- **스프링 데이터 JPA**
  - **스프링 데이터 JPA 소개**
    - - 스프링 데이터 프로젝트
  - **스프링 데이터 JPA 설정하기**
    - - 필요 라이브러리
    - - 환경 설정
  - **공통 인터페이스 기능**
  - **쿼리 메서드 기능**
    - - 메서드 이름으로 쿼리 생성
    - - JPA NamedQuery
    - - @Query, 리파지토리 메서드에 쿼리 정의하기
    - - 파라미터 바인딩
    - - 벌크성 수정 쿼리
    - - 반환 타입
    - - 페이징과 정렬
    - - 힌트
    - - Lock
  - **Specifications (명세)**
  - **사용자 정의 리파지토리 구현**
  - **Web 확장**
    - - 설정
    - - 도메인 클래스 컨버터

        - - 페이징과 정렬을 위한 HandlerMethodArgumentResolver

    - **스프링 데이터 JPA가 사용하는 구현체**
    - **JPA 샵에 적용하기**
        - - 환경 설정
        - - 리파지토리 리펙토링
        - - 명세(Specification) 적용
    - **스프링 데이터 JPA와 QueryDSL 통합하기**
        - - QueryDslPredicateExecutor 사용
        - - QueryDslPredicateExecutor 한계
        - - QueryDslRepositorySupport 사용

스프링 데이터 JPA

대부분의 데이터 접근 계층(Data Access Layer)은 일명 CRUD로 부르는 유사한 등록, 수정, 삭제, 조회 코드를 반복해서 개발해야 한다. JPA를 사용해서 데이터 접근 계층을 개발할 때도 이 같은 문제가 발생한다.

====== JPA의 반복적인 CRUD ======

```
public class MemberRepository {

    @PersistenceContext
    EntityManager em;

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

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

}
```

```
public class ItemRepository {

    @PersistenceContext
    EntityManager em;

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

코드를 보면 회원 리파지토리(`MemberRepository`)와 상품 리파지토리(`ItemRepository`)가 하는 일이 비슷하다. 이런 문제를 해결하려면 제네릭과 상속을 적절히 사용해서 공통 부분을 처리하는 부모 클래스를 만들면 된다. 이것을 보통 `GenericDAO` 라 한다. 하지만 이 방법은 공통 기능을 구현한 부모 클래스에 너무 종속되고 구현 클래스 상속이 가지는 단점에 노출된다.

스프링 데이터 JPA 소개

스프링 데이터 JPA는 스프링 프레임워크에서 JPA를 편리하게 사용할 수 있도록 지원하는 프로젝트다.

이 프로젝트는 데이터 접근 계층을 개발할 때 지루하게 반복되는 CRUD 문제를 세련된 방법으로 해결한다. 우선 CRUD를 처리하기 위한 공통 인터페이스를 제공한다. 그리고 리파지토리를 개발할 때 인터페이스만 작성하면 실행 시점에 스프링 데이터 JPA가 구현 객체를 동적으로 생성해서 주입해준다. 따라서 **데이터 접근 계층을 개발할 때 구현 클래스 없이 인터페이스만 작성해도 개발을 완료할 수 있다.**

회원과 상품 리파지토리를 스프링 데이터 JPA를 사용해서 개발하면 다음과 같이 인터페이스만 작성하면 된다. CRUD를 처리하기 위한 공통 메서드는 스프링 데이터 JPA가 제공하는 `org.springframework.data.jpa.repository.JpaRepository` 인터페이스에 있다. 그리고 방금 언급했듯이 회원과 상품 리파지토리 인터페이스의 구현체는 애플리케이션 실행 시점에 스프링 데이터 JPA가 생성해서 주입해준다. 따라서 개발자가 직접 구현체를 개발하지 않아도 된다.

====== 스프링 데이터 JPA 적용 ======

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

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

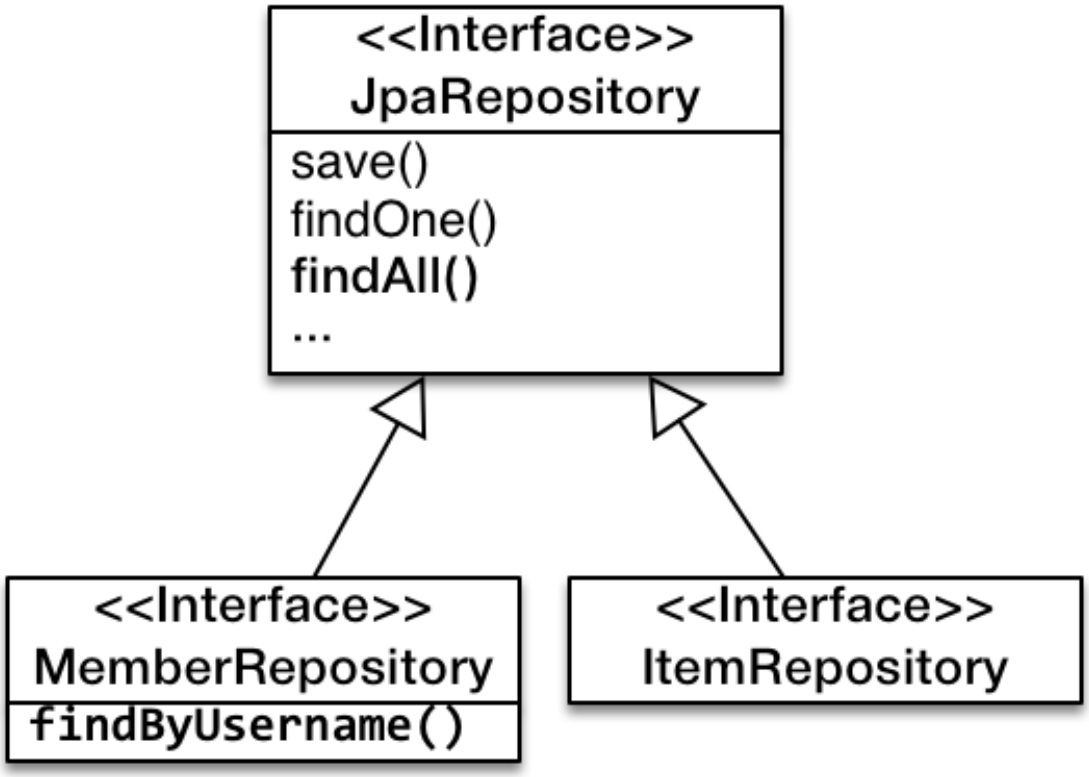

그림 - 스프링 데이터 JPA 사용

쿼리 메서드

일반적인 CRUD 메서드는 `JpaRepository` 인터페이스가 공통으로 제공하므로 문제가 없다. 그런데 `MemberRepository.findByUsername(...)` 처럼 직접 작성한 공통으로 처리할 수 없는 메서드는 어떻게 해야 할까? 놀랍게도 스프링 데이터 JPA는 메서드 이름을 분석해서 다음 JPQL을 실행한다.

