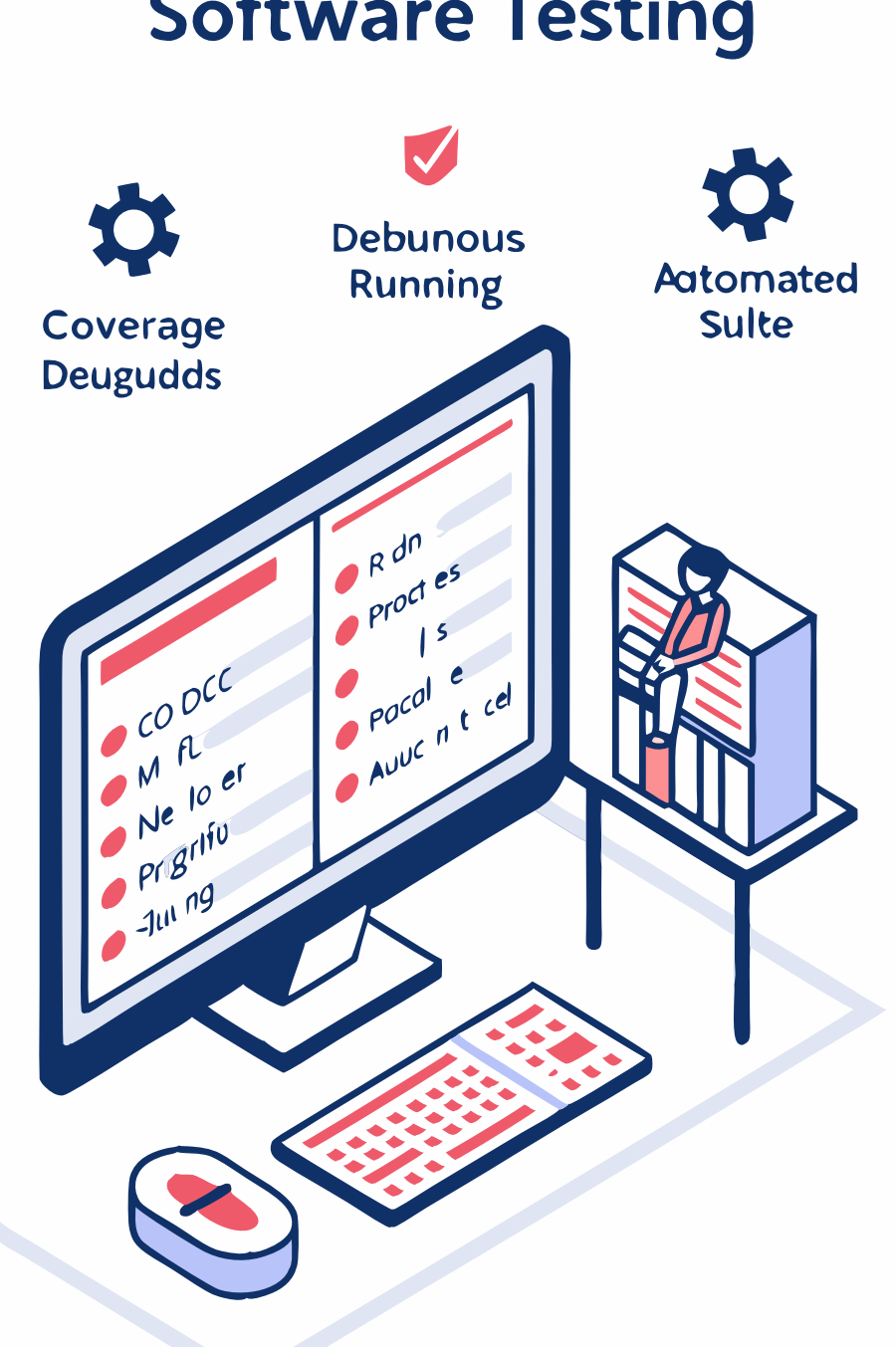
실용주의 소프트웨어 테스트: 이론과 실무의 차이

비즈니스 가치 기반의 실용적 테스트 접근법

발표자: 이태훈



NERD_CON

테스트란 무엇인가?

일반적 정의

소프트웨어가 의도한 대로 기능하는지 체계적으로 검증하는 과정으로써, **요구사항 충족 여부를 확인**하고 **예상치 못한 동작이나 결함을 사전에 발견**하여 **소프트웨어의 신뢰성과 품질을 보장하는 활동**

테스트가 필요한 이유

과거 오류에서 배움

같은 버그 반복 방지

미래의 나 or 동료를 위한 알리바이

코드 변경 시 안전망 역할

3

신뢰할 수 있는 시스템 구축

프로덕션 환경 안정성 확보

로직의 요구사항 명세

테스트가 프로덕션 코드의 의도와 기능을 명세화

커버리지 vs 생존 테스트

커버리지 중심 테스트

수치 목표 달성에 집중

의미 없는 테스트 양산 위험

실제 품질과 연관성 낮음

생존 중심 테스트

핵심 기능 안정성 확보

실제 사용자 시나리오 검증

장애 예방에 효과적

생존과 안정성을 위한 효과적인 테스트 작성시 주의 사항

1 변경 시 오류를 확인할 방법이 없는 경우 리스크

2 생존을 위한 전략

테스트는 개발자의 현실적이고 실질적인 생존 보험

3 단순 커버리지 환상

80% 이상 커버리지도 중요 예외 상황을 놓치면 무용지물

4 핵심 로직에 집중

60% 커버리지라도 핵심과 리스크 포인트를 테스트하면 안정적

NERD_CON

NERD_CON

테스트 작성 원칙

F.I.R.S.T 원칙

Fast
유닛 테스트는 빨라야 함

Isolated
테스트는 고립적이어야 함 (다른 테스트와 종속적이지 않아야 함)

Repeatable
반복 가능해야 함

Self-validating
스스로 검증 가능해야함

Timely
적시에 수행 가능해야 함

Right-BICEP: 테스트 대상과, 올바른 결과 검증을 위한 접근법

Right
결과가 올바른가?

Boundary
경계 조건은 맞는가?

Inverse Relationships
역 관계를 검사할 수 있는가?

Cross check
다른 수단을 활용하여 교차 검사할 수 있는가?

Error conditions
오류 조건을 강제로 일어나게 할 수 있는가?

Performance
성능 조건은 기준에 부합하는가?

