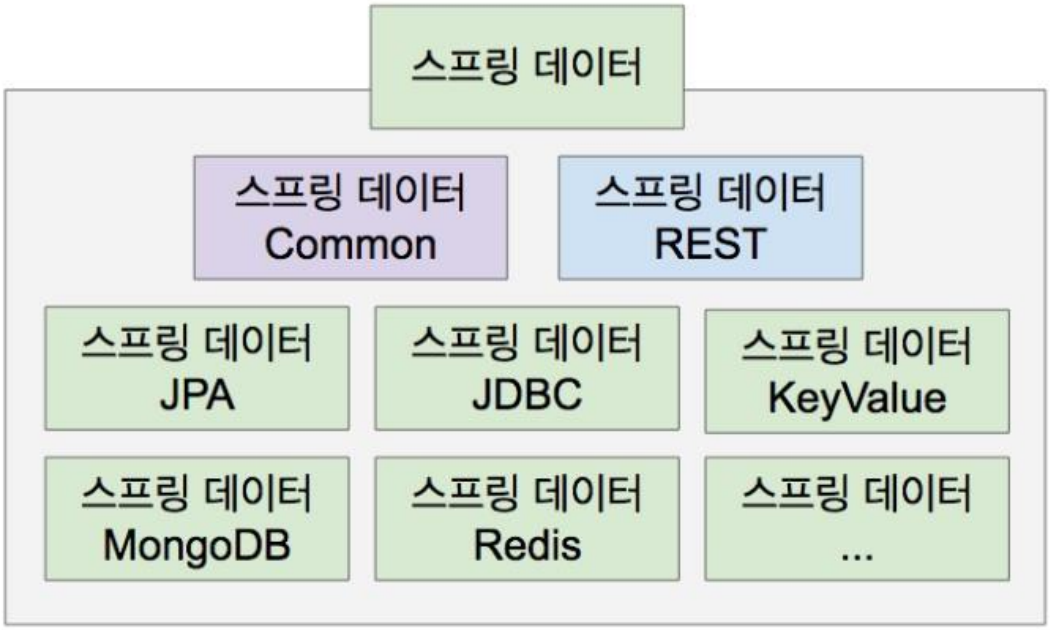
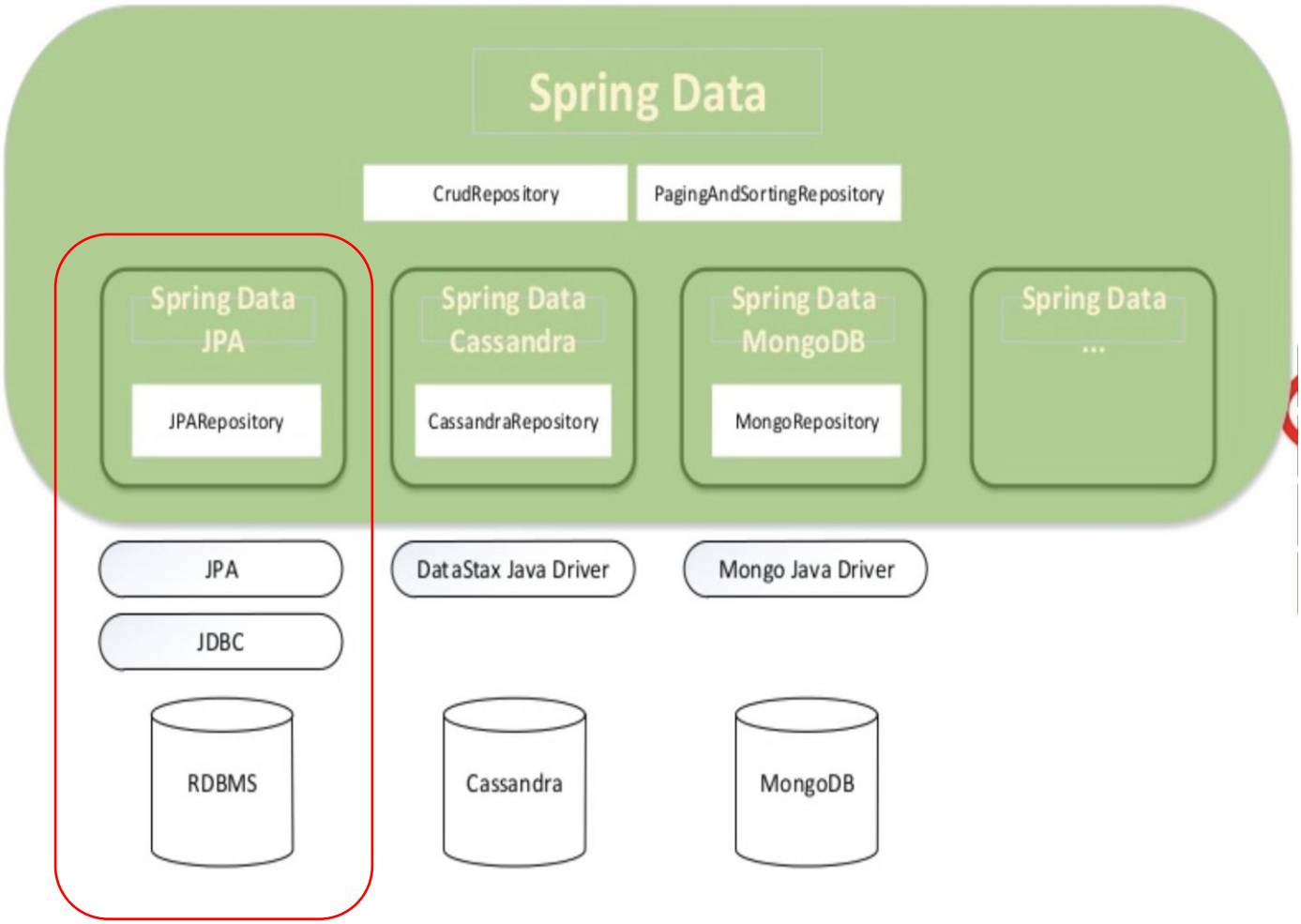
# Spring Data 와 Spring Data JPA