```
`select m from Member m where username =:username`
```

- 스프링 데이터 프로젝트

스프링 데이터 JPA는 스프링 데이터 프로젝트의 하위 프로젝트 중 하나다.

스프링 데이터(Spring Data) 프로젝트는 JPA, MONGODB, NEO4J, REDIS, HADOOP, GEMFIRE 같은 다양한 데이터 저장소에 대한 접근을 추상화해서 다양한 개발자 편의를 제공하고 지루하게 반복하는 데이터 접근 코드를 줄여준다.

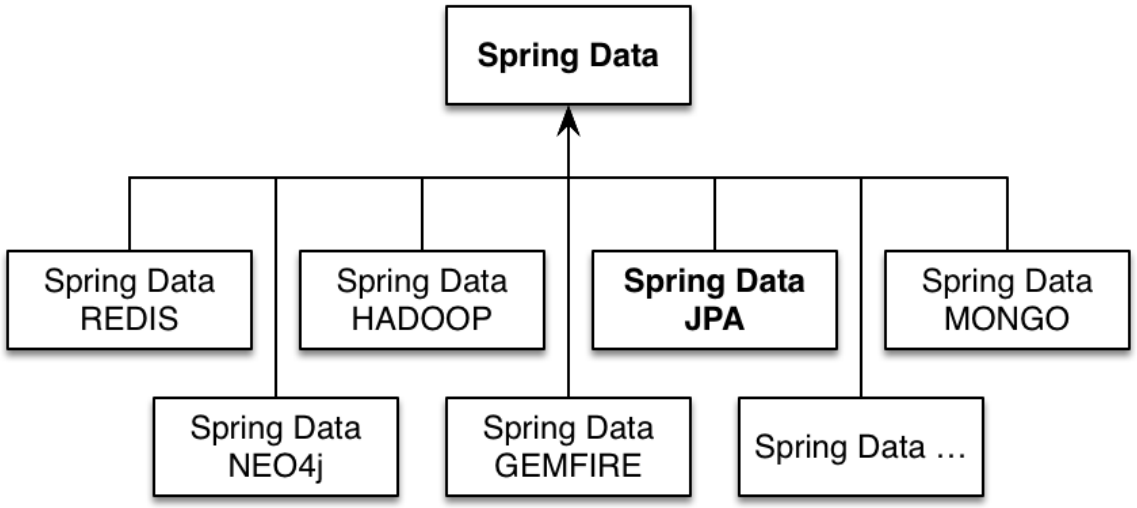

그림 - 스프링 데이터 연관 프로젝트

여기서 **스프링 데이터 JPA 프로젝트**는 JPA에 특화된 기능을 제공한다. 스프링 프레임워크와 JPA를 함께 사용한다면 스프링 데이터 JPA 사용을 적극 추천한다.

스프링 데이터 JPA 설정하기

- 필요 라이브러리

스프링 데이터 JPA는 `spring-data-jpa` 라이브러리가 필요하다. 예제에서는 : `1.8.0.RELEASE` 버전을 사용한다.

====== 스프링 데이터 JPA 메이븐 라이브러리 설정 ======

```
<!-- 스프링 데이터 JPA -->
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
    <version>1.8.0.RELEASE</version>
</dependency>
```

> **참고:** `spring-data-jpa` 는 `spring-data-commons` 에 의존하므로 두 라이브러리를 함께 받는다.

- 환경 설정

====== XML 설정 ======

스프링 설정에 XML을 사용하면 `<jpa:repositories>` 를 사용하고 리파지토리를 검색할 `base-package` 를 적는다. 참고로 해당 패키지와 그 하위 패키지를 검색한다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jpa="http://www.springframework.org/schema/data/jpa"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/jpa
    http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

  <jpa:repositories base-package="jpabook.jpashop.repository" />

</beans>
```

====== JavaConfig 설정 ======

스프링 설정에 JavaConfig를 사용하면 `org.springframework.data.jpa.repository.config.EnableJpaRepositories` 어노테이션을 추가하고 `basePackages` 에는 리파지토리를 검색할 패키지 위치를 적는다.

```
@Configuration
@EnableJpaRepositories(basePackages = "jpabook.jpashop.repository")
public class AppConfig {}
```

환경설정은 이것으로 끝이다.

스프링 데이터 JPA는 애플리케이션을 실행할 때 `basePackage` 에 있는 리파지토리 인터페이스들을 찾아서 해당 인터페이스를 구현한 클래스를 동적으로 생성한 다음 스프링 빈으로 등록한다. 따라서 개발자가 직접 구현 클래스를 만들지 않아도 된다.

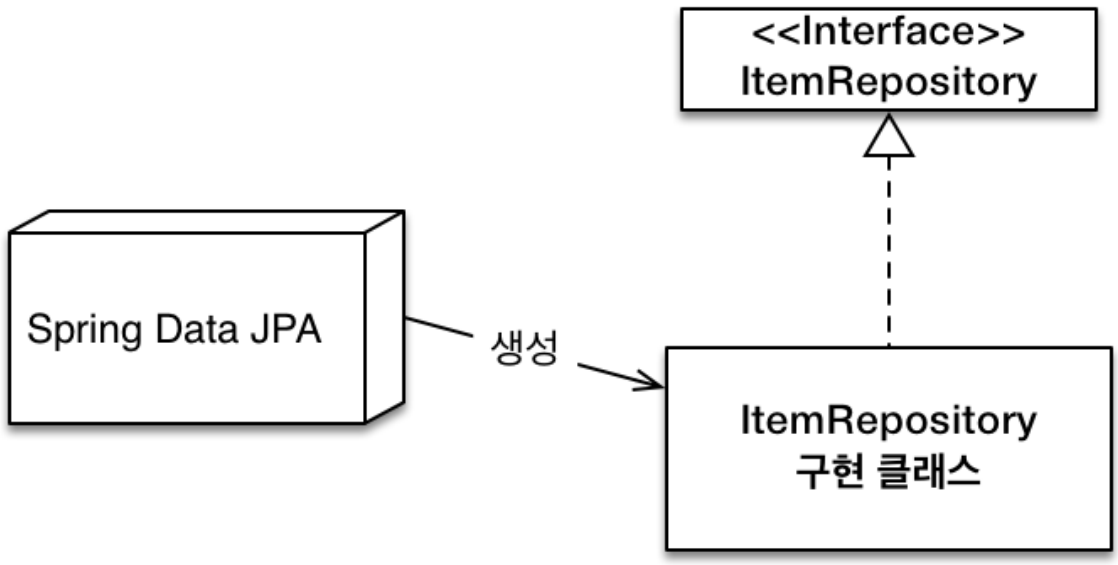

그림 - 구현 클래스 생성

공통 인터페이스 기능

스프링 데이터 JPA는 간단한 CRUD 기능을 공통으로 처리하는 `JpaRepository` 인터페이스를 제공한다. 스프링 데이터 JPA를 사용하는 가장 단순한 방법은 이 인터페이스를 상속받는 것이다. 그리고 제네릭에 엔티티 클래스와 엔티티 클래스가 사용하는 식별자 타입을 지정하면 된다.

====== `JpaRepository` 공통 기능 인터페이스 ======

```
public interface JpaRepository<T, ID extends Serializable> extends PagingAndSort
    ...
}
```

====== `JpaRepository` 를 사용하는 인터페이스 ======

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

상속받은 `JpaRepository<Member, Long>` 부분을 보면 제네릭에 회원 엔티티와 회원 엔티티의 식별자 타입을 지정했다. 이제부터 회원 리파지토리는 `JpaRepository` 인터페이스가 제공하는 다양한 기능을 사용할 수 있다.

`JpaRepository` 인터페이스의 계층 구조를 살펴보자.