NERD_CON

효과적인 테스트 작성 원칙

비즈니스 정책 검증

도메인 규칙 준수 여부 확인

실패 흐름 및 예외 처리 검증

예외 상황에서의 동작과 오류 처리 정확성 확인

테스트의 의도와 역할 명시

테스트는 로직의 의도, 성공/실패 원인이 명확하게 드러나야 함

상태 전이 검증

객체 상태 변화의 정확성 확인

효과적인 테스트 작성 원칙

1. 테스트 깨짐의 중요성

테스트가 실패하도록 유도하기

- **부정 테스트(Negative Test) 활용**
  - 실패상황일때 어떻게 처리되는지 검증
    - 예외 발생 / 비즈니스 로직 흐름 이후 객체가 기대하지 않은 상태로 변경 / 롤백

효과적인 테스트 작성 원칙

2. 테스트코드의 과도한 추상화 방지

도메인도 복잡한데 테스트도 여러 레이어로 나뉘어 있다면 → 코드를 읽는 사람도, 테스트를 고치는 사람도 고통

- 테스트 자체가 테스트 대상보다 **더 이해하기 어려운 코드**가 되어버리는 경우 존재
- **필요에 의한 공통화는 좋지만, 검증 의도는 흐려지면 안 됨**
- **입력과 출력이 명확하게 보이는 상태**여야 함

> ⚠️ **테스트를 위한 추상화가 실제 검증 목적을 가리면 안 됨**

복잡한 추상화 레이어

이해하기 어려운 코드 구조

구현과 테스트 괴리

실제 코드와 동떨어진 테스트

유지보수 비용 증가

변경 시 여러 레이어 수정 필요

디버깅 어려움

문제 원인 추적 복잡

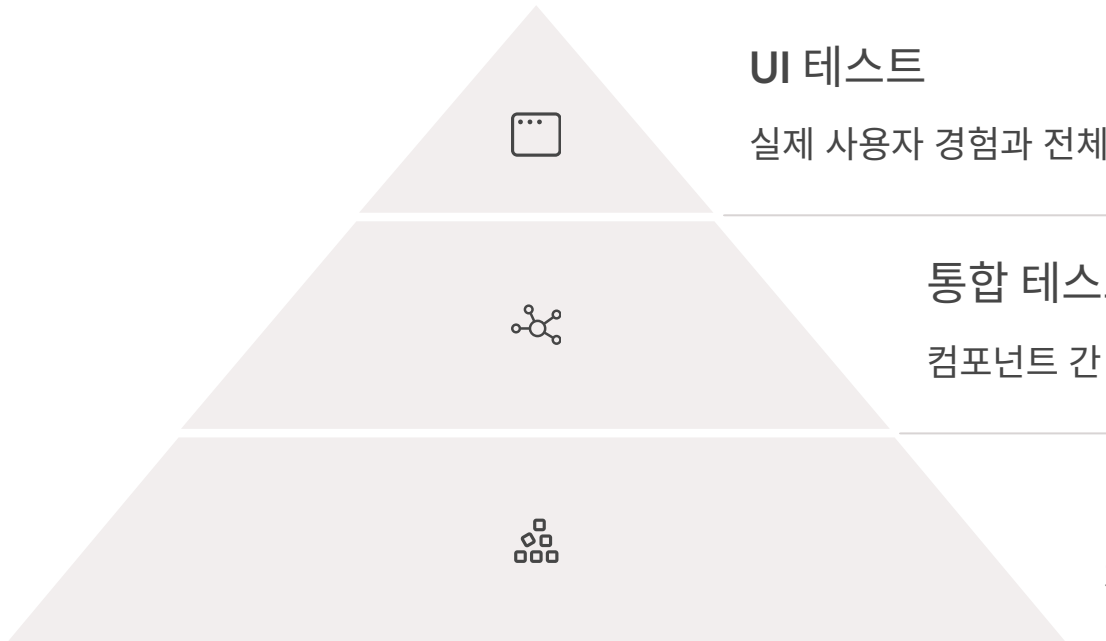
효과적인 테스트 작성 원칙

3. 테스트용 코드가 비즈니스 로직 복사본이 되지 않게 할 것

- Mock, Stub, Builder가 도메인 로직을 그대로 복제하면 실제 코드 변경에도 테스트 검증에 녹아들지 않음
  - **통과용 허수아비:** 테스트가 항상 통과하지만 실제 결함은 찾지 못하는 상황 발생
- 좋은 테스트는 로직을 '설명'하되, '다시 구현'하지 않음
  - 테스트를 위한 복제된 로직은 시간이 지날수록 실제 코드와 괴리가 발생해 테스트의 신뢰성을 상실
- TestFixture를 활용하여 단순 복사가 아닌, 도메인 비즈니스 로직의 흐름을 나타낼수 있는 테스트 데이터셋 중복을 최소화하는 전략 활용
  - 표준화된 테스트 데이터 세트를 구성하여 로직 복제 없이 일관된 테스트 객체 제공
  - 객체 생성과 설정을 분리함으로써 테스트 의도에 집중 가능

