웹을 통한 DAO의 문제점

: 거쳐하는 것들이 웹 서버, servlet, mvc, jsp 등 너무 많기 때문에 어디서 에러가 나는 것인지 정확히 파악하는 데에만 많은 시간이 소요된다.

단위 테스트(unit Test)

: 작은 단위의 코드에 대해 테스트를 수행한 것(mvc 클래스, 서비스 오브젝트, 배포등이 필요없다)

\*개발자 테스트 : 개발자를 위한 테스트(개발자 입장에서 만들어진 것)

자동수행 테스트 코드

: main()에서 단순 데이터를 입력 실행시키는 것만으로 나머지 연결 기능들은 알아서 자동적으로 동작해야 한다.

테스트 검증의 자동화(에러 처리)

If(!user.getName().equals(user2.getName()))

{ System.out.println(“테스트 실패 (name)”); }

… 이후 password 조회실패 시 “실패”

모두 통과할 경우 “성공”

ㅋ

포괄적인 테스트(Comprehensive test)

: 만들어진 코드의 기능을 모두 점검할 수 있는 테스트, 테스트 검증의 자동화를 전반적인 코드에 걸쳐 만드는 것

Junit

: 테스트 프레임 워크로, 프레임 워크의 기본 동작원리가 IoC이다.

* IDE의 포함된 Junit을 이용하면 main()을 구현하지 않아도 되 편리하다.

Get()의 예외처리

만약 get() 메소드의 매개변수에 해당하는 값이 없을 경우에는?

두 가지 방법이 있는데, 하나는 null과 같이 특별한 값을 리턴하는 것이고, 하나는 예외처리를 하는 것이다.

조건 : 어떤 조건을 가지고 ex) dao.deleteAll() assertThat(dao.getCount(), is(0))

행위 : 무엇을 할 때 ex) get(“unknown\_id”);

결과 : 어떤 결과가 나온다 ex) @Test(expected = EmptyResultDataAccessException)

테스트 주도 개발(테스트 우선 개발) TDD : Test Driven Development TFD : Test First Depelopment

: 만들고자 하는 기능의 내용을 담고 있으면서 만들어진 코드를 검증도 해줄 수 있도록 테스트 코드를 먼저 만들고, 테스트를 성공하게 해주는 코드를 작성하는 개발 방법

* 장점

테스트를 먼저 만들어놨기에 테스트에 맞는 코드를 작성해 바로바로 점검해 볼 수 있으므로,

테스트 시간이 매우 줄어든다.

코드 작성 후 테스트를 만드는 것 보다 명확히 오류에 대해서 파악할 수 있어 좋다.

@Before -> @Test -> @After이 있다.

But, Before메소드는 @Test의 개수만큼 반복되기에 애플리케이션 컨텍스트도 세 번 만들어진다.

간단한 빈이면 상관 없지만, 어떤 빈은 오브젝트가 생성될 때 자체적인 초기화 작업을 진행해서 제법 많은 시간을 필요로 하기에 그대로 쓰는 것은 비효율적이다.

다행이도애플리케이션 컨텍스트는 초기화되고 나면 내부의 상태가 바뀌는 일은 거의 없다.

빈은 싱글톤으로 만들었기 때문에상태를 갖지 않는다. userDao 빈을 가져다 add() 메소드 등을 쓴다고 해도 빈의 상태가 바뀌지는 않는다. DB의 상태는 각 테스트에서 알아서 관리할 것이므로 문제 되지 않는다. Junit은 테스트 클래스 전체에 걸쳐 딱 한번만 실행되는 @BeforeClass 스태틱 메소드를 지원한다.

@RunWith

: Junit 프레임워크 테스트 실행 방법을 확장할 때 사용하는 애노테이션이다.

SpringJUnit4ClassRunner라는 Junit 용 테스트 컨텍스트 프레임워크 확장 클래스를 지정해주면

Junit이 테스트를 진행하는 중에 테스트가 사용할 애플리케이션 컨텍스트를 만들고 관리하는 작업을 진행해준다.

@ContextConfiguration

: 자동으로 만들어줄 애플리케이션 컨텍스트의 설정파일 위치를 지정한 것

픽스처

: 테스트를 수행하는 데 필요한 정보나 오브젝트를 의미한다.

픽스처는 여러 테스트에서 반복적으로 사용되기 때문에 @Before 메소드를 이용해 생성해 두면 편리하다.

테스트 클래스의 컨텍스트 공유

: 스프링 테스트 컨텍스트 프레임워크의 기능은 하나의 테스트 클래스 안에서 애플리에킹션 컨텍스트를 공유해주는 것이 전부가 아니다. 여러 개의 테스트 클래스가 있으면 여러 테스트 클래스 모두 같은 설정파일을 가진 애플리케이션 컨텍스트를 사용한다면, 스프링은 테스트 클래스 사이에서도 애플리케이션 컨텍스트를 공유하게 해준다

@Autowired는 컨텍스트 내에서 변수 타입과 일치하는 컨텍스트 내의 빈을 찾아 연결한다.

만약 변수에 @Autowired를 위치시키면 어떻게 될까?

: 변수는 빈이 아니므로 원래는 에러가 나야 하지만, 컨텍스트는 자기 자신을 빈으로 만들기 때문에 컨텍스트 내부에 있는 변수들도 DI를 받을 수 있다.