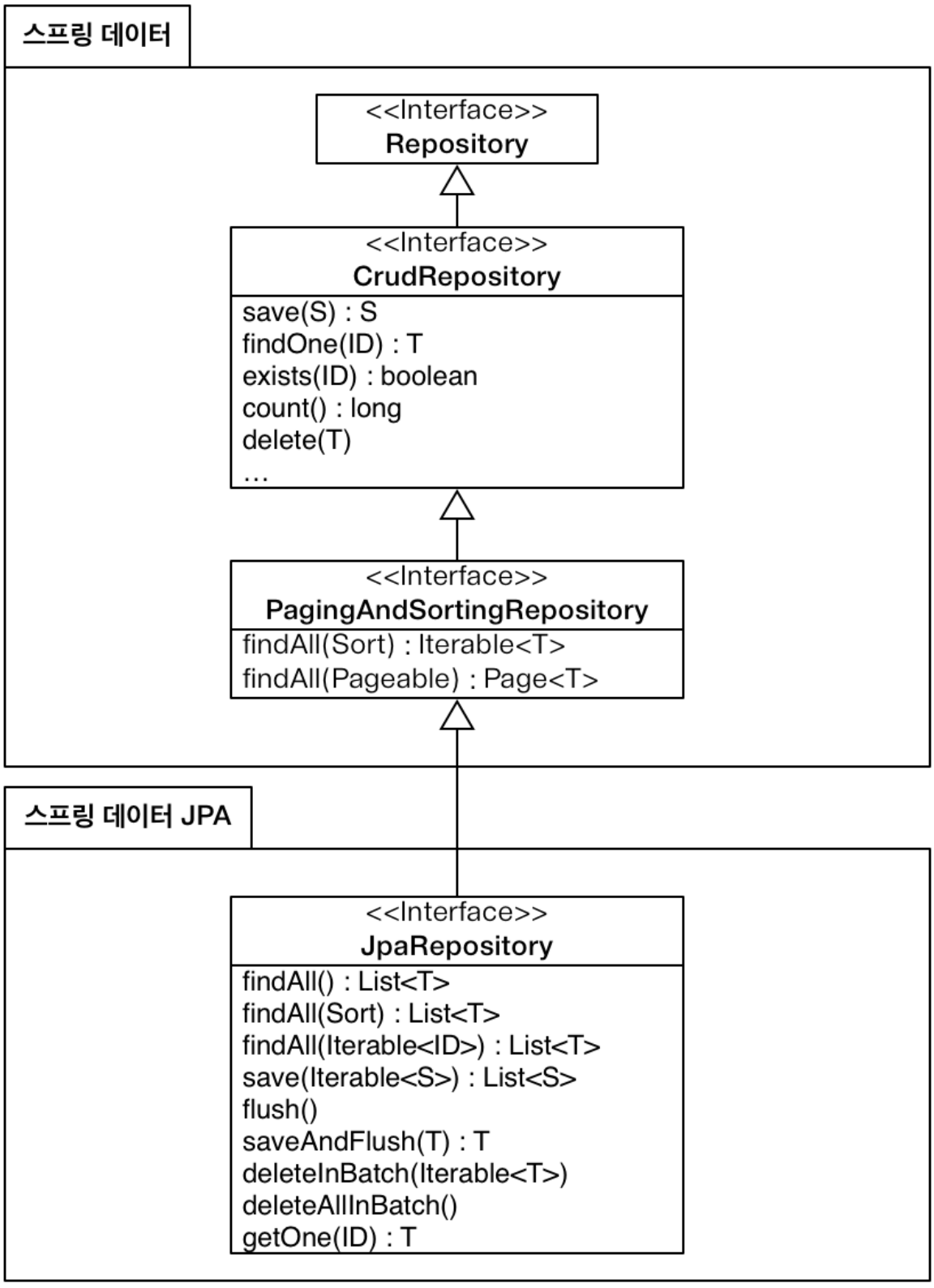

그림 - 공통 인터페이스 구성

[그림 - 공통 인터페이스 구성]을 보면 윗부분에 스프링 데이터 모듈이 있고 그안에 `Repository`, `CrudRepository`, `PagingAndSortingRepositry`가 있는데 이것은 스프링 데이터 프로젝트가 공통으로 사용하는 인터페이스다. 스프링 데이터 JPA가 제공하는 `JpaRepository` 인터페이스는 여기에 추가로 JPA에 특화된 기능을 제공한다.

`JpaRepository` 인터페이스를 상속받으면 사용할 수 있는 주요 메서드 몇 가지를 간단히 소개하겠다. 참고로 T는 엔티티, ID는 엔티티의 식별자 타입, S는 엔티티와 그 자식 타입을 뜻한다.

주요 메서드

- save(S): 새로운 엔티티는 저장하고 이미 있는 엔티티는 수정한다.
- delete(T): 엔티티 하나를 삭제한다. 내부에서 `EntityManager.remove()` 를 호출한다.
- findOne(ID): 엔티티 하나를 조회한다. 내부에서 `EntityManager.find()` 를 호출한다.
- getOne(ID): 엔티티를 프록시로 조회한다. 내부에서 `EntityManager.getReference()` 를 호출한다.
- findAll(…): 모든 엔티티를 조회한다. 정렬(`Sort`)이나 페이징(`Pageable`) 조건을 파라미터로 제공할 수 있다.

`save(S)` 메서드는 엔티티에 식별자 값이 없으면(`null` 이면) 새로운 엔티티로 판단해서 `EntityManager.persist()` 를 호출하고 식별자 값이 있으면 이미 있는 엔티티로 판단해서 `EntityManager.merge()` 를 호출한다. 필요하다면 스프링 데이터 JPA의 기능을 확장해서 신규 엔티티 판단 전략을 변경할 수 있다.

`JpaRepository` 공통 인터페이스를 사용하면 일반적인 CRUD를 해결할 수 있다. 다음은 쿼리 메서드 기능에 대해 알아보자.

쿼리 메서드 기능

쿼리 메서드 기능은 스프링 데이터 JPA가 제공하는 마법 같은 기능이다. 대표적으로 메서드 이름만으로 쿼리를 생성하는 기능이 있는데 인터페이스에 메서드만 선언하면 해당 메서드의 이름으로 적절한 JPQL 쿼리를 생성해서 실행한다.

스프링 데이터 JPA가 제공하는 쿼리 메서드 기능은 크게 3가지가 있다.

- 메서드 이름으로 쿼리 생성
- 메서드 이름으로 JPA NamedQuery 호출
- `@Query` 어노테이션을 사용해서 리파지토리 인터페이스에 쿼리 직접 정의

이 기능들을 활용하면 인터페이스만으로 필요한 대부분의 쿼리 기능을 개발할 수 있다. 순서대로 알아보자.

- 메서드 이름으로 쿼리 생성

이메일과 이름으로 회원을 조회하려면 다음과 같은 메서드를 정의하면 된다.

```
public interface MemberRepository extends Repository<Member, Long> {
    List<Member> findByEmailAndName(String email, String name);
}
```

인터페이스에 정의한 `findByEmailAndName(...)` 메서드를 실행하면 스프링 데이터 JPA는 메서드 이름을 분석해서 JPQL을 생성하고 실행한다.

====== 실행된 JPQL ======

```
select m from Member m where m.email = ?1 and m.name = ?2
```

물론 정해진 규칙에 따라서 메서드 이름을 지어야 한다. 스프링 데이터 JPA 공식 문서가 제공하는 표를 보면 이 기능을 어떻게 사용해야 하는지 쉽게 이해할 수 있다.

==== 쿼리 생성 기능 표(출처:스프링 데이터 JPA 공식 문서[1]) ====

키워드	예시	JP
And	findByLastnameAndFirstname	… x.l an ?2
Or	findByLastnameOrFirstname	… x.l x.f
Is,Equals	findByFirstname,findByFirstnameIs,findByFirstnameEquals	… x.f
Between	findByStartDateBetween	… x.s be 2
LessThan	findByAgeLessThan	… 1

LessThanEqual	findByAgeLessThanEqual	… ?1
GreaterThan	findByAgeGreaterThan	… 1
GreaterThanEqual	findByAgeGreaterThanEqual	… ?1
After	findByStartDateAfter	… x.s
Before	findByStartDateBefore	… x.s
IsNull	findByAgeIsNull	… nu
IsNotNull,NotNull	findByAge(Is)NotNull	… nu
Like	findByFirstnameLike	… x.f
NotLike	findByFirstnameNotLike	… x.f ?1
StartingWith	findByFirstnameStartingWith	… x.f 1 (bo ap
EndingWith	findByFirstnameEndingWith	… x.f 1 (bo pre

Containing	findByFirstnameContaining	… x.f 1 (bo in
OrderBy	findByAgeOrderByLastnameDesc	… 1 c x.l
Not	findByLastnameNot	… x.l
In	findByAgeIn(Collection ages)	… ?1
NotIn	findByAgeNotIn(Collection age)	… in
TRUE	findByActiveTrue()	… = t
FALSE	findByActiveFalse()	… = f
IgnoreCase	findByFirstnameIgnoreCase	… UP = U

참고로 이 기능은 엔티티의 필드명이 변경되면 인터페이스에 정의한 메서드 이름도 꼭 함께 변경해 주어야 한다.

- JPA NamedQuery

JPA Named 쿼리는 이름 그대로 쿼리에 이름을 부여해서 사용하는 방법인데 어노테이션이나 XML에 쿼리를 정의할 수 있다. 상세한 내용은 JPQL 장의 Named 쿼리를 참고하자. 그리고 같은 방법으로 Named 네이티브 쿼리도 지원한다.

스프링 데이터 JPA는 메서드 이름으로 JPA Named 쿼리를 호출하는 기능을 제공한다.

====== `@NamedQuery` 어노테이션으로 Named 쿼리 정의 ======

```
@Entity
@NamedQuery(
        name="Member.findByUsername",
        query="select m from Member m where m.username = :username")
public class Member {
    ...
}
```

====== `orm.xml`의 XML 사용 ======

```
<named-query name="Member.findByUsername">
    <query><![CDATA[
        select m
        from Member m
        where m.username = :username
    ]]></query>
</named-query>
```

이렇게 정의한 Named 쿼리를 JPA에서 직접 호출하려면 다음과 같은 코드를 작성해야 한다.