테스트 피라미드

테스트는 효율성과 비용에 따라 계층화 필요



UI 테스트

실제 사용자 경험과 전체 시스템 흐름 검증

통합 테스트

컴포넌트 간 상호작용 검증

단위 테스트

개별 기능의 **비즈니스 정책, 상태 전이 검증**

기반이 되는 최하위층에 가까울수록 테스트 작성 비용이 낮고 실행 속도가 빠름

상위 레이어로 갈수록 테스트 비용은 증가하지만 시스템 전체 신뢰도는 높아짐

단위 테스트 기본

1

독립된 작은 단위 검증

단일 메소드/클래스 동작 확인

빠른 실행 속도

별도의 환경 구성 없이, 밀리초 단위 피드백

반복 가능성

언제든 동일한 결과 보장

도메인 중심 단위 테스트

도메인 객체 선택

핵심 비즈니스 객체 식별

테스트 케이스 작성

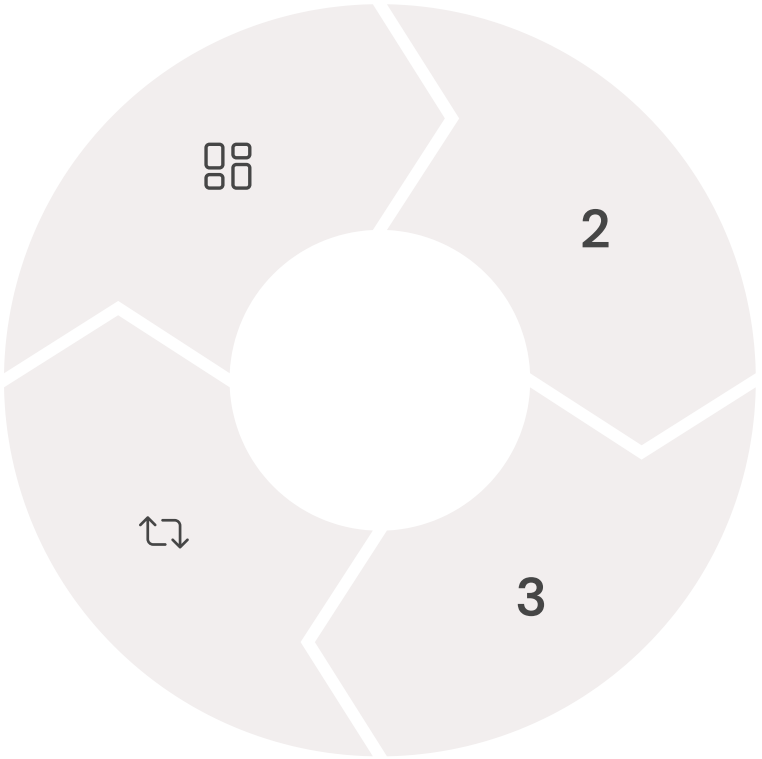
정상/예외 케이스 구분

결과 검증

상태/동작 변화 확인

리팩토링

테스트 코드 개선



단위 테스트는 객체 설계가 잘 되어있을때 효용이 드러난다

단일 책임 원칙 준수

객체가 변경되어야 할 이유가 한가지 일 때, 테스트가 간결해지고 명확한 검증이 가능

느슨한 결합도

객체 간 의존성이 적을수록 독립적인 테스트가 용이하고, 테스트 더블 활용도 Up

높은 응집도

관련 기능이 함께 모여있어 테스트 케이스 작성과 유지보수가 효율적으로 이루어짐

리팩토링 안전성

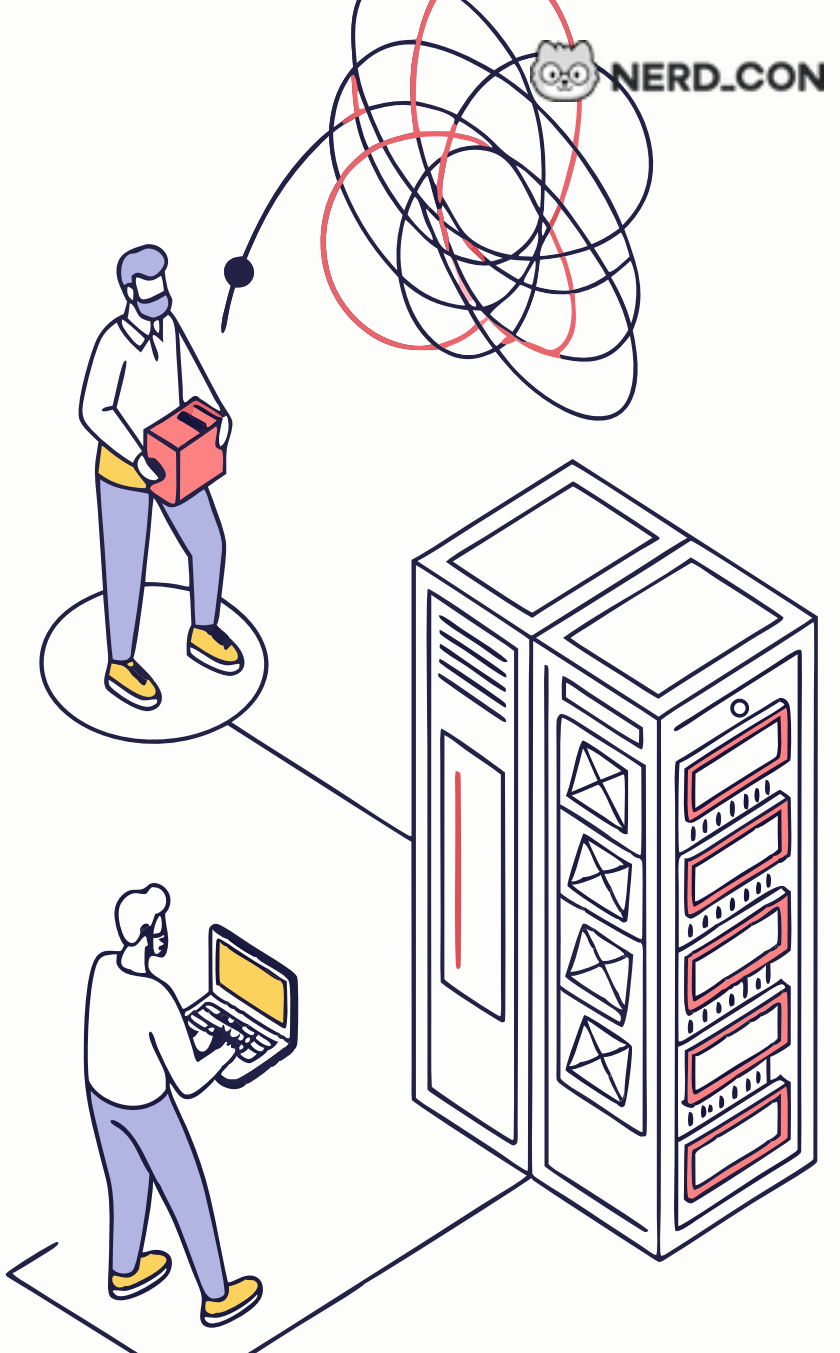
잘 설계된 객체는 내부 구현 변경 시에도 테스트가 깨지지 않는 안정적인 인터페이스 제공

좋은 객체 설계는 테스트 작성을 단순화할 뿐만 아니라, 테스트의 신뢰성과 유지보수성을 크게 향상시킵니다. 반대로 단위 테스트를 작성하는 과정 자체가 객체 설계의 문제점을 드러내는 지표가 되기도 합니다.

그럼 현실적으로 어떻게?

현실에서 마주하는 복잡한 코드는 리팩토링의 가장 큰 장애물, 이상적인 객체 설계와 달리, 실무에서는 얽히고설킨 의존성과 기술 부채를 마주하게 됩니다.

실천 가능한 전략으로 점진적 개선 필요



단위테스트를 위한 현실적 객체 설계 전략

“완벽한 도메인 중심 모델로의 전환”은 현실적으로 불가능 (또한 도메인 중심 아키텍처가 무조건적으로 옳은 구조가 아님, 비즈니스와 시스템 성격마다의 아키텍처 패턴은 모두 다르다)

도메인 중심 테스트가 가능한 지점까지 구조화해보는것을 목표로 함

신규 기능은 도메인 객체부터 설계

→ 레거시와 섞이는 객체이지만, 다른 객체의 영향 없이 일단 한 줄이라도 "testable"한 구조로 만들수 있다.

트랜잭션 스크립트 스타일은 외부 의존성만 분리해도 구조가 명확해진다

→ Test Double(Mock)이 줄어들고 테스트 유지보수가 쉬워짐

비즈니스 정책이 많은 영역(정산, 쿠폰, 할인 등)을 우선적으로 도메인화

→ 테스트 필요성이 크고 가치도 큼

과감히 포기

→ 테스트 작성이 너무 어려우면 과감히 포기

(소프트웨어는 어차피 영원할 수 없음, 그렇지만 유지보수를 잘하기 위한 노력은 필요)

- Test Double / Fixture가 과하게 필요함, assert로 표현하기 어려운 조건부 로직들..