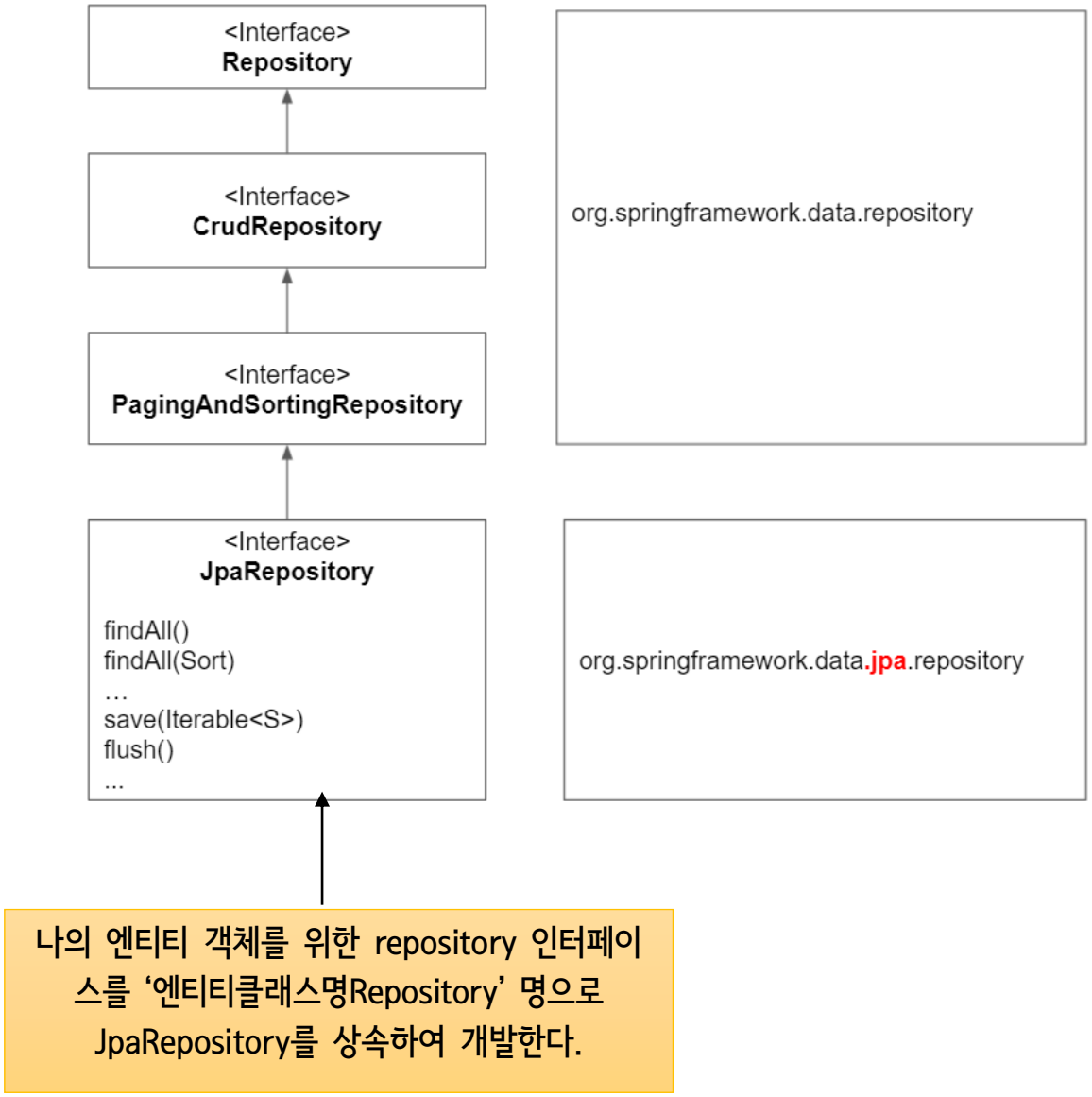



# Spring Data JPA

Spring Data JPA 는 Spring Framework에서 JPA를 편리하게 사용할 수 있도록 지원하는 프로젝트로서 CRUD 처리를 위한 공통 인터페이스를 제공한다.

Repository 개발 시 인터페이스만 작성해도 실행 시점에 Spring Data JPA가 구현 객체를 동적으로 생성해서 주입시키므로 데이터 접근 계층을 개발할 때 구현 클래스 없이 인터페이스만 작성해도 개발을 완료할 수 있도록 지원한다.