====== JPA를 직접 사용해서 Named 쿼리 호출 ======

```
public class MemberRepository {

    public List<Member> findByUsername(String username) {
        ...
        List<Member> resultList = em.createNamedQuery("Member.findByUsername", M
                .setParameter("username", "회원1")
                .getResultList();
    }
}
```

스프링 데이터 JPA를 사용하면 다음처럼 메서드 이름만으로 Named 쿼리를 호출할 수 있다.

====== 스프링 데이터 JPA로 Named 쿼리 호출 ======

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

    List<Member> findByUsername(@Param("username") String username); //**
```

```
}
```

스프링 데이터 JPA는 선언한 “도메인 클래스 + .(점) + 메서드 이름”으로 Named 쿼리를 찾아서 실행한다. 따라서 예제는 `Member.findByUsername`이라는 Named 쿼리를 실행한다.

만약 실행할 Named 쿼리가 없으면 메서드 이름으로 쿼리 생성 전략을 사용한다. (필요하면 전략을 변경할 수 있다.)

참고로 이름기반 파라미터 바인딩은 `org.springframework.data.repository.query.Param` 어노테이션을 사용한다.

- @Query, 리파지토리 메서드에 쿼리 정의하기

리파지토리 메서드에 직접 쿼리를 정의하려면 `@org.springframework.data.jpa.repository.Query` 어노테이션을 사용한다. 이 방법은 실행할 메서드에 정적 쿼리를 직접 작성하므로 이름 없는 Named 쿼리라 할 수 있다. 또한 JPA Named 쿼리처럼 애플리케이션 실행 시점에 문법 오류를 발견할 수 있는 장점이 있다.

====== 메서드에 JPQL 쿼리 작성 ======

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

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

네이티브 SQL을 사용하려면 `@Query` 어노테이션에 `nativeQuery = true`를 설정한다. 참고로 스프링 데이터 JPA가 지원하는 파리미터 바인딩을 사용하면 JPQL은 위치 기반 파리미터를 1부터 시작하지만 네이티브 SQL은 0부터 시작한다.

====== JPA 네이티브 SQL 지원 ======

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

    @Query(value = "SELECT * FROM MEMBER WHERE USERNAME = ?0", nativeQuery = tru
    Member findByUsername(String username);
}
```

- 파라미터 바인딩

스프링 데이터 JPA는 위치 기반 파라미터 바인딩과 이름 기반 파라미터 바인딩을 모두 지원한다. 기본 값은 위치 기반인데 파라미터 순서로 바인딩한다. 이름 기반 파라미터 바인딩을 사용하려면 `org.springframework.data.repository.query.Param(파라미터 이름)` 어노테이션을 사용하면 된다. 코드 가독성과 유지보수를 위해 **이름 기반 파라미터 바인딩을 사용하자.**

```
select m from Member m where m.username = ?1 //위치 기반
select m from Member m where m.username = :name //이름 기반
```

```
import org.springframework.data.repository.query.Param

public interface MemberRepository extends JpaRepository<Member, Long> {

    @Query("select m from Member m where m.username = :name")
    Member findByUsername(@Param("name") String username);
}
```

- 벌크성 수정 쿼리

먼저 JPA로 작성한 벌크성 수정 쿼리부터 보자.

====== JPA를 사용한 벌크성 수정 쿼리 ======

```
int bulkPriceUp(String stockAmount){
    ...
    String qlString =
            "update Product p set p.price = p.price * 1.1 where p.stockAmount <

    int resultCount = em.createQuery(qlString)
                        .setParameter("stockAmount", stockAmount)
                        .executeUpdate(); //**
}
```

다음으로 스프링 데이터 JPA를 사용한 벌크성 수정 쿼리를 보자.

====== 스프링 데이터 JPA를 사용한 벌크성 수정 쿼리 ======

```
@Modifying
@Query("update Product p set p.price = p.price * 1.1 where p.stockAmount < :stoc
int bulkPriceUp(@Param("stockAmount") String stockAmount);
```

스프링 데이터 JPA에서 벌크성 수정, 삭제 쿼리는 `org.springframework.data.jpa.repository.Modifying` 어노테이션을 사용하면 된다.

벌크성 쿼리를 실행하고 나서 영속성 컨텍스트를 초기화 하고 싶으면 `@Modifying(clearAutomatically = true)` 처럼 `clearAutomatically` 옵션을 `true` 로 설정하면 된다. 참고로 이 옵션의 기본값은 `false` 다.

- 반환 타입

스프링 데이터 JPA는 유연한 반환 타입을 지원하는데 결과가 한 건 이상이면 컬렉션 인터페이스를 사용하고, 단건이면 반환 타입을 지정한다.

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

만약 조회 결과가 없으면 컬렉션은 빈 컬렉션을 반환하고 단건은 `null` 을 반환한다. 그리고 단건을 기대하고 반환 타입을 지정했는데 결과가 2건 이상 조회되면 `javax.persistence.NonUniqueResultException` 예외가 발생한다.

참고로 단건으로 지정한 메서드를 호출하면 스프링 데이터 JPA는 내부에서 JPQL의 `Query.getSingleResult()` 메서드를 호출한다. 이 메서드를 호출했을 때 조회 결과가 없으면 `javax.persistence.NoResultException` 예외가 발생하는데 개발자 입장에서 다루기가 상당히 불편하다. 스프링 데이터 JPA는 단건을 조회할 때 이 예외가 발생하면 예외를 무시하고 대신에 `null` 을 반환한다.

- 페이징과 정렬

스프링 데이터 JPA는 쿼리 메서드에 페이징과 정렬 기능을 사용할 수 있도록 2가지 특별한 파라미터를 제공한다.

- `org.springframework.data.domain.Sort` : 정렬 기능
- `org.springframework.data.domain.Pageable` : 페이징 기능 (내부에 `Sort` 포함)

파리미터에 `Pageable`을 사용하면 반환 타입으로 `List`나 `org.springframework.data.domain.Page`를 사용할 수 있다. 반환 타입으로 `Page`를 사용하면 스프링 데이터 JPA는 페이징 기능을 제공하기 위해 검색된 전체 데이터 건수를 조회하는 count 쿼리를 추가로 호출한다.

====== 페이징과 정렬 사용 예제 ======

```
Page<Member> findByName(String name, Pageable pageable); //count 쿼리 사용
List<Member> findByName(String name, Pageable pageable); //count 쿼리 사용 안함
List<Member> findByName(String name, Sort sort);
```

다음 조건으로 페이징과 정렬을 사용하는 예제 코드를 보자.

- 검색 조건: 이름이 김으로 시작하는 회원
- 정렬 조건: 이름으로 내림차순
- 페이징 조건: 첫 번째 페이지, 페이지당 보여줄 데이터는 10건

====== Page 사용 예제 ======

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

    Page<Member> findByNameStartingWith(String name, Pageable Pageable);
}
```

```
//페이징 조건과 정렬 조건 설정
PageRequest pageRequest = new PageRequest(0, 10, new Sort(Direction.DESC, "name'

Page<Member> result = memberRepository.findByNameStartingWith("김", pageRequest)

List<Member> members = result.getContent(); //조회된 데이터
int totalPages = result.getTotalPages(); //전체 페이지 수
boolean hasNextPage = result.hasNextPage(); //다음 페이지 존재 여부
```

`Pagable`은 인터페이스다. 따라서 실제 사용할 때는 해당 인터페이스를 구현한 `org.springframework.data.domain.PageRequest` 객체를 사용한다. `PageRequest` 생성자의 첫 번째 파라미터에는 현재 페이지를, 두 번째 파라미터에는 조회할 데이터 수를 입력한다. 여기에 추가로 정렬 정보도 파라미터로 사용할 수 있다. 참고로 페이지는 0부터 시작한다.

====== Page 인터페이스 ======

```
public interface Page<T> extends Iterable<T> {

    int getNumber();              //현재 페이지
    int getSize();                //페이지 크기
    int getTotalPages();          //전체 페이지 수
    int getNumberOfElements();    //현재 페이지에 나올 데이터 수
    long getTotalElements();      //전체 데이터 수
    boolean hasPreviousPage();    //이전 페이지 여부
    boolean isFirstPage();        //현재 페이지가 첫 페이지 인지 여부
    boolean hasNextPage();        //다음 페이지 여부
    boolean isLastPage();         //현재 페이지가 마지막 페이지 인지 여부
    Pageable nextPageable();      //다음 페이지 객체, 다음 페이지가 없으면 null
    Pageable previousPageable();//다음 페이지 객체, 이전 페이지가 없으면 null
    List<T> getContent();         //조회된 데이터
    boolean hasContent();         //조회된 데이터 존재 여부
    Sort getSort();               //정렬 정보
}
```

`Pageable`과 `Page`를 사용하면 지루하고 반복적인 페이징 처리를 손쉽게 개발할 수 있다.

- 힌트

JPA 쿼리 힌트를 사용하려면 `org.springframework.data.jpa.repository.QueryHints` 어노테이션을 사용하면 된다. 참고로 이것은 SQL 힌트가 아니라 JPA 구현체에게 제공하는 힌트다.