단위 테스트의 한계

1 개별 기능만 검증

객체 상호작용시 워크플로우 검증 불가

2 인프라 상호작용 검증 어려움

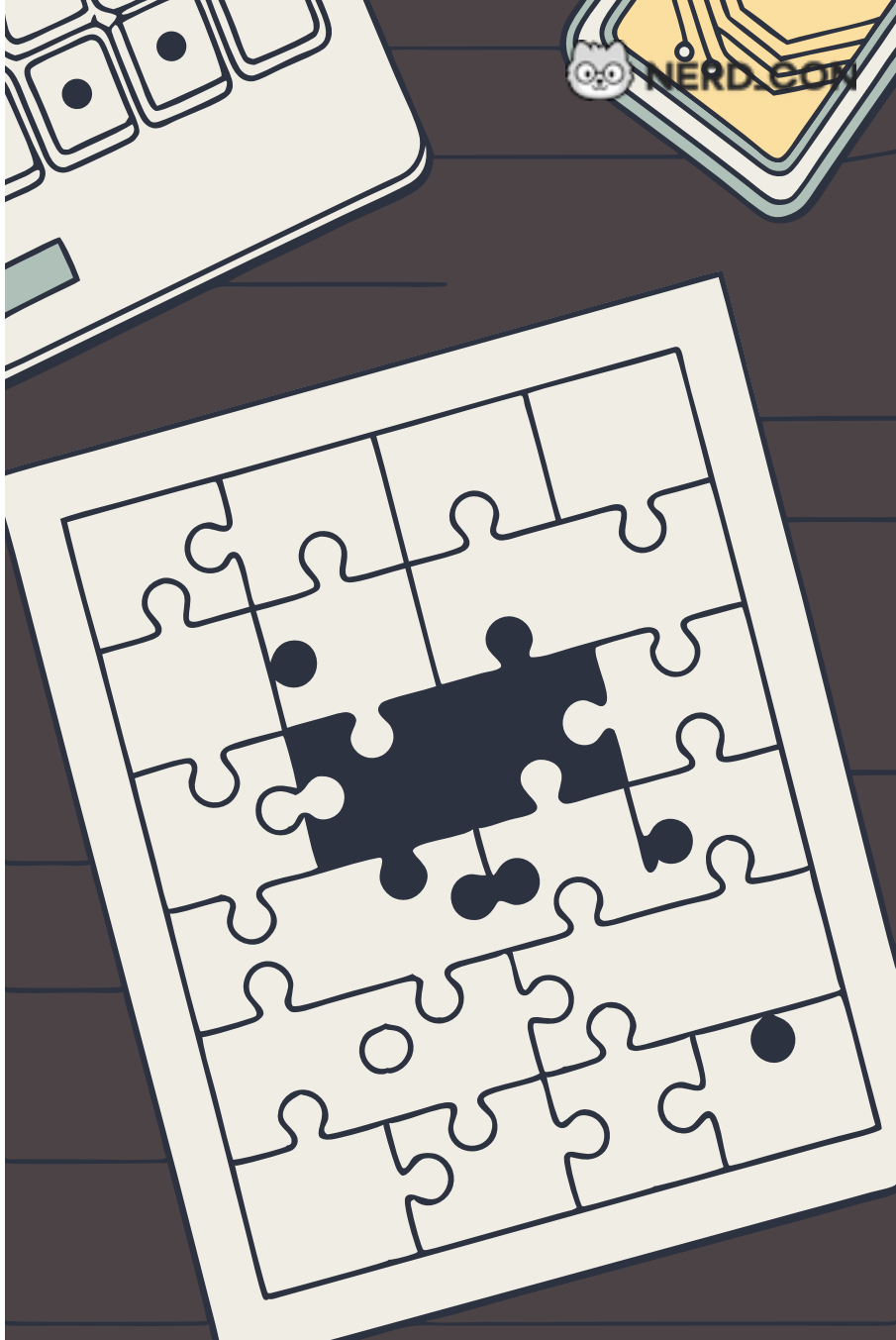
DB, 메세징, 외부 API 등의 실제 인프라 연동 부재

3 Mock 기반 오류

실제 환경과 다른 상호작용 동작 가능성 (간접 대역 설정 오류)

4 일부 버그 탐지 불가

통합 과정에서 발생하는 오류 놓침



통합 테스트의 중요성

컴포넌트 간 상호작용 검증

여러 컴포넌트가 함께 작동할 때 발생할 수 있는 문제를 사전에 발견

운영 환경 유사 인프라 특화 케이스 처리 검증

실제 운영환경 인프라에 특화된 케이스 처리(트랜잭션, 락, 재처리)가 예상대로 동작하는지 확인

시나리오 워크플로우 검증

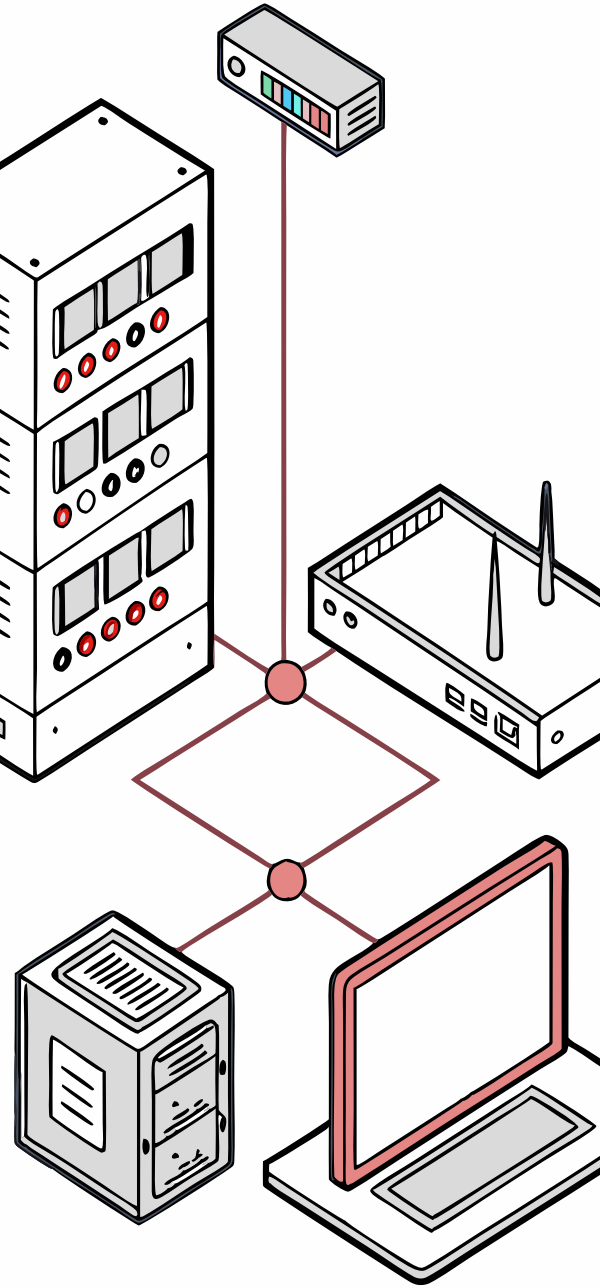
사용자 시나리오에 따른 서드파티 시스템들간의 동작 흐름이 올바르게 작동하는지 확인

