

JPA 기반 프로젝트

- Spring Data JPA
- QueryDSL

2.

Spring Data JPA

반복되는 CRUD

```
public class MemberRepository {
```

```
    public void save(Member member) {...}  
    public Member findOne(Long id) {...}  
    public List<Member> findAll() {...}
```

```
    public Member findByUsername(String username) {...}  
}
```

```
public class ItemRepository {
```

```
    public void save(Item item) {...}  
    public Member findOne(Long id) {...}  
    public List<Member> findAll() {...}
```

```
}
```

스프링 데이터 JPA 소개

- 지루하게 반복되는 CRUD 문제를 세련된 방법으로 해결
- 개발자는 인터페이스만 작성
- 스프링 데이터 JPA가 구현 객체를 동적으로 생성해서 주입

스프링 데이터 JPA 적용 전

```
public class MemberRepository {  
  
    public void save(Member member) {...}  
    public Member findOne(Long id) {...}  
    public List<Member> findAll() {...}  
  
    public Member findByUsername(String username) {...}  
}
```

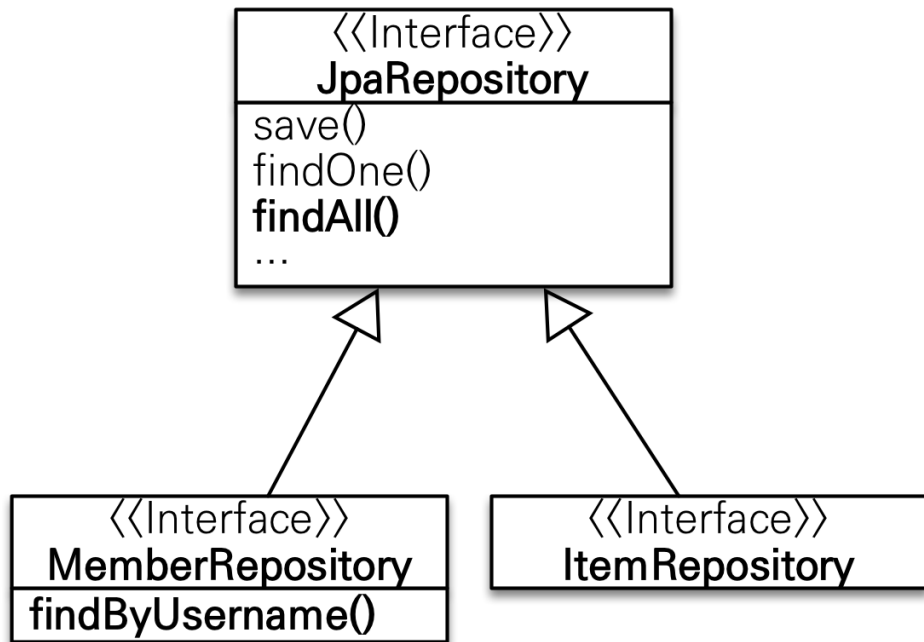
```
public class ItemRepository {  
  
    public void save(Item item) {...}  
    public Member findOne(Long id) {...}  
    public List<Member> findAll() {...}  
}
```

스프링 데이터 JPA 적용 후

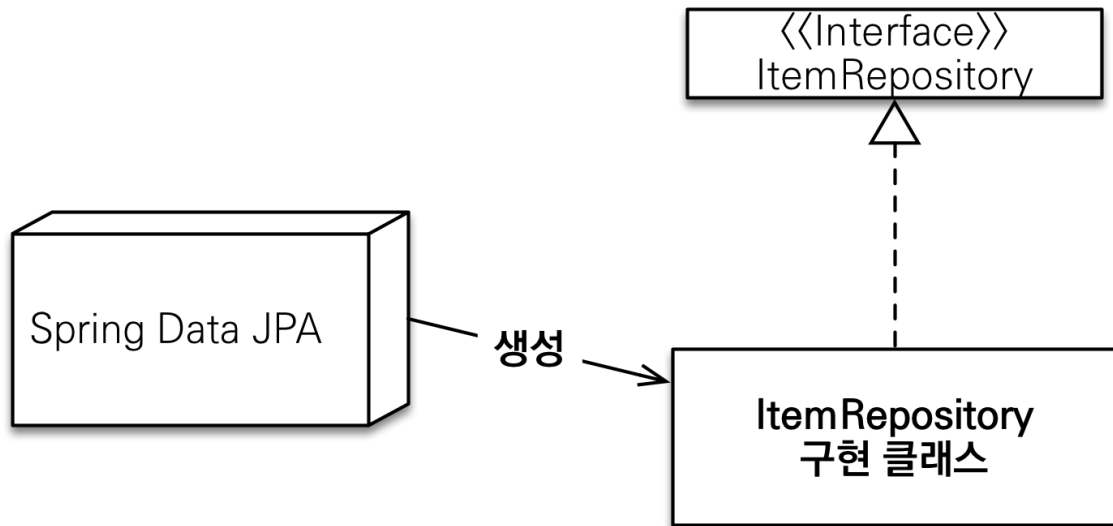
```
public interface MemberRepository extends JpaRepository<Member, Long>{  
    Member findByUsername(String username);  
}
```

```
public interface ItemRepository extends JpaRepository<Item, Long> {  
    //비어있음  
}
```