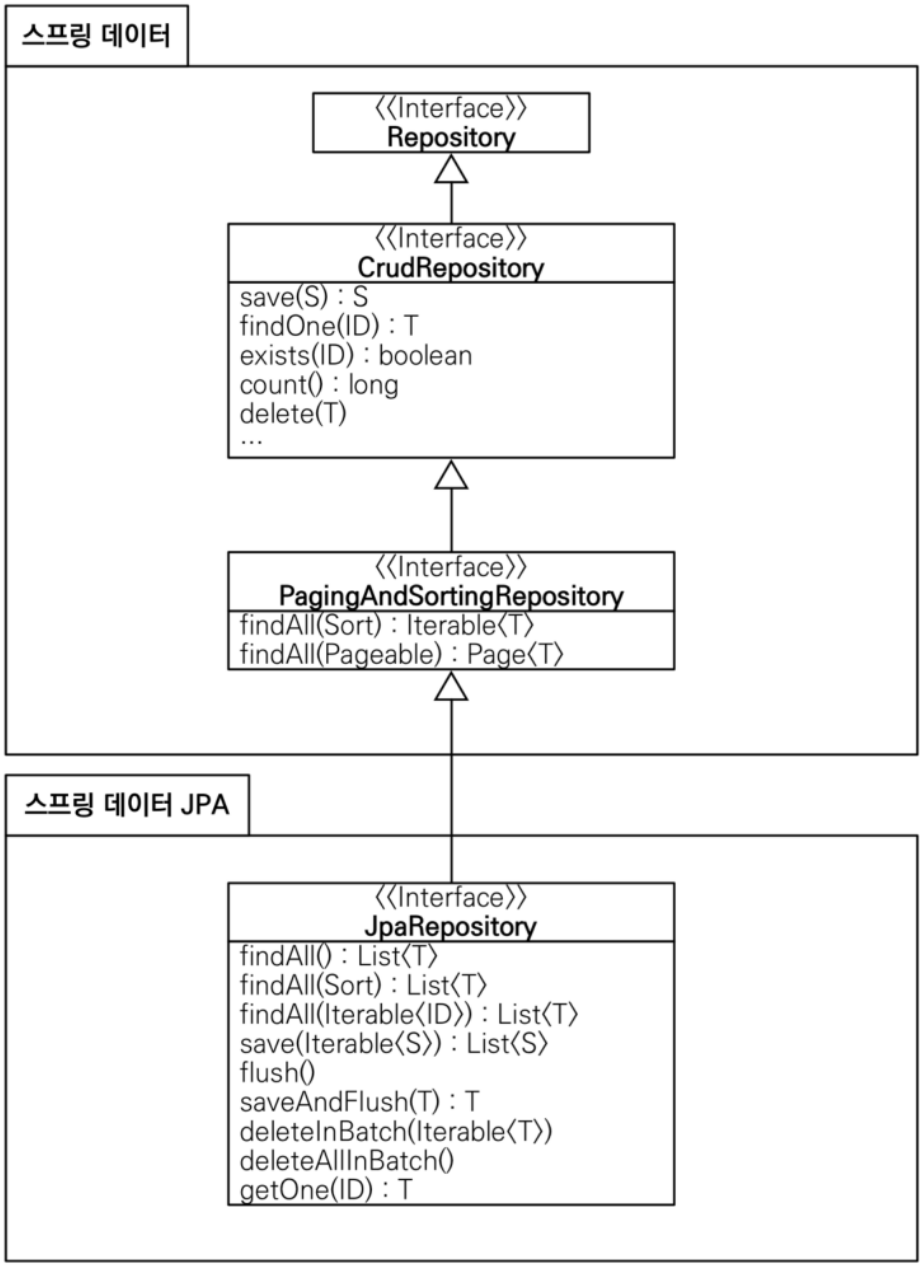
Spring Data JPA를 사용하기 위해 일반적으로 'JpaRepository<T, ID>' 인터페이스를 상속한 Repository 인터페이스를 정의한다. 단지 인터페이스를 상속했을 뿐인데, 기본적인 메서드를 이미 장착한 상태이고, 심지어 정의한 인터페이스를 구현할 필요도 없다. Spring이 알아서 해주게 된다.



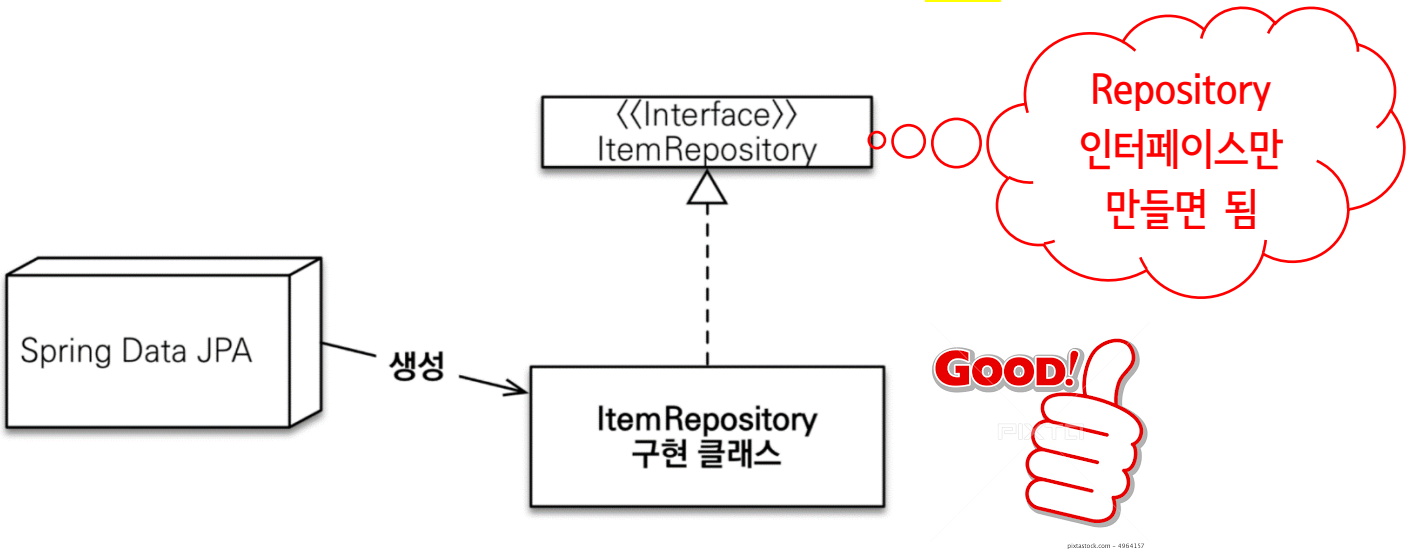
## JpaRepository 주요 메서드

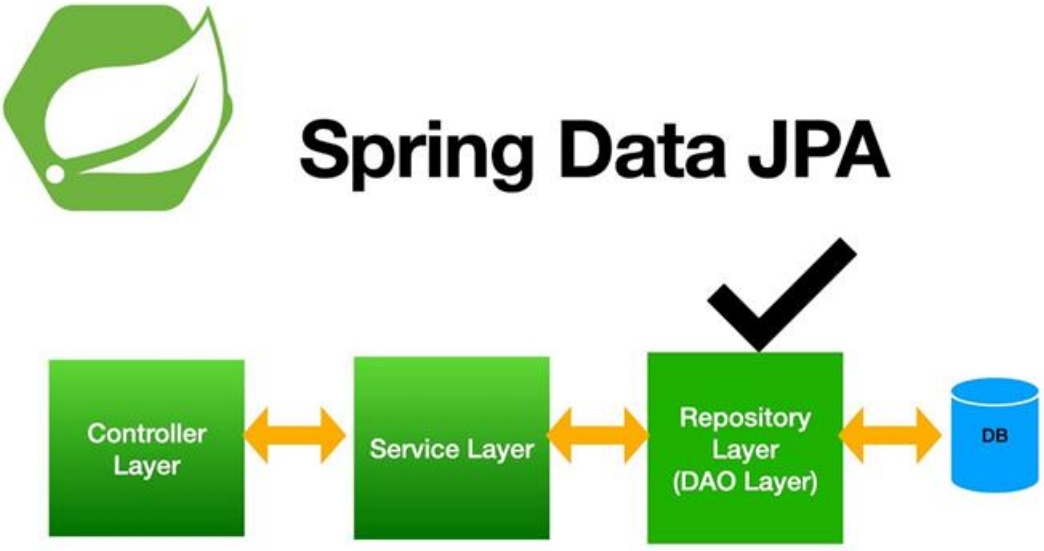
### 이 인터페이스에서 자주 쓰이는 메서드는 다음과 같다.