단위테스트에서 검증되지 못한 경계테스트 검증

단위테스트만으로는 검증하기 어려운 시스템 간 경계 조건과 예외 상황에 대한 검증 가능

프레임워크 특화 구성 및 동작 검증

프레임워크의 환경설정, 객체 주입, 생명주기 및 이벤트 처리 등의 특수한 기능이 의도한 대로 동작하는지 검증



NERD_CON

실무에서 필요한 통합테스트 영역

데이터베이스

쿼리, 트랜잭션, 락 등

Cache

캐시/무효화 동작, 만료 정책 확인

외부 API

요청/응답 처리, 에러 핸들링

메세징

비동기 처리, 이벤트 발행/구독

보안

인증/인가, 토큰 처리, 접근 제어

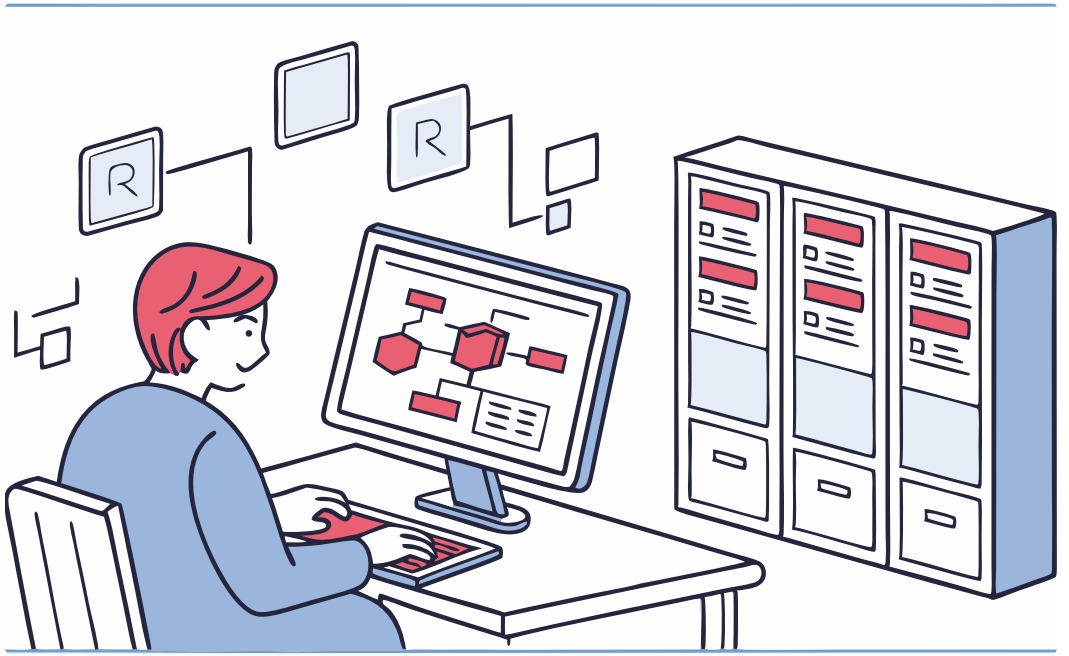
성능 및 부하

성능/부하테스트와는 별도

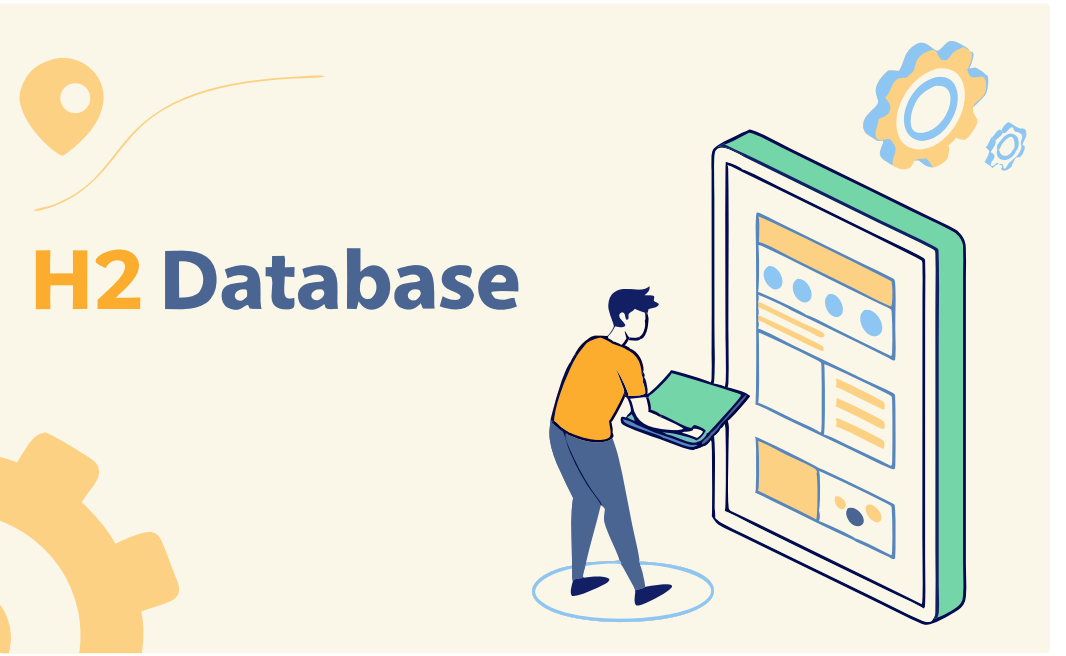
성능/부하 확인을 위한 간접 검증, 타임아웃 동작 검증

In-Memory기반 테스트 지원 도구

인메모리 기반 테스트 도구는 빠른 테스트 실행과 환경 구성의 편리함을 제공하지만, **실제 환경과의 차이로 인한 한계점도 존재**



Embedded Redis



H2 Database

운영 환경과의 격차를 줄이는 신뢰할 수 있는 통합테스트 인프라 구성

Testcontainers 활용

의존성 추가

testcontainers 라이브러리 설정

컨테이너 정의

MySQL, Redis, Kafka 등 상호작용 인프라 컨테이너 구성

테스트 실행

격리된 실제 인프라로 안정적인 테스트

Docker 구성 접근법

Docker Compose 작성

테스트용 인프라 서비스 정의

환경 설정

로컬 개발 및 CI/CD 파이프라인 연동

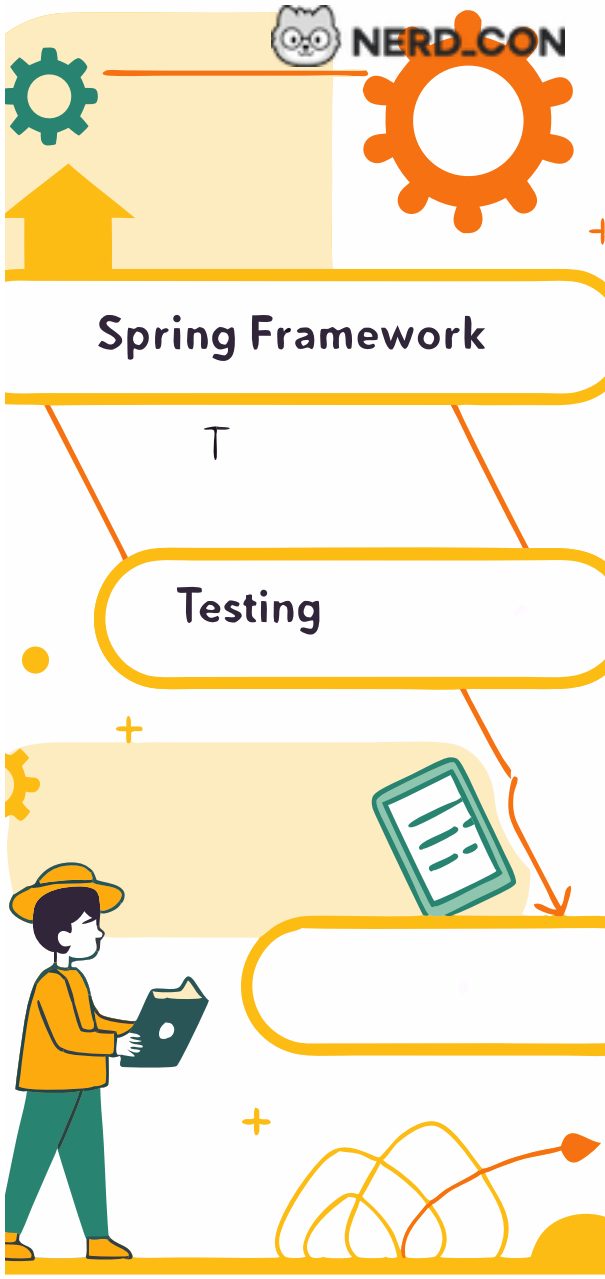
일관된 테스트

로컬 환경개발부터 통합테스트 및 배포까지 동일한 환경 보장

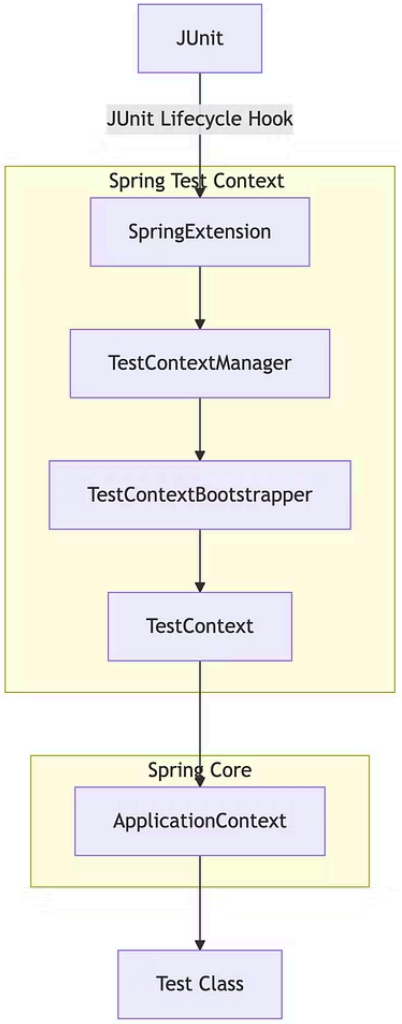
Spring Framework 기반 통합 테스트

Spring 통합 테스트 어노테이션 지원

@SpringBootTest	전체 애플리케이션 컨텍스트 로드
@WebMvcTest	웹 레이어만 테스트
@DataJpaTest	JPA 컴포넌트만 테스트
@TestConfiguration	테스트용 Bean 구성 정의
@MockBean, @SpyBean	특정 Bean을 모의 객체로 대체