스프링 데이터 JPA 적용 후 클래스 다이어그램



스프링 데이터 JPA가 구현 클래스 생성



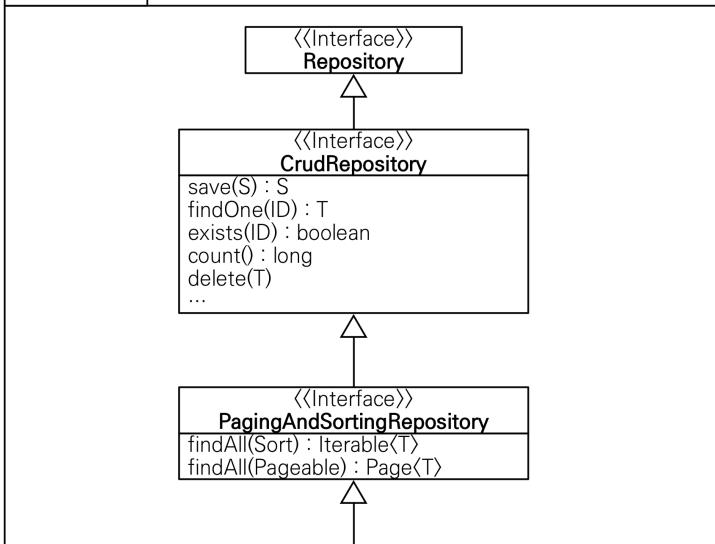
공통 인터페이스 기능

- JpaRepository 인터페이스: 공통 CRUD 제공
- 제네릭은 <엔티티, 식별자>로 설정

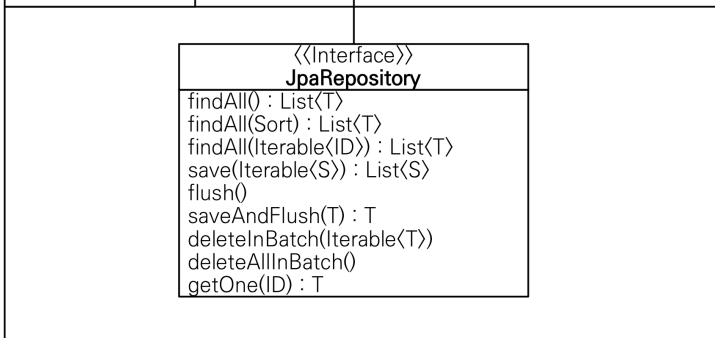
```
public interface MemberRepository extends JpaRepository<Member, Long> {  
    //비어있음  
}
```

공통 인터페이스 기능

스프링 데이터



스프링 데이터 JPA



쿼리 메서드 기능

- 메서드 이름으로 쿼리 생성
- @Query 어노테이션으로 쿼리 직접 정의

메서드 이름으로 쿼리 생성

- 메서드 이름만으로 JPQL 쿼리 생성

```
public interface MemberRepository extends JpaRepository<Member, Long> {  
  
    List<Member> findByName(String username);  
  
}
```

메서드 이름으로 쿼리 생성 - 사용 코드

```
List<Member> member = memberRepository.findByName("hello")
```

실행된 SQL

```
SELECT * FROM MEMBER M WHERE M.NAME = 'hello'
```

이름으로 검색 + 정렬

```
public interface MemberRepository extends JpaRepository<Member, Long> {  
    List<Member> findByName(String username, Sort sort);  
}
```

실행된 SQL

```
SELECT * FROM MEMBER M WHERE M.NAME = 'hello'  
ORDER BY AGE DESC
```

이름으로 검색 + 정렬 + 페이징

```
public interface MemberRepository extends JpaRepository<Member, Long> {  
    Page<Member> findByName(String username, Pageable pageable);  
}
```

실행된 SQL 2가지

```
SELECT * //데이터 조회  
FROM  
  ( SELECT ROW_.*, ROWNUM ROWNUM_  
    FROM  
      ( SELECT M.*  
        FROM MEMBER M WHERE M.NAME = 'hello'  
        ORDER BY M.NAME  
      ) ROW_  
    WHERE ROWNUM <= ?  
  )  
WHERE ROWNUM_ > ?
```

```
//전체 수 조회  
SELECT COUNT(1)  
FROM MEMBER M WHERE M.NAME = 'hello'
```

