Week 4

섹션 7.고급매핑

강의 요약

상속관계 매핑

DB는 상속관계가 없음, 객체는 상속 관계를 지원

@Inheritance(strategy=InheritanceType.XXX)

기본 : JOINED: 조인 전략

단순한 구조 : SINGLE_TABLE: 단일 테이블 전략(Default)

비추!! : TABLE_PER_CLASS: 구현 클래스마다 테이블 전략

@DiscriminatorColumn(name="DTYPE")

@DiscriminatorValue("XXX")

@MappedSuperclass

테이블과 관계 없고, 단순히 엔티티가 공통으로 사용하는 매핑 정보를 모으는 역할

주로 등록일, 수정일, 등록자, 수정자 같은 전체 엔티티에서 공통으로 적용하는 정보를 모을 때 사용

?

실시간 처리에 적합한 것, 통계처럼 복잡한 쿼리에서는 적합하지 않음.

-> 수 조 단위의 정산을 하는데, JPA로 다 처리한다.

크리티컬한 결제 같은 시스템도 JPA로 다 처리한다.

통계 쿼리처럼 매우 복잡한 SQL은 거의 다 **QueryDSL**로 처리하고, DTO로 뽑아낸다.

정말 안될 경우 네이티브 쿼리 사용한다.

QueryDSL이 뭐지?

JPA -> JPQL QueryDSL -> Gradle

JPQL(Java Persistence Query Language)

EntityManager의 find() 메서드로 데이터 조회 -> 복잡한걸 조회하기 부족해!! -> JPQL 개발

JPQL 엔티티 객체를 대상으로 쿼리를 작성함

특징
- 테이블이 아닌 객체를 검색하는 객체지향 쿼리
- SQL을 추상화했기 때문에 특정 벤더에 종속적이지 않음
- JPA에서는 JPQL을 분석해 SQL을 생성한 후 DB에서 조회

SQL과 다른점
- 엔티티와 속성은 대소문자를 구분, JPQL은 키워드는 구분하지 않음

```
SELECT m FROM Member AS m WHERE m.username = 'KIM '
```

- from뒤에 Entity(name = “이름”) : 엔티티이름을 사용함, 기본값은 클래스 이름
- JPQL에서 별칭은 필수. 대신 AS는 생략 가능

TypedQuery

```java
public static void typedQuery(EntityManager em) {
        String jpql = "SELECT b FROM Book b ";
        TypedQuery<Book> query = em.createQuery(jpql, Book.class);

        List<Book> bookList = query.getResultList();
        for( Book book : bookList) {
                System.out.println(book.getTitle());
        }
}
```

1. em.createQuery 메서드를 호출할 때 두 번째 인자로 엔티티 클래스를 넘겨줌

2. 반환되는 엔티티가 정해져 있을 때 사용하는 타입

3. 결과 조회, 결과가 없을 경우 빈 컬렉션으로 반환

Query

```
public static void Query(EntityManager em) {
        String jpql = "SELECT b.no, b.title FROM Book b";
        Query query = em.createQuery(jpql);
        List<Object> list = query.getResultList();
        for( Object object : list ) {
            Object[] results = (Object[]) object;

            for( Object result : results ) {
                System.out.print ( result );
            }
            System.out.println();
        }
}
```

1. 반환 타입이 명확하지 않을 경우 사용
   List의 제네릭 타입으로 Object 사용

2. 여러 엔티티나 컬럼을 선택할 경우 사용
   TypeQuery에서 선택적으로 명시하면 오류남

3. getSingleReuslt() : 결과가 하나일 때만 사용
결과가 없으면 javax.persistence.NoResultException
결과가 1보다 많으면 javax.persistence.NonUniqueResultException

setParameter()

```
public static void namedParameter(EntityManager em, String param1) {
        String jpql = "SELECT b FROM Book b WHERE title = :foo";
        TypedQuery<Book> query = em.createQuery(jpql, Book.class);
        query.setParameter("foo", param1);
        List<Book> bookList = query.getResultList();
        for( Book book : bookList) {
                System.out.println(book.getTitle());
        }
}
```