```
@QueryHints(value = { @QueryHint(name = "org.hibernate.readOnly", value = "true"
Page<Member> findByName(String name, Pagable pageable);
```

`forCounting` 속성은 반환 타입으로 `Page` 인터페이스를 적용하면 추가로 호출하는 페이징을 위한 count 쿼리에도 쿼리 힌트를 적용할지를 설정하는 옵션이다. (기본값 `true`)

- Lock

쿼리 시 락을 걸려면 `org.springframework.data.jpa.repository.Lock` 어노테이션을 사용하면 된다. JPA가 제공하는 락에 대해서는 고급주제를 참고하자.

```
@Lock(LockModeType.PESSIMISTIC_WRITE)
List<Member> findByName(String name);
```

Specifications (명세)

책 도메인 주도 설계(Domain Driven Design)는 SPECIFICATION(명세)라는 개념을 소개하는데 스프링 데이터 JPA는 JPA Criteria로 이 개념을 사용할 수 있도록 지원한다.

명세(Specification)을 이해하기 위한 핵심 단어는 술어(predicate)인데 이것은 단순히 참이나 거짓으로 평가된다. 그리고 이것은 AND, OR 같은 연산자로 조합할 수 있다. 예를 들어 데이터를 검색하기 위한 제약 조건 하나하나를 술어라 할 수 있다. 이 술어를 스프링 데이터 JPA는 `org.springframework.data.jpa.domain.Specification` 클래스로 정의했다.

`Specification`은 컴포지트 패턴(Composite pattern)[2]으로 구성되어 있어서 여러 `Specification`을 조합할 수 있다. 따라서 다양한 검색조건을 조립해서 새로운 검색조건을 쉽게 만들 수 있다.

명세 기능을 사용하려면 리파지토리에서 `org.springframework.data.jpa.repository.JpaSpecificationExecutor` 인터페이스를 상속 받으면 된다.

====== `JpaSpecificationExecutor` 상속 ======

```
public interface OrderRepository extends JpaRepository<Order, Long>, JpaSpecific

}
```

====== `JpaSpecificationExecutor` 인터페이스 ======

```
public interface JpaSpecificationExecutor<T> {

    T findOne(Specification<T> spec);
    List<T> findAll(Specification<T> spec);
    Page<T> findAll(Specification<T> spec, Pageable pageable);
    List<T> findAll(Specification<T> spec, Sort sort);
    long count(Specification<T> spec);
}
```

`JpaSpecificationExecutor`의 메서드들은 `Specification`을 파라미터로 받아서 검색 조건으로 사용한다.

이제 명세를 사용하는 예제를 보자. 우선 명세를 사용하는 코드를 보고나서 명세를 정의하는 코드를 보자.

====== 명세 사용 코드 ======

```
import static org.springframework.data.jpa.domain.Specifications.*; //where()
import static jpabook.jpashop.domain.spec.OrderSpec.*; //**

public List<Order> findOrders(String name) {

    List<Order> result = orderRepository.findAll(
            where(memberName(name)).and(isOrderStatus()) //**
    );

    return result;
}
```

`Specifications`는 명세들을 조립할 수 있도록 도와주는 클래스인데 `where()`, `and()`, `or()`, `not()` 메서드를 제공한다.

`findAll`을 보면 회원 이름 명세(`memberName`)와 주문 상태 명세(`isOrderStatus`)를 `and`로 조합해서 검색 조건으로 사용한다.

참고로 명세 기능을 사용할 때 예제처럼 자바의 `import static`를 적용하면 더 읽기 쉬운 코드가 된다.

이제 `OrderSpec` 명세를 정의하는 코드를 보자.

====== `OrderSpec` 명세 정의 코드 ======

```
package jpabook.jpashop.domain;

import org.springframework.data.jpa.domain.Specification;
import org.springframework.util.StringUtils;
import javax.persistence.criteria.*;

public class OrderSpec {

    public static Specification<Order> memberName(final String memberName) {
        return new Specification<Order>() {
            public Predicate toPredicate(Root<Order> root, CriteriaQuery<?> quer

                if (StringUtils.isEmpty(memberName)) return null;
```

```
                Join<Order, Member> m = root.join("member", JoinType.INNER); //
                return builder.equal(m.get("name"), memberName);
            }
        };
    }

    public static Specification<Order> isOrderStatus() {
        return new Specification<Order>() {
            public Predicate toPredicate(Root<Order> root, CriteriaQuery<?> quer

                return builder.equal(root.get("status"), OrderStatus.ORDER);
            }
        };
    }
}
```

명세를 정의하려면 `Specification` 인터페이스를 구현하면 된다. 예제에서는 편의상 내부 무명 클래스를 사용했다. 명세를 정의할 때는 `toPredicate(...)` 메서드만 구현하면 되는데 JPA Criteria의 `Root`, `CriteriaQuery`, `CriteriaBuilder` 클래스가 모두 파라미터로 주어진다. 이 파라미터들을 활용해서 적절한 검색 조건을 반환하면 된다. JPA Criteria에 대한 이해가 부족하다면 JPA Criteria 편을 참고하자.

사용자 정의 리파지토리 구현

스프링 데이터 JPA로 리파지토리를 개발하면 인터페이스만 정의하고 구현체는 만들지 않는다. 하지만 다양한 이유로 메서드를 직접 구현해야 할 때도 있다. 그렇다고 리파지토리를 직접 구현하면 공통 인터페이스가 제공하는 기능까지 모두 구현해야 한다. 스프링 데이터 JPA는 이런 문제를 우회해서 필요한 메서드만 구현할 수 있는 방법을 제공한다.

먼저 직접 구현할 메서드를 위한 사용자 정의 인터페이스를 작성해야 한다. 이때 인터페이스 이름은 자유롭게 지으면 된다.

====== 사용자 정의 인터페이스 ======

```
public interface MemberRepositoryCustom {
    public List<Member> findMemberCustom();
}
```

다음으로 사용자 정의 인터페이스를 구현한 클래스를 작성해야 한다. 이때 클래스 이름을 짓는 규칙이 있는데 리파지토리 인터페이스 이름 + `Impl` 로 지어야 한다. 이렇게 하면 스프링 데이터 JPA가 사용자 정의 구현 클래스로 인식한다.

====== 사용자 정의 구현 클래스 ======

```
public class MemberRepositoryImpl implements MemberRepositoryCustom {

    @Override
    public List<Member> findMemberCustom() {
        ... //사용자 정의 구현
    }
}
```

마지막으로 리파지토리 인터페이스에서 사용자 정의 인터페이스를 상속받으면 된다.