이름으로 검색 + 정렬 + 페이징, 사용 코드

```
Pagable page = new PageRequest(1, 20, new Sort...);
```

```
Page<Member> result = memberRepoitory.findByName("hello", page);
```

```
int total = result.getTotalElements(); //전체 수
```

```
List<Member> members = result.getContent(); //데이터
```

전체 페이지수, 다음 페이지 및 페이징을 위한 API 다 구현되어 있음

@Query, JPQL 정의

- @Query를 사용해서 직접 JPQL 지정

```
public interface MemberRepository extends JpaRepository<Member, Long> {
```

```
    @Query("select m from Member m where m.username = ?1")  
    Member findByUsername(String username, Pageable pageable);
```

```
}
```

반환 타입

```
List<Member> findByName(String name); //컬렉션
```

```
Member findByEmail(String email); //단건
```

Web 페이징과 정렬 기능

- 컨트롤러에서 페이징 처리 객체를 바로 받을 수 있음
- page: 현재 페이지
- size: 한 페이지에 노출할 데이터 건수
- sort: 정렬 조건

/members?**page**=0&**size**=20&**sort**=name,desc

```
@RequestMapping(value = "/members", method = RequestMethod.GET)  
String list(Pageable pageable, Model model) {}
```

Web 도메인 클래스 컨버터 기능

- 컨트롤러에서 식별자로 도메인 클래스 찾을

/members/100

```
@RequestMapping("/members/{memberId}")  
Member member(@PathVariable("memberId") Member member) {  
    return member;  
}
```

3.

QueryDSL

QueryDSL 소개

- SQL, JPQL을 코드로 작성할 수 있도록 도와주는 빌더 API
- JPA 크리테리아에 비해서 편리하고 실용적임
- 오픈소스

SQL, JPQL의 문제점

- SQL, JPQL은 문자, Type-check 불가능
- 해당 로직 실행 전까지 작동여부 확인 불가

```
SELECT * FROM MEMBERR WHERE MEMBER_ID = '100'
```



실행 시점에 오류 발견

QueryDSL 장점

- 문자가 아닌 코드로 작성
- 컴파일 시점에 문법 오류 발견
- 코드 자동완성(IDE 도움)
- 단순하고 쉬움: 코드 모양이 JPQL과 거의 비슷
- 동적 쿼리

QueryDSL - 동작원리 쿼리타입 생성



QueryDSL 사용

```
//JPQL
```

```
select m from Member m where m.age > 18
```

```
JPAFactoryQuery query = new JPAQueryFactory(em);  
QMember m = QMember.member;
```

```
List<Member> list =  
    query.selectFrom(m)  
        .where(m.age.gt(18))  
        .orderBy(m.name.desc())  
        .fetch();
```

QueryDSL - 조인

```
JPAQueryFactory query = new JPAQueryFactory(em);  
QMember m = QMember.member;  
QTeam t = QTeam.team;
```

```
List<Member> list =  
    query.selectFrom(m)  
        .join(m.team, t)  
        .where(t.name.eq("teamA"))  
        .fetch();
```

QueryDSL - 페이징 API

```
JPAQueryFactory query = new JPAQueryFactory(em);  
QMember m = QMember.member;
```

```
List<Member> list =  
    query.selectFrom(m)  
        .orderBy(m.age.desc())  
        .offset(10)  
        .limit(20)  
        .fetch();
```


QueryDSL - 동적 쿼리

```
String name = "member";  
int age = 9;  
  
QMember m = QMember.member;  
  
BooleanBuilder builder = new BooleanBuilder();  
if (name != null) {  
    builder.and(m.name.contains(name));  
}  
if (age != 0) {  
    builder.and(m.age.gt(age));  
}  
  
List<Member> list =  
    query.selectFrom(m)  
        .where(builder)  
        .fetch();
```

QueryDSL - 이것은 자바다!

서비스 필수 제약조건

```
return query.selectFrom(coupon)
    .where(
        coupon.type.eq(typeParam),
        coupon.status.eq("LIVE"),
        marketing.viewCount.lt(marketing.maxCount)
    )
    .fetch();
```



QueryDSL - 이것은 자바다!