1

2

이름 기준 파라미터 바인딩

콜론(:)을 사용하여 데이터가 추가될 곳을 지정.
query.setParameter() 메서드를 호출해 데이터를 동적으로 바인딩.

```
public static void NamedParameter2(EntityManager em, String param1) {
        String jpql = "SELECT b FROM Book b WHERE title = ?1";
        TypedQuery<Book> query = em.createQuery(jpql, Book.class);
        query.setParameter(1, param1);
        List<Book> bookList = query.getResultList();
        for( Book book : bookList) {
                System.out.println(book.getTitle());
        }
}
```

1

2

위치 기준 파라미터 바인딩

물음표(?)을 사용하여 데이터가 추가될 곳을 지정.
query.setParameter() 메서드를 호출해 데이터를 동적으로 바인딩.
위치 값은 1부터 시작

Projection

SELECT 절에 조회할 대상을 지정하는 것.

- 엔티티 : 원하는 객체를 바로 조회

```
SELECT m FROM Member m
SELECT m.team FROM Member m
```

- 엠비디드 타입 : 복합 값 타입(address가 @Embedded 타입)

```
String query = "SELECT o.address FROM Order o";
List<Address> addresses = em
                          .createQuery(query, Address.class)
                          .getResultList();

select
    order.city
   , order.street
   , order.zipcode
from Orders order
```

- 스칼라 타입 : 숫자, 문자 등 기본 타입

```
List resultList = em
                  .createQuery("select m.name, m.age from Member m")
                  .getResultList();


select
    m.name
   , m.age
from Member m
```

DTO 사용

```
// DTO 사용 ( new 명령어 )
public static void useDTO (EntityManager em) {
        String jpql = "SELECT new com.victolee.example.dto.BookDTO(b.no, b.title) FROM Book b";
        TypedQuery<BookDTO> query = em.createQuery(jpql, BookDTO.class);

        List<BookDTO> list = query.getResultList();
        for( BookDTO dto : list) {
                System.out.println(dto.getTitle());
        }
}
```

- SELECT 다음 NEW 명령어 사용 시 반환받을 클래스 지정(패지키 명을 포함한 전체 클래스 명을 기입해야함)
  이 클래스의 생성자에 JPQL 조회 결과를 넘겨줄 수 있음(기본 생성자 + 순서와 타입이 일치하는 생성자 필요)
- New 명령어를 사용한 클래스로 TypeQuery 사용이 가능하여 객체 변환 작업에 효율적

QueryDSL

객체지향 쿼리 언어 종류
JPQL, Criteria, QueryDSL, NativeQuery

JPQL을 코드로 작성할 수 있도록 도와주는 빌더 API
JPA 표준 스펙이 아니라서 약간의 설정이 더 필요함. 하지만 Criteria 보다는 덜 복잡함.(표준 스펙은 Criteria)
복잡한 쿼리와 동적 쿼리를 깔끔히 해결해줌.

JPQL과 QueryDSL 뭐가 다르지?

JPQL

SQL, JPQL은 문자열이라서 Type-Check가 불가능함.
해당 쿼리 실행 시점에 오류를 발견한다.

QueryDSL

문자가 아닌 코드로 작성
컴파일 시점에 문법 오류 체크
단순하고 쉬움 코드모양이 JPQL과 거의 비슷
동적쿼리 가능
원하는 필드만 뽑아서 DTO로 뽑아내는 기능도 QueryDSL이 다 지원함

JPQL

```
public void startJPQL() {
        // memeber1을 찾아라
        Member result = em.createQuery(
                "select m from Member m " +
                "where m.username = :username", Member.class)
                .setParameter("username", "member1")
                .getSingleResult();
    }
```

QueryDSL

```
public void startQueryDsl() {
        // memeber1을 찾아라
        Member result = queryFactory
                .select(member)
                .from(member)
                .where(member.username.eq("member1"))
                .fetchOne();
    }
```

select(), from(), selectFrom()

where(), update(), set(), delete(),,,,

이 외에도 다양하게 제공

Criteria

```
//EntityManager나 EntityManagerFactory에서 CriteriaBuilder를 가져옴
CriteriaBuilder cb = em.getCriteriaBuilder();

//CriteriaBuilder로부터 CriteriaQuery객체를 얻어온다.
CriteriaQuery<Member> query = cb.createQuery(Member.class);

//조회의 시작점을 뜻하는 Root객체 생성
Root<Member> m = query.from(Member.class);

//쿼리생성
CriteriaQuery<Member> cq =
query.select(m).where(cb.equal(m.get("student_number"),
"0991022");
Member member = em.createQuery(cq).getMember();
```