```
public interface MemberRepository extends JpaRepository<Member, Long>, MemberRep
}
```

만약 사용자 정의 구현 클래스 이름 끝에 `Impl` 대신 다른 이름을 붙이고 싶으면 `repository-impl-postfix` 속성을 변경하면 된다. 참고로 `Impl` 이 기본값이다.

====== XML ======

```
<repositories base-package="jpabook.jpashop.repository" repository-impl-postfix=
```

====== JavaConfig ======

```
@EnableJpaRepositories(basePackages = "jpabook.jpashop.repository", repositoryIm
```

Web 확장

스프링 데이터 프로젝트는 스프링 MVC에서 사용할 수 있는 편리한 기능을 제공한다.

- 설정

스프링 데이터가 제공하는 Web 확장 기능을 활성화하려면 `org.springframework.data.web.config.SpringDataWebConfiguration`를 스프링 빈으로 등록하면 된다.

====== XML ======

```
<bean class="org.springframework.data.web.config.SpringDataWebConfiguration" />
```

JavaConfig를 사용하면 `org.springframework.data.web.config.EnableSpringDataWebSupport` 어노테이션을 사용하면 된다.

====== JavaConfig ======

```
@Configuration
@EnableWebMvc
@EnableSpringDataWebSupport //**
public class WebAppConfig {
    ...
}
```

설정을 완료하면 도메인 클래스 컨버터와 페이징과 정렬을 위한 `HandlerMethodArgumentResolver`가 스프링 빈으로 등록된다.

등록되는 도메인 클래스 컨버터 : `org.springframework.data.repository.support.DomainClassConverter`

- 도메인 클래스 컨버터

도메인 클래스 컨버터는 HTTP 파라미터로 넘어온 엔티티의 아이디로 엔티티 객체를 찾아서 바인딩 해준다.

예를 들어 특정 회원을 수정하는 화면을 보여주려면 컨트롤러는 HTTP 요청으로 넘어온 회원의 아이디를 사용해서 리파지토리를 통해 회원 엔티티를 조회해야 한다. 다음과 같은 URL을 호출했다고 가정하자.

수정화면 요청 URL: `/member/memberUpdateForm?id=1`

====== 회원의 아이디로 회원 엔티티 조회 ======

```
@Controller
public class MemberController {

    @Autowired MemberRepository memberRepository;

    @RequestMapping("member/memberUpdateForm")
    public String memberUpdateForm(@RequestParam("id") Long id, Model model) {

        Member member = memberRepository.findOne(id); //회원을 찾는다.
        model.addAttribute("member", member);
        return "member/memberSaveForm";
    }
}
```

코드를 보면 컨트롤러에서 파라미터로 넘어온 회원 아이디로 회원 엔티티를 찾는다. 그리고 찾아온 회원 엔티티를 `model`을 사용해서 뷰에 넘겨준다.

이번에는 도메인 클래스 컨버터를 적용한 코드를 보자.

====== 도메인 클래스 컨버터 적용 ======

```
@Controller
public class MemberController {

    @RequestMapping("member/memberUpdateForm")
    public String memberUpdateForm(@RequestParam("id") Member member, Model mode
        model.addAttribute("member", member);
        return "member/memberSaveForm";
    }
}
```

`@RequestParam("id") Member member` 부분을 보면 HTTP 요청으로 회원 아이디(`id`)를 받지만 도메인 클래스 컨버터가 중간에 동작해서 아이디를 회원 엔티티 객체로 변환해서 넘겨준다. 따라서 컨트롤러를 단순하게 사용할 수 있다.

참고로 도메인 클래스 컨버터는 해당 엔티티와 관련된 리파지토리를 사용해서 엔티티를 찾는다. 여기서는 회원 리파지토리를 통해서 회원 아이디로 회원 엔티티를 찾는다.

> **주의:** 도메인 클래스 컨버터를 통해 넘어온 회원 엔티티를 컨트롤러에서 직접 수정해도 실제 데이터베이스에는 반영되지 않는다. 참고로 이것은 스프링 데이터와는 관련이 없고 순전히 영속성 컨텍스트의 동작 방식과 관련이 있다. OSIV는 다음 웹 애플리케이션 심화편에서 설

> 명한다.
>
> - **OSIV를 사용하지 않으면:** 조회한 엔티티는 준영속 상태다. 따라서 변경 감지기능이 동작하지 않는다. 만약 수정한 내용을 데이터베이스에 반영하고 싶으면 병합(merge)를 사용해야 한다.
> - **OSIV를 사용하면:** 조회한 엔티티는 영속 상태다. 하지만 OSIV의 특성상 컨트롤러와 뷰에서는 영속성 컨텍스트를 플러시 하지 않는다. 따라서 수정한 내용을 데이터베이스 반영하지 않는다. 만약 수정한 내용을 데이터베이스에 반영하고 싶으면 트랜잭션을 시작하는 서비스 계층을 호출해야 한다. 해당 서비스 계층이 종료될 때 플러시와 트랜잭션 커밋이 일어나서 영속성 컨텍스트의 변경 내용을 데이터베이스에 반영해 줄 것이다.

- 페이징과 정렬을 위한 `HandlerMethodArgumentResolver`

스프링 데이터가 제공하는 페이징과 정렬 기능을 스프링 MVC에서 편리하게 사용할 수 있도록 `HandlerMethodArgumentResolver`를 제공한다.

- 페이징 기능: `PageableHandlerMethodArgumentResolver`
- 정렬 기능 : `SortHandlerMethodArgumentResolver`

바로 예제를 보자.

====== 예제 ======

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

    Page<Member> page = memberService.findMembers(pageable);
    model.addAttribute("members", page.getContent());
    return "members/memberList";
}
```

파라미터로 `Pageable`을 바로 받은 것을 확인할 수 있다. `Pageable`은 다음 요청 파라미터 정보로 만들어진다. (`Pageable`은 인터페이스다 실제는 `org.springframework.data.domain.PageRequest` 객체가 생성된다.)

요청 파라미터

- page: 현재 페이지, 0부터 시작

- size: 한 페이지에 노출할 데이터 건수
- sort: 정렬 조건을 정의한다. 예) 정렬 속성,정렬 속성…(ASC | DESC), 정렬 방향을 변경하고 싶으면 `sort` 파라미터를 추가하면 된다.

예제: `/members?page=0&size=20&sort=name,desc&sort=address.city`

> **참고:** 페이지를 1부터 시작하고 싶으면 `PageableHandlerMethodArgumentResolver`를 스프링 빈으로 직접 등록하고 `setOneIndexedParameters`를 `true`로 설정하면 된다.

접두사

사용해야할 페이징 정보가 둘 이상이면 접두사를 사용해서 구분할 수 있다. 접두사는 스프링 프레임워크가 제공하는 `@Qualifier` 어노테이션을 사용한다. 그리고 "{접두사명}_" 로 구분한다.

```
public String list(
    @Qualifier("member") Pageable memberPageable,
    @Qualifier("order") Pageable orderPageable, ...
```

예제: `/members?member_page=0&order_page=1`

기본값

`Pageable`의 기본값은 `page=0`, `size=20`이다. 만약 기본값을 변경하고 싶으면 `@PageableDefault` 어노테이션을 사용하면 된다.

```
@RequestMapping(value = "/members_page", method = RequestMethod.GET)
public String list(@PageableDefault(size = 12, sort = "name", direction = Sort.D
    ...
}
```

스프링 데이터 JPA가 사용하는 구현체

스프링 데이터 JPA가 제공하는 공통 인터페이스는 `org.springframework.data.jpa.repository.support.SimpleJpaRepository` 클래스가 구현한다. 코드 일부를 분석해보자.

====== SimpleJpaRepository ======

```
@Repository //**
@Transactional(readOnly = true) //**
public class SimpleJpaRepository<T, ID extends Serializable> implements JpaRepos
        JpaSpecificationExecutor<T> {

    @Transactional //**
    public <S extends T> S save(S entity) {

        if (entityInformation.isNew(entity)) {
            em.persist(entity);
            return entity;
        } else {
            return em.merge(entity);
        }
    }
    ...
}
```

`@Repository` 적용: JPA 예외를 스프링이 추상화한 예외로 변환한다.

`@Transactional` 트랜잭션 적용: JPA의 모든 변경은 트랜잭션 안에서 이루어져야 한다. 스프링 데이터 JPA가 제공하는 공통 인터페이스를 사용하면 데이터를 변경(등록, 수정, 삭제)하는 메서드에 `@Transactional`로 트랜잭션 처리가 되어있다. 따라서 서비스 계층에서 트랜잭션을 시작하지 않으면 리파지토리에서 트랜잭션을 시작한다. 물론 서비스 계층에서 트랜잭션을 시작했으면 리파지토리도 해당 트랜잭션을 전파 받아 그대로 사용한다.

`@Transactional(readOnly = true)` : 데이터를 조회하는 메서드에는 `readOnly = true` 옵션이 적용되어 있다. 데이터를 변경하지 않는 트랜잭션에서 `readOnly = true` 옵션을 사용하면 플러시를 생략해서 약간의 성능 향상을 얻을 수 있는데 자세한 내용은 고급 주제에서 설명하겠다.