Spring 통합테스트 동작 과정



NERD_CON

Spring 통합테스트 문제

ApplicationContext 로딩 과정으로 인한 테스트속도 저하

초기 설정 준비

테스트 실행 시 스프링이 테스트용 ApplicationContext

컴포넌트 스캐닝

Spring Bean 스캔 및 AutoConfiguration

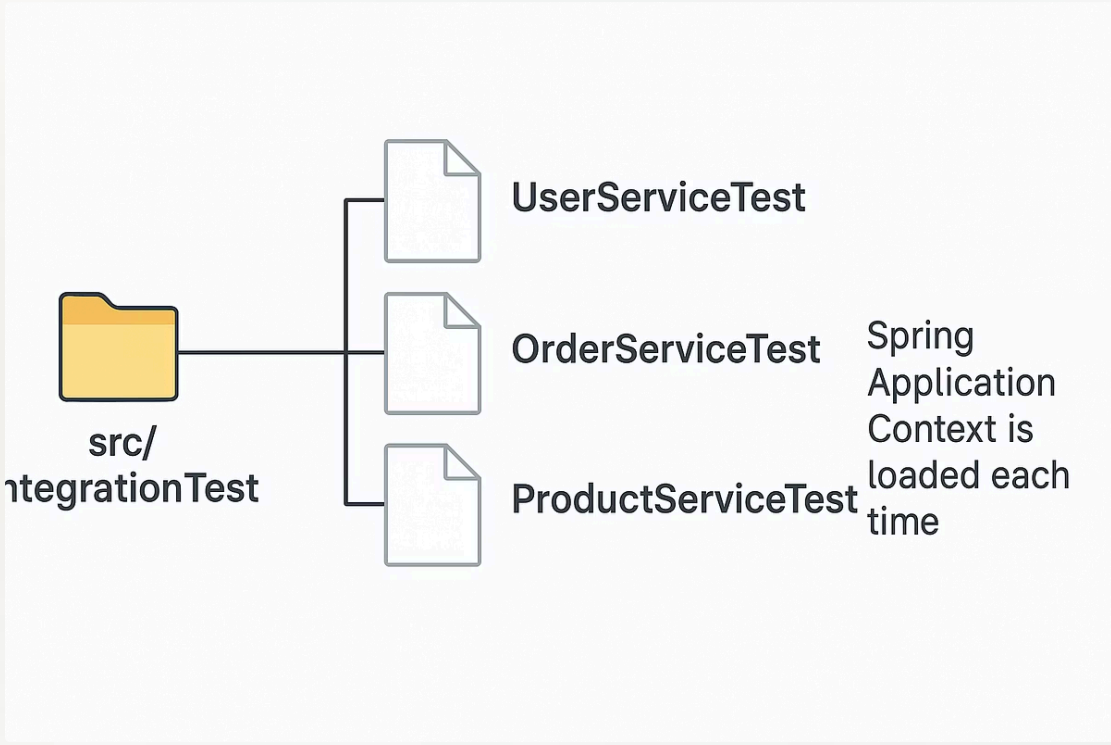
빈 인스턴스화 및 주입

스캐닝 완료된 Bean 인스턴스화 & 의존성 주입

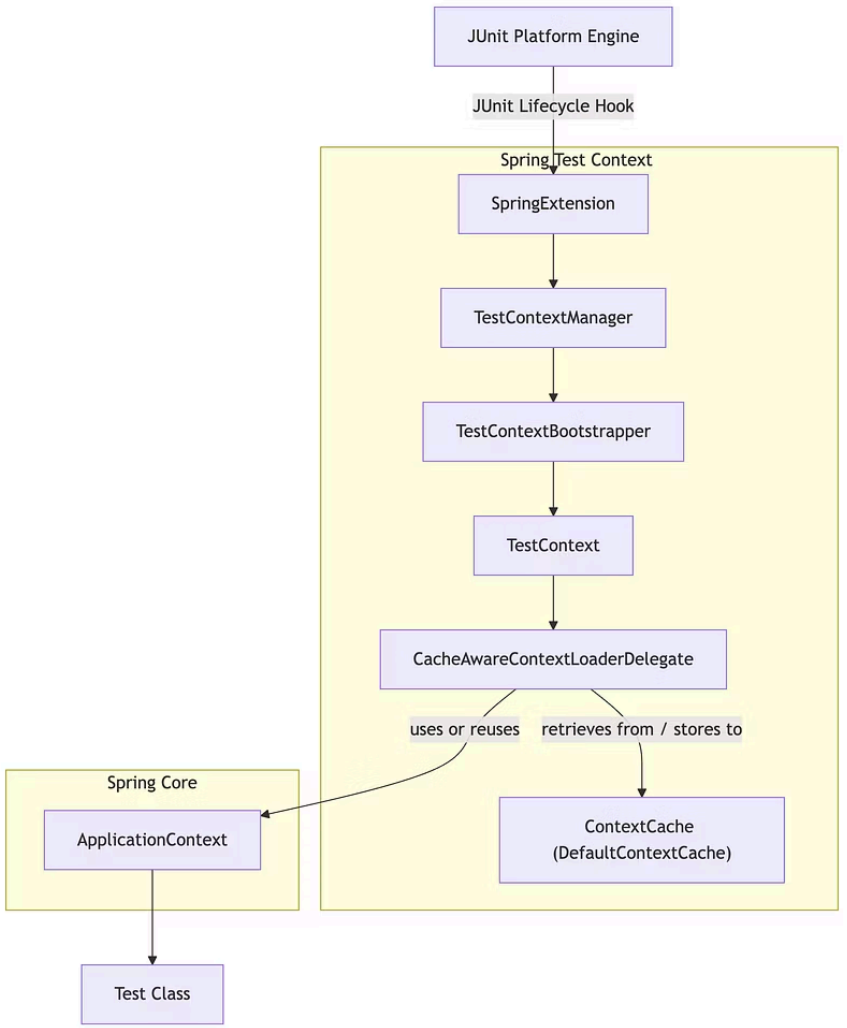
초기화 로직 실행

테스트 준비 완료

이 모든 과정이 **매 테스트**마다 반복되어 실행 시간 증가



Spring Test ContextCaching 활용



Spring 통합테스트 실전 활용

Spring Test Context Caching 활용

Context Caching

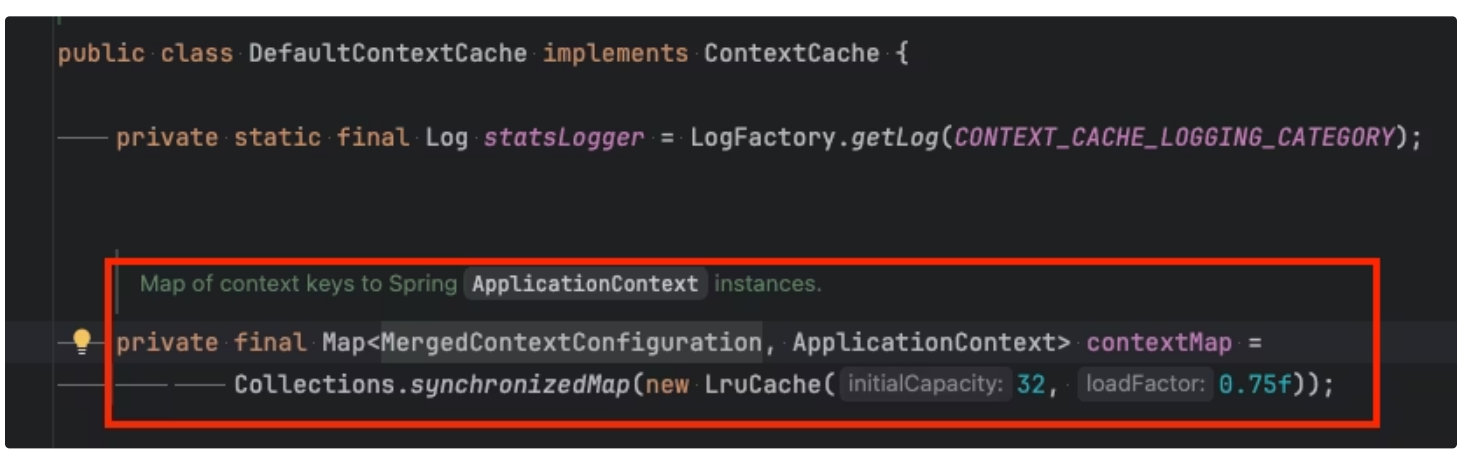
Once the TestContext framework loads an `ApplicationContext` (or `WebApplicationContext`) for a test, that context is cached and reused for all subsequent tests that declare the same unique context configuration within the same test suite. To understand how caching works, it is important to understand what is meant by "unique" and "test suite."

An `ApplicationContext` can be uniquely identified by the combination of configuration parameters that is used to load it. Consequently, the unique combination of configuration parameters is used to generate a key under which the context is cached. The TestContext framework uses the following configuration parameters to build the context cache key:

- `locations` (from `@ContextConfiguration`)
- `classes` (from `@ContextConfiguration`)
- `contextInitializerClasses` (from `@ContextConfiguration`)
- `contextCustomizers` (from `ContextCustomizerFactory`) – this includes `@DynamicPropertySource` methods, bean overrides (such as `@TestBean`, `@MockitoBean`, `@MockitoSpyBean` etc.), as well as various features from Spring Boot's testing support.