| 메서드 | 설명 |
| --- | --- |
| `<S extends T> S save(S)` | 새로운 엔티티는 저장하고, 이미 있는 엔티티는 병합한다. |
| `delete(T)` | 엔티티 하나를 삭제한다. |
| `Optional<T> findById(ID)` | ID로 엔티티 하나를 조회한다. |
| `List<T> findAll(...)` | 모든 엔티티를 조회한다.<br>정렬(Sort)이나 페이징(Pageable) 조건을 파라미터로 제공할 수 있다. |



```
public interface ItemRepository extends JpaRepository<T, ID> {
}
```




## 쿼리 메서드

위의 메서드들은 모든 엔티티에 대해 공통으로 쓰일 수 있는 메서드를 제공하지만, 사실 비즈니스 로직을 다루는 것은 그리 간단하지 않다. 조건을 지정하여 조회하거나, 제거하거나 저장할 수 있는 기능들을 커스터마이징 해야 한다.

Spring Data JPA는 Repository를 커스터마이징 하기위해 쿼리 메서드 기능을 제공하는데, 3가지 방법이 있다.

1) 메서드 이름으로 쿼리 생성 -> 간단한 쿼리 처리 시 좋음
2) @Query 안에 JPQL 정의 -> 복잡한 쿼리 처리 시 좋음
3) 메서드 이름으로 JPA NamedQuery 호출 (잘 안쓰임)

[ 메서드 이름으로 쿼리 생성 ]

| 쿼리 종류 | 이름 규칙 |
| --- | --- |
| 조회 | `find...By`, `read...By`, `query...By`, `get...By` |
| COUNT | `count...By` 반환타입 long |
| EXISTS | `exists...By` 반환타입 boolean |
| 삭제 | `delete...By`, `remove...By` |
| DISTINCT | `findDistinct`, `findMemberDistinctBy` |
| LIMIT | `findFirst3`, `findFirst`, `findTop`, `findTop3` |

- 수업시간에 다룰 쿼리 메서드의 예

```
public interface EmpRepository extends JpaRepository<Emp, Integer>{
}

public interface VisitorRepository extends JpaRepository<Visitor, Integer>{
	public List<Visitor> findByMemoContains(String keyword);
}

public interface MeetingRepository extends JpaRepository<Meeting, Integer>{
	public List<Meeting> findByTitleContains(String keyword);
	public List<Meeting> findByName(String name);
}

public interface ReplyRepository extends JpaRepository<Reply, Integer>{
	public List<Reply> findByRefid(Meeting vo);
}

public interface EmpRepository2 extends JpaRepository<Emp, Integer>{

	public List<Emp> findByEname(String name);
	public List<Emp> findByEnameIgnoreCase(String name);
	public List<Emp> findByJob(String job);
	public List<Emp> findByJobOrDeptno(String job, int dno);
	public List<Emp> findByJobAndDeptno(String job, int dno);
```

```
	public List<Emp> findDistinctByJob(String job);
	public List<Emp> findByDeptno(int dno);
	public List<Emp> findBySalGreaterThan(int inputsal);
	public List<Emp> findBySalGreaterThanEqual(int inputsal);
	public List<Emp> findBySalLessThan(int inputsal);
	public List<Emp> findBySalLessThanEqual(int inputsal);
	public List<Emp> findBySalBetween(int minsal, int maxsal);
	public List<Emp> findByCommNull();
	public List<Emp> findByCommNotNull();
	public List<Emp> findByHiredateAfter(java.sql.Date d);
	public List<Emp> findByHiredateBefore(java.sql.Date d);
	public List<Emp> findByEnameStartsWith(String partname);
	public List<Emp> findByEnameContains(String partname);
	public List<Emp> findByDeptnoOrderBySalDesc(int dno);
	public List<Emp> findTop3ByDeptnoOrderBySalDesc(int dno);
}

public interface MemberTeamRepository extends JpaRepository<Member, Integer>{

	@Query("select m from Member m join m.team t where t.teamname = :tn")
	public List<Member> aaa(@Param("tn") String tname);
	@Query("select t.teamname from Member m join m.team t where m.username = :un")
	public String bbb(@Param("un") String uname);

	public List<Member> findByUsername(String username);
	public Long countByUsername(String username);
	public Long countBy();
}
```

```java
@Getter
@Setter
@ToString
@Entity
public class Member {
  @Id
  @Column(name = "MEMBER_ID")
  @GeneratedValue(strategy = GenerationType.
IDENTITY)
  private int id;
  private String username;
  // 연관관계 매핑
  @ManyToOne
  @JoinColumn(name = "TEAM_ID")
  private Team team;

  public Member() {
  }
  public Member(String username,
                Team team) {
    super();
    this.username = username;
    this.team = team;
  }
}
```

```java
@Getter
@Setter
@ToString
@Entity
public class Team {
  @Id
  @Column(name = "TEAM_ID")
  private String id;
  private String teamname;

  public Team() {
  }
  public Team(String id, String teamname) {
    super();
    this.id = id;
    this.teamname = teamname;
  }
}
```