`save()` 메서드: 이 메서드는 저장할 엔티티가 새로운 엔티티면 저장(`persist`)하고 이미 있는 엔티티면 병합(`merge`)한다.
새로운 엔티티를 판단하는 기본 전략은 엔티티의 식별자로 판단하는데 식별자가 객체일 때 `null` , 자바 기본 타입일 때 숫자 0 값이면 새로운 엔티티로 판단한다. 필요하면 엔티티에 `Persistable` 인터페이스를 구현해서 판단 로직을 변경할 수 있다.

====== Persistable ======

```
package org.springframework.data.domain;
```

```
public interface Persistable<ID extends Serializable> extends Serializable {
    ID getId();
    boolean isNew();
}
```

이것으로 스프링 데이터 JPA를 알아보았다. 이제 앞장에서 만든 웹 애플리케이션에 스프링 데이터 JPA를 적용해보자.

JPA 샵에 적용하기

> 예제 코드 : `ch18-springdata-shop`

스프링 프레임워크와 JPA로 개발한 웹 애플리케이션에 스프링 데이터 JPA를 적용해보자. 다음 순서대로 코드를 변경하면서 스프링 데이터 JPA가 얼마나 유용한지 확인해보자.

- 환경 설정
- 리파지토리 리펙토링
- 명세(Specification) 적용
- 기타

> 참고: 완성된 프로젝트는 `ch18-springdata-shop`이다. 실행 방법은 `ch17-jpa-shop`과 같다.

- 환경 설정

`pom.xml`에 `spring-data-jpa` 라이브러리를 추가하자. 예제에서는 `1.8.0.RELEASE` 버전을 사용한다.

```
<!-- 스프링 데이터 JPA -->
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
    <version>1.8.0.RELEASE</version>
</dependency>
```

====== XML 설정 ======

`src/main/resources/appConfig.xml`에 `<jpa:repositories>`를 추가하고 `base-package` 속성에 리파지토리 위치를 지정하자.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jpa="http://www.springframework.org/schema/data/jpa"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/jpa
    http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

    <!-- 추가 -->
    <jpa:repositories base-package="jpabook.jpashop.repository" />

    ...

</beans>
```

이것으로 스프링 데이터 JPA를 사용할 준비가 끝났다.

- 리파지토리 리펙토링

지금부터 본격적으로 리파지토리들이 스프링 데이터 JPA를 사용하도록 리펙토링 해보자.

회원 리파지토리 리펙토링

====== 회원 리파지토리 리펙토링 전 ======

```
package jpabook.jpashop.repository;

@Repository
public class MemberRepository {

    @PersistenceContext
    EntityManager em;

    public void save(Member member) {
        em.persist(member);
    }

    public Member fineOne(Long id) {
        return em.find(Member.class, id);
    }
```

```
    public List<Member> findAll() {
        return em.createQuery("select m from Member m", Member.class)
                .getResultList();
    }

    public List<Member> findByName(String name) {
        return em.createQuery("select m from Member m where m.name = :name", Mem
                .setParameter("name", name)
                .getResultList();
    }
}
```

====== 회원 리파지토리 리펙토링 후 ======

```
package jpabook.jpashop.repository;

import jpabook.jpashop.domain.Member;
import org.springframework.data.jpa.repository.JpaRepository;
import java.util.List;

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

회원 리파지토리 리펙토링 후 코드를 보자. 우선 클래스를 인터페이스로 변경하고 스프링 데이터 JPA가 제공하는 `JpaRepository`를 상속 받았다. 이때 제네릭 타입을 `<Member, Long>`으로 지정해서 리파지토리가 관리하는 엔티티 타입과 엔티티의 식별자 타입을 정의했다.

코드를 보면 `save()`, `fineOne()`, `findAll()` 메서드를 제거했다. 이런 기본 메서드는 상속받은 `JpaRepository`가 모두 제공한다. 남겨진 메서드는 `findByName()`인데 스프링 데이터 JPA가 해당 메서드의 이름을 분석해서 메서드 이름으로 적절한 쿼리를 실행해 줄 것이다.

다음으로 상품 리파지토리를 리펙토링 해보자.

상품 리파지토리 리펙토링

====== 상품 리파지토리 리펙토링 전 ======

```
package jpabook.jpashop.repository;

@Repository
public class ItemRepository {
```

```
    @PersistenceContext
    EntityManager em;

    public void save(Item item) {
        if (item.getId() == null) {
            em.persist(item);
        } else {
            em.merge(item);
        }
    }

    public Item fineOne(Long id) {
        return em.find(Item.class, id);
    }

    public List<Item> findAll() {
        return em.createQuery("select i from Item i",Item.class).getResultList()
    }
}
```

====== 상품 리파지토리 리펙토링 후 ======

```
package jpabook.jpashop.repository;

import jpabook.jpashop.domain.item.Item;
import org.springframework.data.jpa.repository.JpaRepository;

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

상품 리파지토리가 제공하는 모든 기능은 스프링 데이터 JPA가 제공하는 공통 인터페이스만으로 충분하다.

마지막으로 주문 리파지토리를 리펙토링해보자.

주문 리파지토리 리펙토링

====== 주문 리파지토리 리펙토링 전 ======

```
package jpabook.jpashop.repository;

@Repository
public class OrderRepository {
```

```
    @PersistenceContext
    EntityManager em;

    public void save(Order order) {
        em.persist(order);
    }

    public Order fineOne(Long id) {
        return em.find(Order.class, id);
    }

    public List<Order> findAll(OrderSearch orderSearch) {

        CriteriaBuilder cb = em.getCriteriaBuilder();
        CriteriaQuery<Order> cq = cb.createQuery(Order.class);
        Root<Order> o = cq.from(Order.class);

        List<Predicate> criteria = new ArrayList<Predicate>();

        //주문 상태 검색
        if (orderSearch.getOrderStatus() != null) {
            Predicate status = cb.equal(o.get("status"), orderSearch.getOrderSta
            criteria.add(status);
        }
        //회원명 검색
        if (StringUtils.hasText(orderSearch.getMemberName())) {
            Join<Order, Member> m = o.join("member", JoinType.INNER); //회원과 조인
            Predicate name = cb.like(m.<String>get("name"), "%" + orderSearch.ge
            criteria.add(name);
        }

        cq.where(cb.and(criteria.toArray(new Predicate[criteria.size()])));
        TypedQuery<Order> query = em.createQuery(cq).setMaxResults(1000); //최대
        return query.getResultList();
    }
}
```

====== 주문 리파지토리 리펙토링 후 ======

```
package jpabook.jpashop.repository;

import jpabook.jpashop.domain.Order;
import jpabook.jpashop.repository.custom.CustomOrderRepository;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.JpaSpecificationExecutor;

public interface OrderRepository extends JpaRepository<Order, Long>, JpaSpecific
```

```
    }
```

주문 리파지토리에는 검색이라는 복잡한 로직이 있다. 스프링 데이터 JPA가 제공하는 명세(Specification) 기능을 사용해서 검색을 구현해보자. 명세 기능을 사용하기 위해 `JpaSpecificationExecutor`를 추가로 상속 받았다.

- 명세(Specification) 적용

명세로 검색하는 기능을 사용하려면 리파지토리에 `org.springframework.data.jpa.repository.JpaSpecificationExecutor`를 추가로 상속받아야 한다. 이제 명세를 작성하기 위한 클래스인 `OrderSpec`을 추가하자.

====== `OrderSpec` 추가 ======

```
package jpabook.jpashop.domain;

import org.springframework.data.jpa.domain.Specification;
import org.springframework.util.StringUtils;

import javax.persistence.criteria.*;

public class OrderSpec {

    public static Specification<Order> memberNameLike(final String memberName) {
        return new Specification<Order>() {
            public Predicate toPredicate(Root<Order> root, CriteriaQuery<?> quer

                if (StringUtils.isEmpty(memberName)) return null;

                Join<Order, Member> m = root.join("member", JoinType.INNER); //
                return builder.like(m.<String>get("name"), "%" + memberName + "%
            }
        };
    }

    public static Specification<Order> orderStatusEq(final OrderStatus orderStat
        return new Specification<Order>() {
            public Predicate toPredicate(Root<Order> root, CriteriaQuery<?> quer

                if (orderStatus == null) return null;

                return builder.equal(root.get("status"), orderStatus);
            }
        };
    }
```