QueryDSL

```
//JPAQuery 객체를 생성한다.
//생성자 매개변수로 EntityManager인스턴스 전달
JPAQuery query = new JPAQuery(em);

//조회의 시작점인 엔티티를 Q_.class로 생성해준다.
QMember member = QMember.member;

//쿼리, 결과조회
Member member = query.from(member)
                     .where(member.student_number.eq("0991022"))
                     .get(member);
```

QueryDSL이 훨씬 좋은것 같은데 그럼 JPQL은 언제 쓰는거지?

단순한거는 @Query써서 바로 쓰는게 편하다.

QueryDSL 사용하려면 Custom Repository를 넣어줘야하는데 이거 은근 귀찮,,,

그래서 정말 단순한거는 JPQL로 나머지는 다 QueryDSL로.

근데 막상 보면 저도 거의 QueryDSL 써요.

Maven vs Gradle

Maven

- Apache의 이름 아래 2004년 출시
- Ant를 사용하던 개발자들의 불편함을 해소 + 부가기능 추가
- 내가 사용할 라이브러리 뿐만 아니라 해당 라이브러리가 작동하는데 필요한 다른 라이브러리들까지 관리하고 네트워크를 통해 자동으로 다운받아줌.
- 프로젝트의 전체적인 라이프사이클을 관리하는 도구