- `contextLoader` (from `@ContextConfiguration`)
- `parent` (from `@ContextHierarchy`)
- `activeProfiles` (from `@ActiveProfiles`)
- `propertySourceDescriptors` (from `@TestPropertySource`)
- `propertySourceProperties` (from `@TestPropertySource`)
- `resourceBasePath` (from `@WebAppConfiguration`)

For example, if `TestClassA` specifies `{"app-config.xml", "test-config.xml"}` for the `locations` (or `value`) attribute of `@ContextConfiguration`, the TestContext framework loads the corresponding `ApplicationContext` and stores it in a `static` context cache under a key that is based solely on those locations. So, if `TestClassB` also defines `{"app-config.xml", "test-config.xml"}` for its locations (either explicitly or implicitly through inheritance) but does not define `@WebAppConfiguration`, a different `ContextLoader`, different active profiles, different context initializers, different test property sources, or a different parent context, then the same `ApplicationContext` is shared by both test classes. This means that the setup cost for loading an application context is incurred only once (per test suite), and subsequent test execution is much faster.

Context Caching의 기준이 되는 Key를 적절하게 테스트 맥락에 따라 설정하여 캐싱을 활용할 것



Spring 통합테스트 실전 활용

Test Context 최소화 및 분리

- 각 모듈 성격에 따른 Spring Bean 구성 및 자동화
- 불필요한 AutoConfiguration 최소화
- Spring 통합 환경 Test dataset 자동화

Slice Testing

- 애플리케이션 계층에 맞는 Slice Testing 어노테이션 적극 활용
  - @DataJpaTest, @WebMvcTest ...