| Keyword | Sample | JPQL snippet |
| --- | --- | --- |
| `And` | `findByLastnameAndFirstname` | `… where x.lastname = ?1 and x.firstname = ?2` |
| `Or` | `findByLastnameOrFirstname` | `… where x.lastname = ?1 or x.firstname = ?2` |
| `Is,Equals` | `findByFirstname`, `findByFirstnameIs`, `findByFirstnameEquals` | `… where x.firstname = 1?` |
| `Between` | `findByStartDateBetween` | `… where x.startDate between 1? and ?2` |
| `LessThan` | `findByAgeLessThan` | `… where x.age < ?1` |
| `LessThanEqual` | `findByAgeLessThanEqual` | `… where x.age <= ?1` |
| `GreaterThan` | `findByAgeGreaterThan` | `… where x.age > ?1` |
| `GreaterThanEqual` | `findByAgeGreaterThanEqual` | `… where x.age >= ?1` |
| `After` | `findByStartDateAfter` | `… where x.startDate > ?1` |
| `Before` | `findByStartDateBefore` | `… where x.startDate < ?1` |
| `IsNull` | `findByAgeIsNull` | `… where x.age is null` |
| `IsNotNull,NotNull` | `findByAge(Is)NotNull` | `… where x.age not null` |
| `Like` | `findByFirstnameLike` | `… where x.firstname like ?1` |
| `NotLike` | `findByFirstnameNotLike` | `… where x.firstname not like ?1` |
| `StartingWith` | `findByFirstnameStartingWith` | `… where x.firstname like ?1` (parameter bound with appended `%`) |
| `EndingWith` | `findByFirstnameEndingWith` | `… where x.firstname like ?1` (parameter bound with prepended `%`) |
| `Containing` | `findByFirstnameContaining` | `… where x.firstname like ?1` (parameter bound wrapped in `%`) |
| `OrderBy` | `findByAgeOrderByLastnameDesc` | `… where x.age = ?1 order by x.lastname desc` |
| `Not` | `findByLastnameNot` | `… where x.lastname <> ?1` |
| `In` | `findByAgeIn(Collection<Age> ages)` | `… where x.age in ?1` |
| `NotIn` | `findByAgeNotIn(Collection<Age> age)` | `… where x.age not in ?1` |
| `True` | `findByActiveTrue()` | `… where x.active = true` |
| `False` | `findByActiveFalse()` | `… where x.active = false` |
| `IgnoreCase` | `findByFirstnameIgnoreCase` | `… where UPPER(x.firstame) = UPPER(?1)` |

[ @Query를 이용하여 JPQL을 직성해서 쿼리 수행 ]

```
public interface SpringDataJpaItemRepository extends JpaRepository<Item, Long> {
        List<Item> findByItemNameLike(String itemName);
        List<Item> findByPriceLessThanEqual(Integer price);
        //쿼리 메서드 (아래 메서드와 같은 기능 수행)
        List<Item> findByItemNameLikeAndPriceLessThanEqual(
                                        String itemName, Integer price);
        //쿼리 직접 실행
        @Query(
            "select i from Item i where i.itemName like :itemName and i.price <= :price")
        List<Item> findItems(@Param("itemName") String itemName,
                                            @Param("price") Integer price);
}
```

Spring Data JPA가 제공하는 JpaRepository 인터페이스를 상속받으면 상속하는 것만으로도 기본적인 CRUD 기능을 사용할 수 있다. 그런데 이름으로 검색하거나, 가격으로 검색하는 기능은 공통으로 제공할 수 있는 기능이 아니다. 따라서 이런 경우네는 쿼리 메서드 기능을 사용하거나 @Query 를 사용해서 직접 쿼리를 실행한다.

위의 코드 데이터를 조건에 따라 4가지로 분류해서 검색한다.

- 모든 데이터 조회
- 이름 조회
- 가격 조회
- 이름 + 가격 조회

- findAll()
  코드에는 보이지 않지만 JpaRepository 공통 인터페이스가 제공하는 기능이다.
  모든 Item 을 조회한다.
  실행되는 JPQL - select i from Item i
- findByItemNameLike()
  이름 조건만 검색했을 때 사용하는 쿼리 메서드이다.
  실행되는 JPQL - select i from Item i where i.name like ?
- findByPriceLessThanEqual()
  가격 조건만 검색했을 때 사용하는 쿼리 메서드이다.
  실행되는 JPQL - select i from Item i where i.price <= ?
- findByItemNameLikeAndPriceLessThanEqual()
  이름과 가격 조건을 검색했을 때 사용하는 쿼리 메서드이다.
  실행되는 JPQL - select i from Item i where i.itemName like ? and i.price <= ?

- findItems()

  메서드 이름으로 쿼리를 실행하는 기능은 다음과 같은 단점이 있다.

  1. 조건이 많으면 메서드 이름이 너무 길어진다.
  2. 조인 같은 복잡한 조건을 사용할 수 없다.