```
}
```

다음으로 검색조건을 가지고 있는 `OrderSearch` 객체에 자신이 가진 검색조건으로 `Specification`을 생성하도록 코드를 추가하자.

====== 검색 객체가 Specification 생성하도록 추가 ======

```
package jpabook.jpashop.domain;

import org.springframework.data.jpa.domain.Specifications;

import static jpabook.jpashop.domain.OrderSpec.memberNameLike;
import static jpabook.jpashop.domain.OrderSpec.orderStatusEq;
import static org.springframework.data.jpa.domain.Specifications.where;

public class OrderSearch {

    private String memberName;      //회원명
    private OrderStatus orderStatus;//주문 상태

    //...Getter, Setter

    //추가
    public Specifications<Order> toSpecification() {
        return where(memberNameLike(memberName))
                .and(orderStatusEq(orderStatus));
    }

}
```

리파지토리의 검색 코드가 명세(`Specification`)를 파라미터로 넘기도록 변경하자.

====== `OrderService.findOrders` 리펙토링 전 ======

```
public List<Order> findOrders(OrderSearch orderSearch) {
    return orderRepository.findAll(orderSearch);
}
```

====== `OrderService.findOrders` 리펙토링 후 ======

```
public List<Order> findOrders(OrderSearch orderSearch) {
    return orderRepository.findAll(orderSearch.toSpecification());
```

```
}
```

기존 리파지토리 코드들을 스프링 데이터 JPA를 사용하도록 리펙토링 했다. 다음으로 스프링 데이터 JPA에 추가로 QueryDSL을 사용해보자.

스프링 데이터 JPA와 QueryDSL 통합하기

스프링 데이터 JPA는 2가지 방법으로 QueryDSL을 지원한다.

- `org.springframework.data.querydsl.QueryDslPredicateExecutor`
- `org.springframework.data.querydsl.QueryDslRepositorySupport`

- QueryDslPredicateExecutor 사용

첫 번째 방법은 리파지토리에서 `QueryDslPredicateExecutor` 를 상속 받으면 된다.

```
public interface ItemRepository extends JpaRepository<Item, Long>, QueryDslPredi
}
```

이제 상품 리파지토리에서 QueryDSL을 사용할 수 있다.

다음 예제는 QueryDSL이 생성한 쿼리 타입으로 장난감이라는 이름을 포함하고 있으면서 가격이 10000 ~ 20000원인 상품을 검색한다.

====== 사용 예제 ======

```
QItem item = QItem.item;
Iterable<Item> result = itemRepository.findAll(
        item.name.contains("장난감").and(item.price.between(10000, 20000))
);
```

`QueryDslPredicateExecutor` 인터페이스를 보면 QueryDSL을 검색조건으로 사용하면서 스프링 데이터 JPA가 제공하는 페이징과 정렬 기능도 함께 사용할 수 있다.

====== `QueryDslPredicateExecutor` 인터페이스 ======

```
public interface QueryDslPredicateExecutor<T> {
```

```
    T findOne(Predicate predicate);
    Iterable<T> findAll(Predicate predicate);
    Iterable<T> findAll(Predicate predicate, OrderSpecifier<?>... orders);
    Page<T> findAll(Predicate predicate, Pageable pageable);
    long count(Predicate predicate);
}
```

- QueryDslPredicateExecutor 한계

`QueryDslPredicateExecutor`는 스프링 데이터 JPA에서 편리하게 QueryDSL을 사용할 수 있지만 기능에 한계가 있다. 예를 들어 `join`, `fetch`를 사용할 수 없다. (JPQL에서 이야기하는 묵시적 조인은 가능하다.) 따라서 QueryDSL이 제공하는 다양한 기능을 사용하려면 `JPAQuery`를 직접 사용하거나 스프링 데이터 JPA가 제공하는 `QueryDslRepositorySupport`를 사용해야 한다.

- QueryDslRepositorySupport 사용

QueryDSL의 모든 기능을 사용하려면 `JPAQuery` 객체를 직접 생성해서 사용하면 된다. 이때 스프링 데이터 JPA가 제공하는 `QueryDslRepositorySupport`를 상속 받아 사용하면 조금 더 편리하게 QueryDSL을 사용할 수 있다.

사용 예제를 보자.

====== CustomOrderRepository 사용자 정의 리파지토리 ======

```
package jpabook.jpashop.repository.custom;

import jpabook.jpashop.domain.Order;
import jpabook.jpashop.domain.OrderSearch;
import java.util.List;

public interface CustomOrderRepository {

    public List<Order> search(OrderSearch orderSearch);

}
```

스프링 데이터 JPA가 제공하는 공통 인터페이스는 직접 구현할 수 없기 때문에 `CustomOrderRepository`라는 사용자 정의 리파지토리를 만들었다.

====== QueryDslRepositorySupport 사용 코드 ======

```
package jpabook.jpashop.repository.custom;

import com.mysema.query.jpa.JPQLQuery;
import jpabook.jpashop.domain.Order;
import jpabook.jpashop.domain.OrderSearch;
import jpabook.jpashop.domain.QMember;
import jpabook.jpashop.domain.QOrder;
import org.springframework.data.jpa.repository.support.QueryDslRepositorySupport
import org.springframework.util.StringUtils;
import java.util.List;

public class OrderRepositoryImpl extends QueryDslRepositorySupport implements Cu

    public OrderRepositoryImpl() {
        super(Order.class);
    }

    @Override
    public List<Order> search(OrderSearch orderSearch) {

        QOrder order = QOrder.order;
        QMember member = QMember.member;

        JPQLQuery query = from(order);

        if (StringUtils.hasText(orderSearch.getMemberName())) {
            query.leftJoin(order.member, member)
                    .where(member.name.contains(orderSearch.getMemberName()));
        }

        if (orderSearch.getOrderStatus() != null) {
            query.where(order.status.eq(orderSearch.getOrderStatus()));
        }

        return query.list(order);
    }
}
```

예제는 웹 애플리케이션 만들기에서 사용했던 주문 내역 검색 기능을 `QueryDslRepositorySupport`를 사용해서 QueryDSL로 구현한 예제다. 검색 조건에 따라 동적으로 쿼리를 생성한다. 참고로 생성자에서 `QueryDslRepositorySupport`에 엔티티 클래스 정보를 넘겨주어야 한다.

다음은 `QueryDslRepositorySupport`의 핵심 기능만 간추려보았다.

====== QueryDslRepositorySupport 코드 ======

```
package org.springframework.data.jpa.repository.support;

@Repository
public abstract class QueryDslRepositorySupport {

    //엔티티 매니저 반환
    protected EntityManager getEntityManager() {
        return entityManager;
    }

    //from 절 반환
    protected JPQLQuery from(EntityPath<?>... paths) {
        return querydsl.createQuery(paths);
    }

    //QueryDSL delete 절 반환
    protected DeleteClause<JPADeleteClause> delete(EntityPath<?> path) {
        return new JPADeleteClause(entityManager, path);
    }

    //QueryDSL update 절 반환
    protected UpdateClause<JPAUpdateClause> update(EntityPath<?> path) {
        return new JPAUpdateClause(entityManager, path);
    }

    //스프링 데이터 JPA가 제공하는 Querydsl을 편하게 사용하도록 돕는 헬퍼 객체 반환
    protected Querydsl getQuerydsl() {
        return this.querydsl;
    }
}
```

정리

지금까지 스프링 데이터 JPA 기능을 학습하고 앞서 만든 웹 애플리케이션에 적용해보았다. 적용한 예제를 보면 지루한 데이터 접근 계층의 코드가 상당히 많이 줄어든 것을 알 수 있다.

스프링 데이터 JPA는 지속해서 버전업하면서 다양한 기능들을 추가하고 있다. 나는 스프링 프레임워크와 JPA를 함께 사용한다면 스프링 데이터 JPA는 선택이 아닌 필수라 생각한다.

---

1. http://docs.spring.io/spring-data/jpa/docs/1.8.0.RELEASE/reference/html/jpa.repositories.html) ↩
2. http://ko.wikipedia.org/wiki/컴포지트_패턴 ↩