Spring Bean TestDouble 최소화

- 적절한 인터페이스 설계로 하여금 테스트 전용 Fake Object 활용
- MockBean, SpyBean 공통 구성 (TestConfiguration)

NERD_CON

Spring 통합테스트 실전 활용

애플리케이션 유형별 **TestContext** 구성

웹 애플리케이션

Controller 중심 테스트와 WebMvcTest 활용, Security 구성, Interceptor, Filter

데이터 중심 애플리케이션

Repository 레이어 테스트를 위한 DataJpaTest, 인메모리 DB 설정

배치 애플리케이션

SpringBatchTest 활용 JobLauncherTestUtils, Job, StepScope 등 테스트

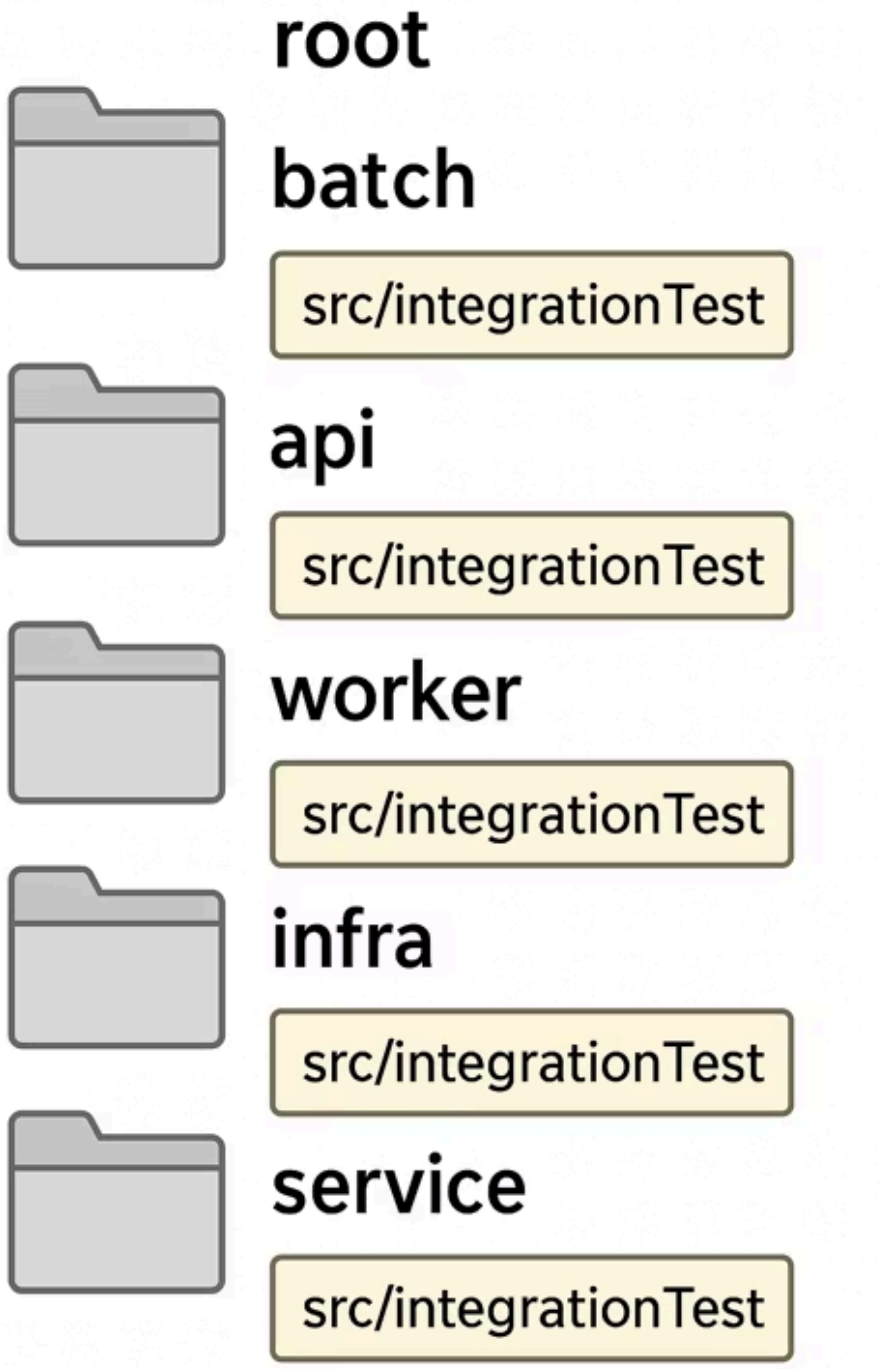
외부 시스템 **infra**

외부 서비스 의존성을 위한
WireMock/MockRestServiceServer 구성

이벤트 기반 애플리케이션

메시지 브로커 테스트 설정

각 컴포넌트 특성에 맞는 설정으로 테스트 실행 속도와 안정성 확보



NERD_CON

Spring 통합테스트 실전 활용

테스트 설계 시 마주하는 중요한 균형점 (테스트 격리 vs 속도)

높은 격리성

매번 DB 초기화

완벽한 격리를 제공하지만 테스트 실행 속도가 현저히 느려집니다. 각 테스트 시나리오의 완전한 독립성이 필요한 경우에 적합합니다.

병렬 실행

분리된 실행 환경을 제공하여 격리와 속도를 모두 개선할 수 있지만, 추가적인 인프라 설정이 필요합니다.

높은 실행 속도

트랜잭션 롤백

적절한 수준의 격리와 빠른 속도의 균형을 제공. 대부분의 통합 테스트 시나리오에서 효율적인 방법

공유 상태

테스트 간 격리가 없어 속도가 매우 빠르지만, 테스트 간 의존성으로 인한 부작용이 발생할 수 있습니다. 읽기 전용 테스트에 적합합니다.

상황에 따라 애플리케이션 컨텍스트를 완전히 격리하거나 공유하는 전략 간의 트레이드오프를 신중히 고려할 필요가 있음. 테스트의 신뢰성과 효율성 사이의 균형을 찾는 것이 중요

핵심 요약: 실무 중심 테스트 전략

계층적 접근

단위, 통합, E2E 테스트의 적절한 조합 (계층 경계 조건간의 테스트 커버)

실용적 구현

비즈니스 의도를 명확히 드러내는 테스트 코드, 커버리지보다 실제 비즈니스 가치 중심의 테스트

균형 전략

테스트 격리와 실행 속도 사이의 균형점 찾기

프레임워크 활용

Spring 테스트 도구의 효율적 사용과 캐싱 최적화

테스트는 단순한 코드 검증을 넘어 설계 개선, 배포 불안 해소, 그리고 팀 커뮤니케이션 도구로서 가치를 지닙니다. 이론적 완벽함보다 실용적 접근이 중요합니다.

애플리케이션의 특성과 팀 상황에 맞는 균형잡힌 테스트 전략이 장기적으로 안정성과 생산성을 함께 향상시킵니다.