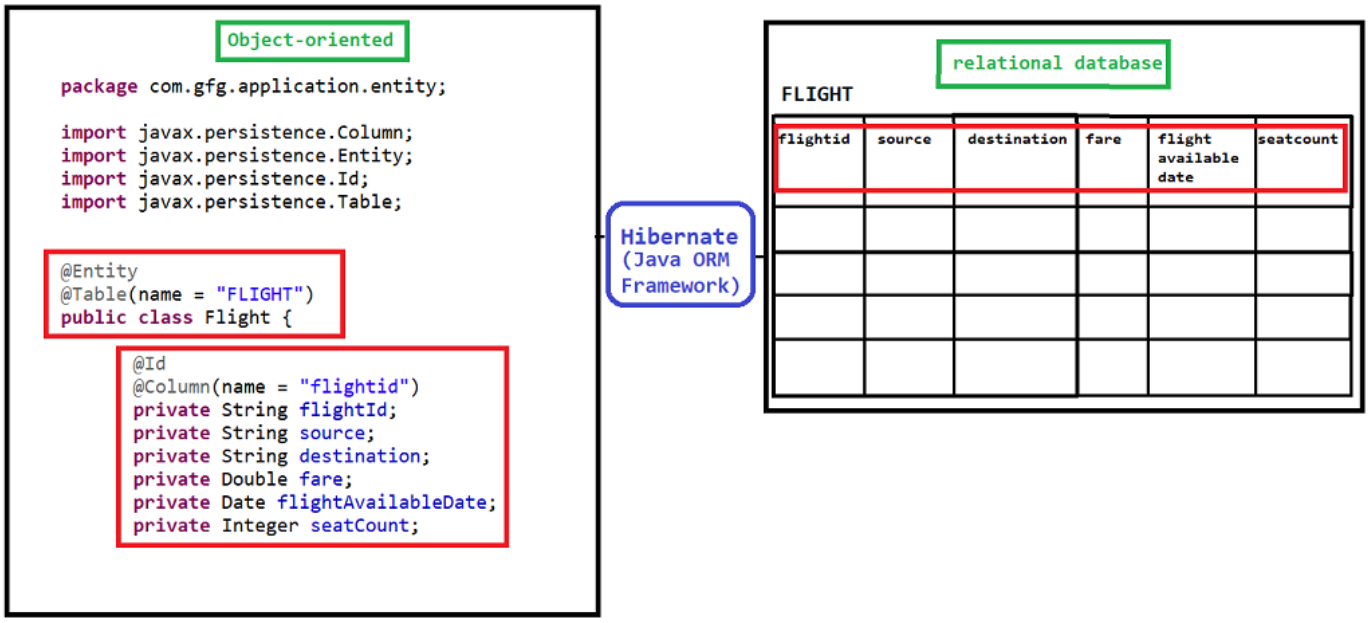
  메서드 이름으로 쿼리를 실행하는 기능은 간단한 경우에는 매우 유용하지만, 복잡해지면 직접 JPQL 쿼리를 작성하는 것이 좋다. 쿼리를 직접 실행하려면 @Query 어노테이션을 사용한다.

메서드 이름으로 쿼리를 실행할 때는 파라미터를 순서대로 입력하면 되지만, 쿼리를 직접 실행할 때는 파라미터를 명시적으로 바인딩 한다.

파라미터 바인딩은 @Param("itemName") 어노테이션을 사용하고, 어노테이션의 값에 파라미터 이름을 입력한다.

쿼리 메서드를 이용하는 방법과 @Query로 JPQL을 사용하는 방법 모두 복잡한 동적 쿼리에는 좀 부족한 편이다. 복잡한 동적 쿼리는 QueryDSL을 사용하는 것이 좋다.

## [ JPA 에서 가장 중용한 구현 - 엔티티 개발 ]




## [ 객체관의 관계 ]

PK FK

| HUMAN_ID | HUMAN_NAME | ANIMAL_ID |
|---|---|---|

```java
@Entity
@Table(name = "human")
public class Human {
    @Id
    @GeneratedValue
    @Column(name = "human_id")
    private long id;

    @Column(name = "human_name")
    private String name;

    @ManyToOne
    @JoinColumn(name = "animal_id")
    private Animal animal;
}
```

PK

| ANIMAL_ID | ANIMAL_NAME |
|---|---|

```java
@Entity
@Table(name = "animal")
public class Animal {
    @Id
    @GeneratedValue
    @Column(name = "animal_id")
    private long id;

    @Column(name = "animal_name")
    private String name;
}
```

[ Spring Boot Application 에 설정된 어노테이션 설명 ]

```
@SpringBootApplication
@ComponentScan(basePackages={"com.example.demo","thymeleaf.exam",
"springjpa.exam"})
@EnableJpaRepositories(basePackages = {"springjpa.exam.repository"})
@EntityScan(basePackages = {"springjpa.exam.entity"})
```

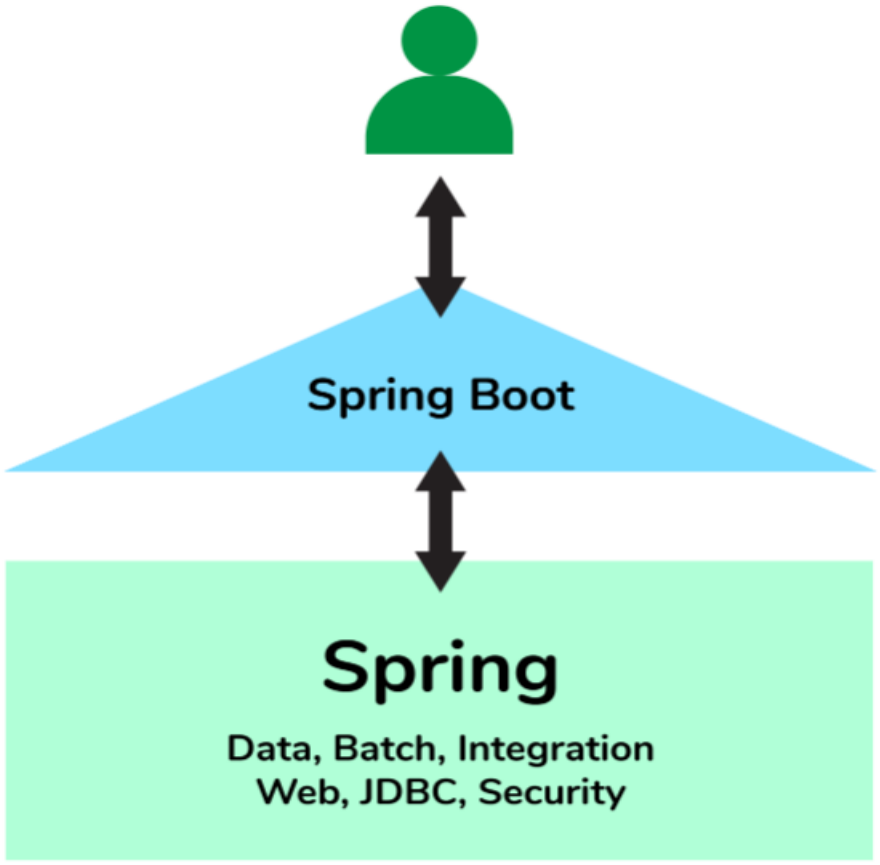
- @SpringBootApplication 어노테이션은 스프링 부트의 가장 기본적인 설정을 선언한다.
  내부적으로 @ComponentScan과 @EnableAutoConfiguration을 설정한다.