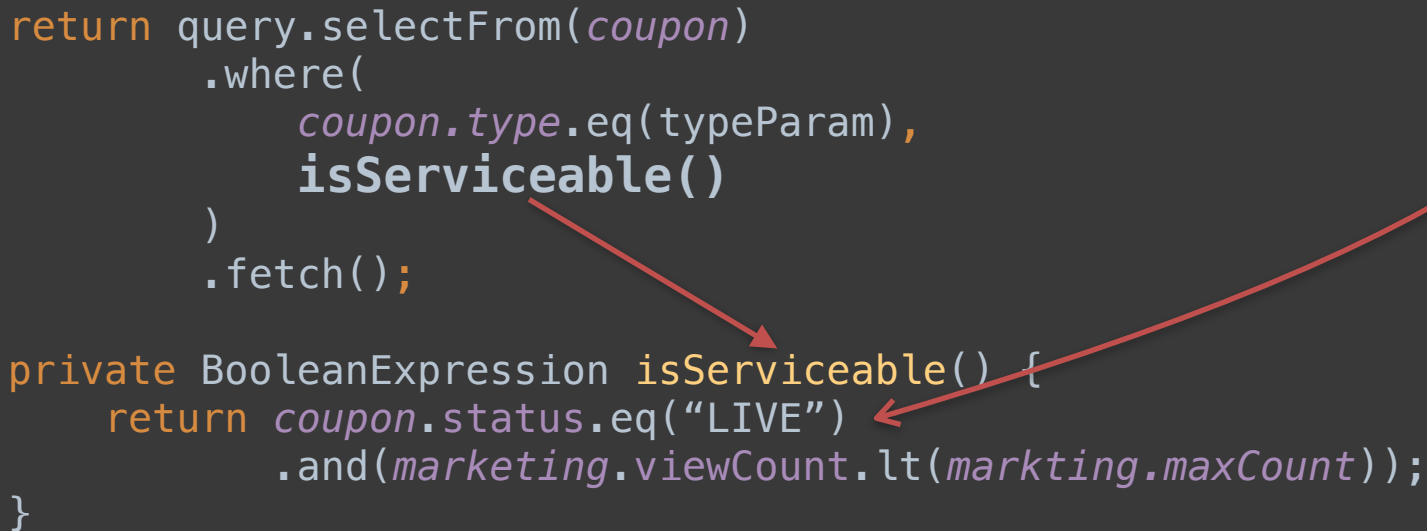
제약조건 조립가능

- 가독성, 재사용

서비스 필수 제약조건

```
return query.selectFrom(coupon)
    .where(
        coupon.type.eq(typeParam),
        isServiceable()
    )
    .fetch();

private BooleanExpression isServiceable() {
    return coupon.status.eq("LIVE")
        .and(marketing.viewCount.lt(marketing.maxCount));
}
```



5.

실무 경험 공유

실무 경험

- 테이블 중심에서 객체 중심으로 개발 패러다임이 변화
- 유연한 데이터베이스 변경의 장점과 테스트
 - Junit 통합 테스트시에 H2 DB 메모리 모드
 - 로컬 PC에는 H2 DB 서버 모드로 실행
 - 개발 운영은 MySQL, Oracle
- 데이터베이스 변경 경험(개발 도중 MySQL -> Oracle 바뀐적도 있다.)
- 테스트, 통합 테스트시에 CRUD는 믿고 간다.

실무 경험

- 빠른 오류 발견
 - 컴파일 시점!
 - 늦어도 애플리케이션 로딩 시점
- (최소한 쿼리 문법 실수나 오류는 거의 발생하지 않는다.)
- 대부분 비즈니스 로직 오류

실무 경험 - 성능

- JPA 자체로 인한 성능 저하 이슈는 거의 없음.
- 성능 이슈 대부분은 JPA를 잘 이해하지 못해서 발생
 - 즉시 로딩: 쿼리가 튼 -> 지연 로딩으로 변경
 - N+1 문제 -> 대부분 페치 조인으로 해결
- 내부 파서 문제: 2000줄 짜리 동적 쿼리 생성 1초
 - 정적 쿼리로 변경(하이버네이트는 파싱된 결과 재사용)

실무 경험 - 생산성

- 단순 코딩 시간 줄어듦 -> 개발 생산성 향상 -> 잉여 시간 발생
- 비즈니스 로직 작성시 흐름이 끊기지 않음
- 남는 시간에 더 많은 테스트 작성
- 남는 시간에 기술 공부
- 남는 시간에 코드 **금칠**...
- 팀원 대부분 다시는 과거로 돌아가고 싶어하지 않음

많이 하는 질문

1. ORM 프레임워크를 사용하면 SQL과 데이터베이스는 잘 몰라도 되나요?
2. 성능이 느리진 않나요?
3. 통계 쿼리처럼 매우 복잡한 SQL은 어떻게 하나요?
4. MyBatis와 어떤 차이가 있나요?
5. 하이버네이트 프레임워크를 신뢰할 수 있나요?
6. 제 주위에는 MyBatis(iBatis, myBatis)만 사용하는데요?
7. 학습곡선이 높다고 하던데요?

팀 서버 기술 스택

- Java 8
- Spring Framework
- JPA, Hibernate
- Spring Data JPA
- QueryDSL
- JUnit, Spock(Test)

우아한 형제들에서는?

- 수 조 단위 빌링 시스템
- 수 조 단위 정산 시스템

Q&A

Thank You