- Pom.xml 사용
- xml 기반의 빌드 처리를 작성함. 간단한 내용이면 상관 없지만 복잡하게 되면 xml 기반 의한 묘사는 상당히 어려움

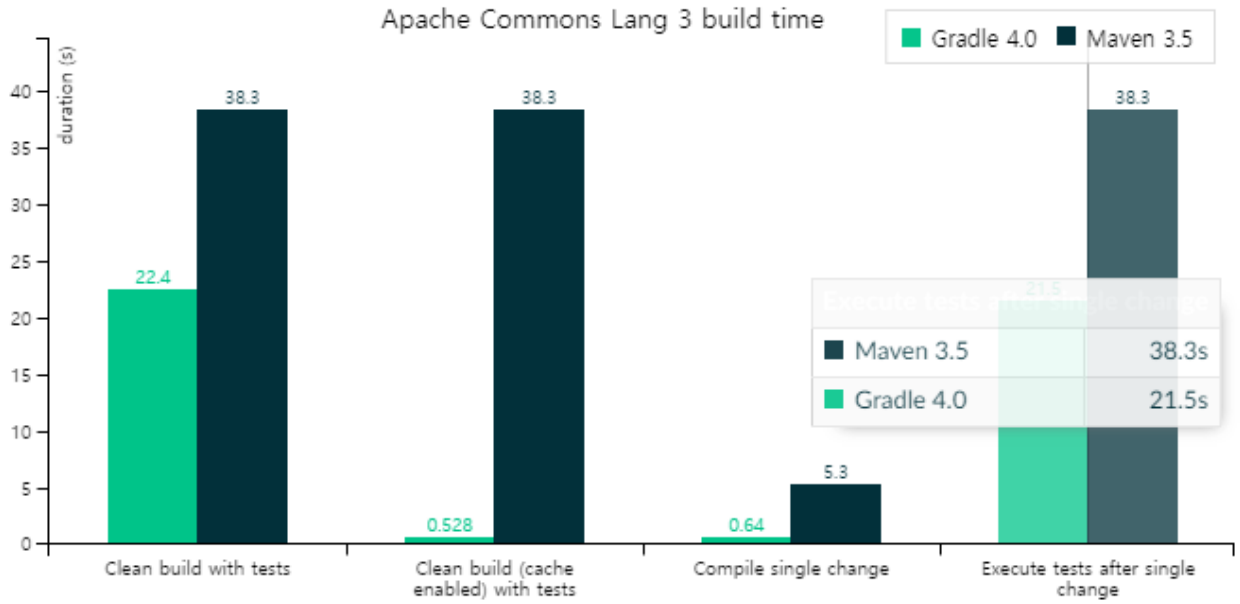
Gradle

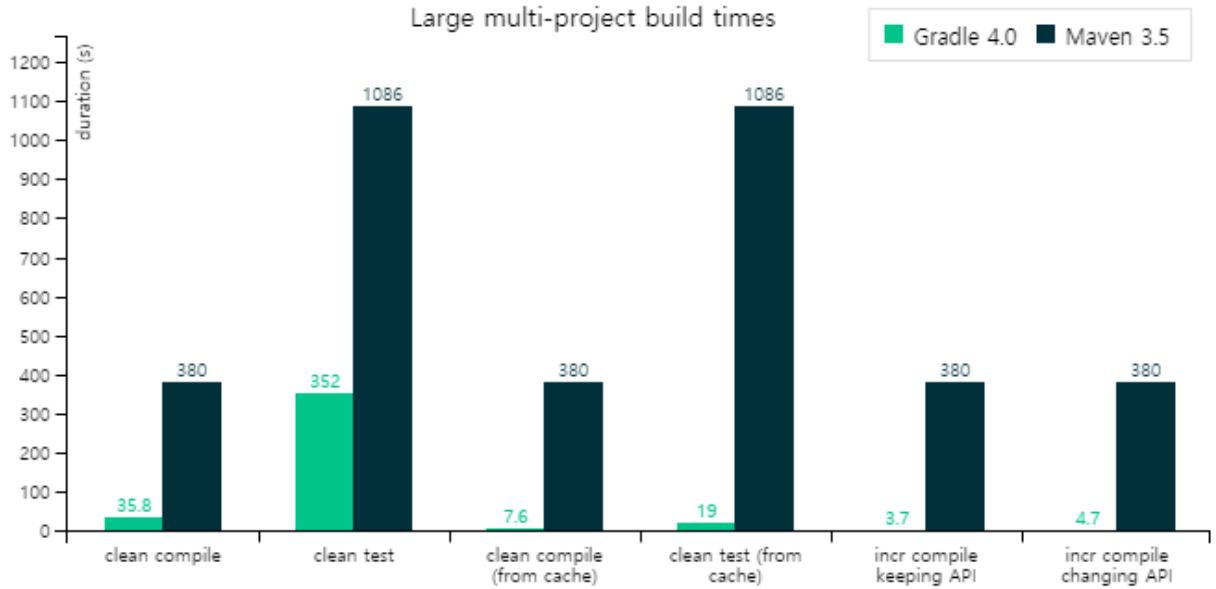
- Ant와 Maven의 장점을 모아모아 2012년 출시
- Android OS의 빌드 도구로 채택 됨 (JAVA, C, 파이썬 지원)
- Ant 빌더와 Groovy 스크립트를 기반으로 구축되어 기존 Ant의 역할과 배포 스크립트의 기능을 모두 사용가능

**Groovy는 자바 가상 머신에서 작동하는 동적 타이핑 프로그래밍 언어

- build.gradle 사용
- 별도의 빌드스크립트를 통해 사용할 어플리케이션 버전, 라이브러리등의 항목을 설정 할 수 있음.
- 스크립트 언어로 구성되어 있어 maven의 xml과 달리
변수선언, if, else, for 등의 로직이 구현 가능해 간결하게 구성할 수 있음.

Gradle





Gradle이 Maven보다 최대 100배 빠름

- Groovy를 사용해서 동적인 빌드는 Groovy 스크립트로 플러그인을 호출하거나 직접 코드를 짜면 됨
- 설정 주입(Configuration Injection) 방식을 사용해 공통 모듈을 상속해서 사용하는 단점을 커버함.
- 설정 주입 시 프로젝트의 조건을 체크할 수 있어서 프로젝트별로 주입되는 설정을 다르게 할수있음.
- 멀티 프로젝트에 매우 적합.