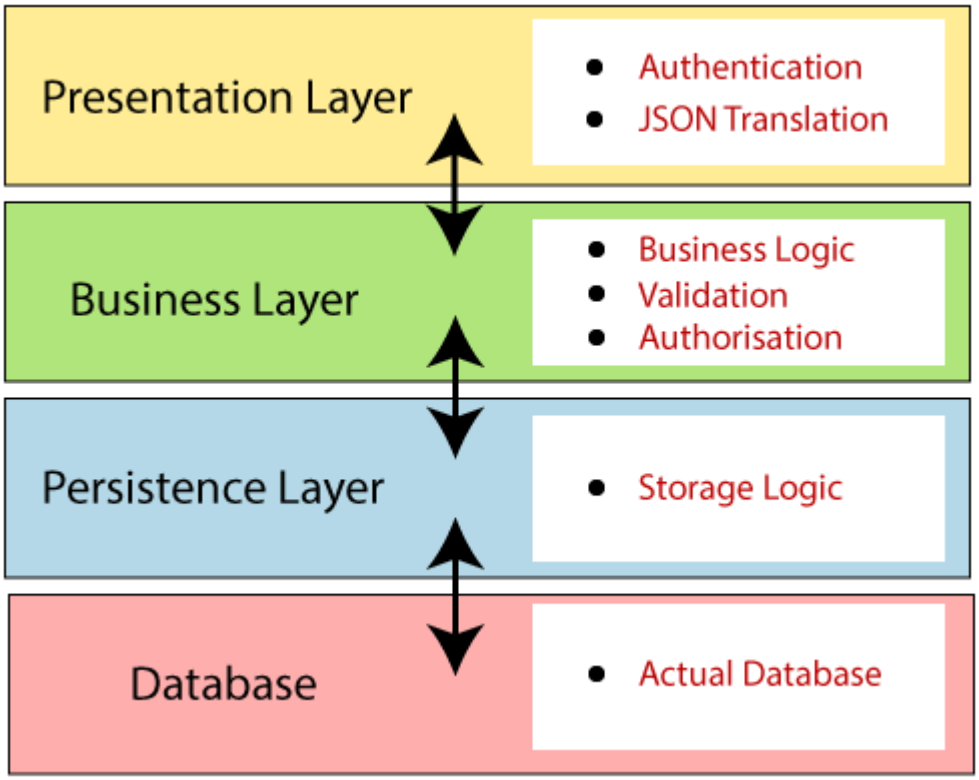
- @ComponentScan 어노테이션은 스프링 3.1부터 도입된 어노테이션이며 스캔 위치를 설정한다.

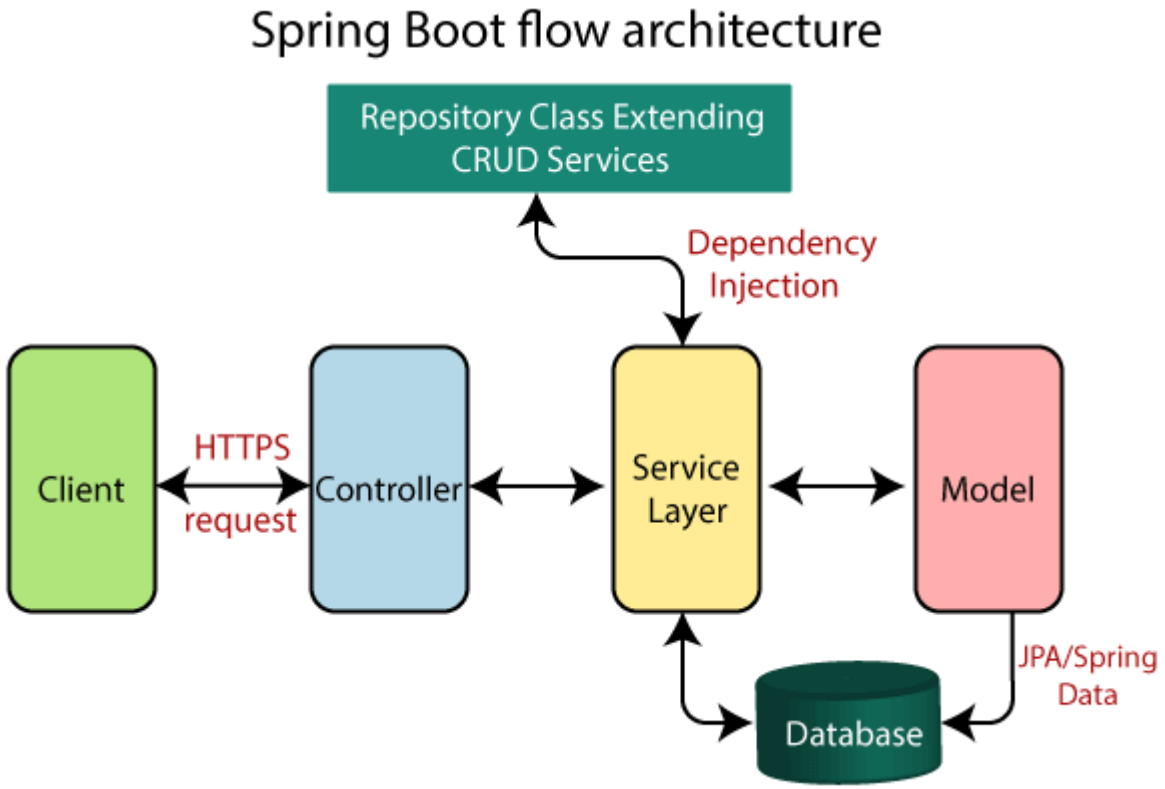
- @EnableJpaRepositories 어노테이션은 JPA Repository들을 활성화하기 위한 어노테이션이다.

- @EntityScan 어노테이션은 엔티티 클래스를 스캔할 곳을 지정하는데 사용한다.