단. @Autowired는 같은 타입의 빈이 두 개 이상 있는 경우에는 타입만으로는 어떤 빈을 가져올지 결정할 수 없다.

테스트 DB에 연결해주는 DataSource를테스트 내에서 직접 만들 수 있다.

SingleConnectionDataSouce

: DB커넥션을 하나만 만들어두고 계속 사용하는 스프링이 제공하는 가장 빠른 연결 클래스

다중사용자 환경에서는 사용할 수 없겠지만, 순차적으로 진행되는 테스트에는 상관없다.

@DirtiesContext

: 스프링의 테스트 컨텍스트 프레임워크에게 해당 클래스의 테스트에서 애플리케이션 컨텍스트의 상태를 변경한다는 것을 알려준다.

dirtiescontext가 붙은 프레임은 xml설정 파일을 공유하지 않게 하는 기능을 가진다.

그리고 이 애노테이션은 클래스 뿐만 아니라 메소드에도 사용이 가능함으로 메소드 레벨에 붙여주는 편이 낫다. 해당 메소드의 실행이 끝나고 나면 이후에 진행되는 테스트를 위해 변경된 애플리케이션 컨텍스트는 폐기되고 새로운 컨텍스트가 만들어진다.

테스트 코드에서 빈 오브젝트에 수동으로 DI 하는 방법은 장점보다 단점이 많다.

코드가 많아져 번거롭기만 하고 애플리케이션 컨텍스트도 매번 새로 만들어야 한다.

이를 위한 방식이 DataSource 클래스가 빈으로 정의된 테스트 전용 설정파일을 따로 만들어두는 방법이다.

컨테이너 없는 DI 테스트

: 스프링 컨테이너를 사용하지 않고 테스트를 만드는 것이다.

Dao 클래스나 DataSource 클래스에서 직접 스프링 DI 컨테이너에 의존하지는 않았기에

기존에 사용했던 방법(수동)을 토대로 DaoFactory를 만들어 UserDao가 동작함을 확인할 수 있다.

DI는 객체지향 프로그래밍 스타일이다. 따라서 DI를 위해 컨테이너가 반드시 필요한 것은 아니다.

침투적 기술

: 기술을 적용했을 때 애플리케이션코드에 기술 관련 API가 등장하거나, 특정 인터페이스나 클래스를 사용하도록 강제하는 기술을 말한다.

침투적 기술을 사용하면 애플리케이션 코드가 해당 기술에 종속되는 결과를 가져온다.

비침투적 기술

: 애플리케이션 로직을 담은 코드에 아무런 영향을 주지 않고 적용이 가능하다. 따라서 기술에 종속적이지 않은 순수한 코드를 유지할 수 있게 한다. 스프링은 이런 비침투적 기술의 대표적인 예이다. 그래서 스프링 컨테이너 없는 DI 테스트도 가능한 것이다.

세가지 테스트 방식의 활용 타이밍

왠만하면 실행시간도 짧고 간결한 수동DI를 이용하자.

하지만 구조가 복잡해지면 이를 위한 수동 DI는 꽤나 불편해지고, 구현에 힘들어질 것이다.

이 때는 스프링의 설정을 이용한 DI를 활용하자(@RunWith, @Autowired, @ContextConfiguration 등)

그리고, 테스트 설정을 따로 만들었다고 하더라도 때로는 예외적인 의존관계를 강제로 구성해서 테스트 해야 할 경우가 있다. 이 때는 컨텍스트에서 DI 받은 오브젝트에 다시 테스트 코드로 수동 DI를 해서 테스트 하는 방법을 사용하면 된다(@DirtiesContext, 추가 수동 DI)

학습테스트란?

: 자신이 만들지 않은 프레임워크나 다른 개발팀에서 만들어서 제공한 라이브러리 등에 대해서도 테스트를 작성해야 한다. 이를 학습테스트라고 한다.

* 목적

: 자신이 사용할 API나 프레임워크의 기능을 테스트 해 보면서 사용 방법을 익히려는 것이다.

따라서 테스트 이지만 기능에 대한 검증이 목적이 아니다.

* 장점

: 다양한 조건에 따른 기능을 손쉽게 확인해 볼 수 있다.

학습 테스트 코드를 개발 중에 참고할 수 있다.

프레임워크나 제품을 업그레이드할 때 호환성 검증을 도와준다.

테스트 작성에 대한 좋은 훈련이 된다.

버그 테스트

: 코드에 오류가있을 때 그오류를 가장 잘 드러내줄 수 있는 테스트

* 장점

테스트의완성도를 높여준다

버그의 내용을 명확하게 분석하게 해준다.

기술적인 문제를 해결하는 데 도움이 된다.

* 동등분할(equivalence partitioning)

: 같은 결과를 내는 값의 범위를 구분해서 각 대표 값으로 테스트를 하는 방법을 말한다.

어떤 작업의 결과의 종류가 true false 또는 예외 발생 세가지라면 각 결과를 내는 입력 값이나 상황의 조합을 만들어 모든 경우에 대한 테스트를 해보는 것이 좋다.

* 경계값 분석(boundary value analysis)

: 에러는 동등분할 범위의 경계에서 주로 많이 발생한다는 특징을 이용해서 경계의 근처에 있는 값을 이용해 테스트하는 방법이다. 보통 숫자의 입력 값인 경우 0이나 그 주변 값 또는 정수의 최대값, 최소값 등으로 테스트해보면 도움이 될 때가 많다.