JUnit 라이브러리를 의존성 리스트에 추가

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="ht
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.programming.mitra</groupId>
    <artifactId>java-build-tools</artifactId>
    <packaging>jar</packaging>
    <version>1.0</version>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.11</version>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.3.2</version>
            </plugin>
        </plugins>
    </build>
</project>
```

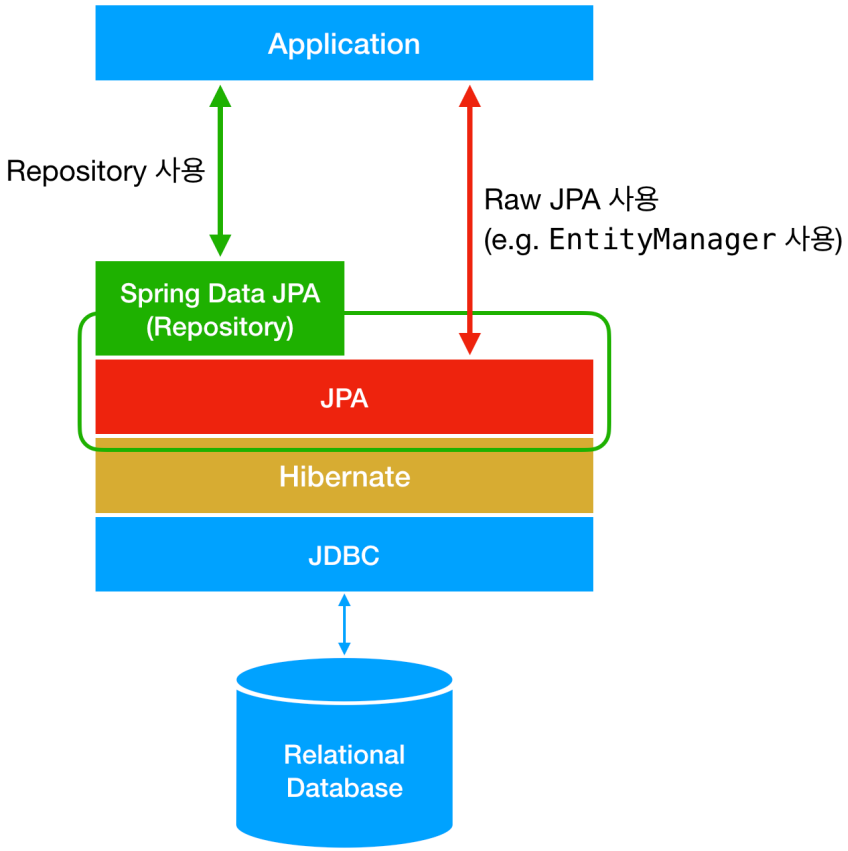
플러그인 추가 (Maven CheckStyle, FindBugs 및 PMD)

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-checkstyle-plugin</artifactId>
    <version>2.12.1</version>
    <executions>
        <execution>
            <configuration>
                <configLocation>config/checkstyle/checkstyle.xml
                <consoleOutput>true</consoleOutput>
                <failsOnError>true</failsOnError>
            </configuration>
            <goals>
                <goal>check</goal>
            </goals>
        </execution>
    </executions>
</plugin>
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>findbugs-maven-plugin</artifactId>
    <version>2.5.4</version>
    <executions>
        <execution>
            <goals>
                <goal>check</goal>
            </goals>
        </execution>
    </executions>
</plugin>
 <plugin>
     <groupId>org.apache.maven.plugins</groupId>
     <artifactId>maven-pmd-plugin</artifactId>
     <version>3.1</version>
     <executions>
         <execution>
             <goals>
                 <goal>check</goal>
             </goals>
         </execution>
     </executions>
 </plugin>
```

아주 간단하게 해결!!!

```
apply plugin:'java'
apply plugin:'checkstyle'
apply plugin:'findbugs'
apply plugin:'pmd'
version ='1.0'
repositories {
    mavenCentral()
}
dependencies {
    testCompile group:'junit', name:'junit', version:'4.11'
}
```

Spring Data JPA



스프링 부트 + JPA

EntityManager X

Repository O

JPA

JAVA에서 제공하는 인터페이스

자바 어플리케이션에서 관계형 데이터베이스를 어떻게 사용해야하는지 정의하는 한 방법

ORM 기술에 대한 명세서

Hibernate

JPA의 구현체, ORM(객체관계매핑) 프레임워크 중 하나

**ORM : 자바 어플리케이션 내의 객체들을 RDB에 있는 테이블로의 자동화된 영속화.

예전부터 사용해오던 ResultSet 객체를 사용 목적에 맞는 형태의 객체로 변환하는 작업.

Spring Data JPA

JPA를 쓰기 편하게 만들어놓은 모듈

JPA를 한단계 추상화시킨 Repository라는 인터페이스를 제공함으로써 이루어짐