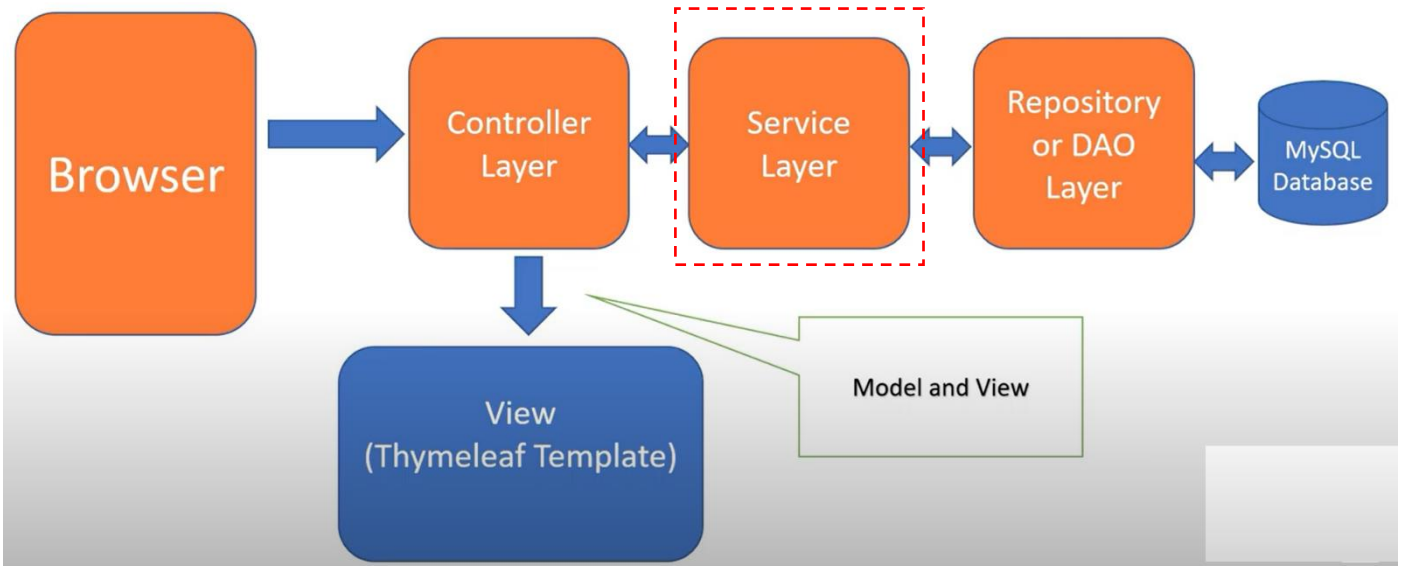
## Spring Boot ?




Presentation Layer
Authentication
JSON Translation
Business Layer
Business Logic
Validation
Authorisation
Persistence Layer
Storage Logic
Database
Actual Database


Spring Boot flow architecture
Repository Class Extending CRUD Services
Dependency Injection
Client
HTTPS request
Controller
Service Layer
Model
JPA/Spring Data
Database

[ 우리가 학습한 Spring Boot 구조 ]




## [ Spring Boot 테스트 ]

스프링 부트는 서블릿 기반의 웹 개발을 위한 spring-boot-starter-web, 유효성 검증을 위한 spring-boot-starter-validation 등 spring-boot-starter 의존성을 제공하고 있다. 테스트를 위한 spring-boot-starter-test 역시 존재하는데, 다음과 같은 라이브러리들이 포함된다.

JUnit 5: 자바 애플리케이션의 단위 테스트를 위한 사실상의 표준 테스트 도구
Spring Test & Spring Boot Test: 스프링 부트 애플리케이션에 대한 유틸리티 및 통합 테스트 지원
AssertJ: 유연한 검증 라이브러리
Hamcrest: 객체 Matcher를 위한 라이브러리
Mockito: 자바 모킹 프레임워크
JSONassert: JSON 검증을 위한 도구
JsonPath: JSON용 XPath

spring-boot-starter-test 의존성을 추가하면 스프링이 미리 만들어둔 테스트를 위한 다양한 어노테이션을 사용해 편리하게 테스트를 작성할 수 있다.

@SpringBootTest
@WebMvcTest
@DataJpaTest
@RestClientTest
@JsonTest
@JdbcTest
.....

[ @DataJpaTest 애노테이션 ]

**@DataJpaTest**

JPA 레포지토리 테스트를 위해서는 @DataJpaTest를 이용할 수 있다. @DataJpaTest는 기본적으로 @Entity가 있는 엔티티 클래스들을 스캔하며 테스트를 위한 TestEntityManager를 사용해 JPA 레포지토리들을 설정해준다.

@DataJpaTest의 기본적인 동작
앞서 설명하였듯 스프링은 테스트에 @Transactional이 있으면 테스트가 끝난 후 자동으로 트랜잭션을 롤백한다. @DataJpaTest에는 @Transactional 어노테이션이 들어있어서 기본적으로 모든 테스트가 롤백된다. 만약 롤백을 원하지 않는다면 @Rollback(false)를 추가하면 된다.

```
@DataJpaTest
@Rollback(false)
class MyRepositoryTests {

    // ...

}
```

또한 만약 H2와 같은 내장 데이터베이스가 클래스 패스에 존재한다면 내장 데이터베이스가 자동 구성된다. spring-boot-test 의존성에는 기본적으로 H2가 들어있으므로 별다른 설정을 주지 않는다면 H2로 설정된다. 내장 데이터베이스로 설정되기를 원하지 않는다면 다음과 같이 AutoConfigureTestDatabase의 replace 속성을 NONE으로 주면 된다.

```
@DataJpaTest
@AutoConfigureTestDatabase(replace = Replace.NONE)
class MyRepositoryTests {

    // ...

}
```

| 테스트 어노테이션 | @SpringBootTest | @WebMvcTest | @DataJpaTest |
|---|---|---|---|
| 테스트 종류 | 통합테스트 | 부분테스트<br>(슬라이스 테스트) | 부분테스트<br>(슬라이스 테스트) |
| 등록되는 빈 | 모든 빈 | Controller와 연관된 빈 | JPA Repository와 연관된 빈 |