Spring Data - Query Approaches

So far, you have learnt how to perform basic CRUD operations using Spring Data.

Consider the following entity class and the corresponding repository interface:

```
1.  @Entity
2.  public class Customer {
3.    @Id
4.    private Integer customerId;
5.    private String emailId;
6.    private String name;
7.    private LocalDate dateOfBirth;
8.
9.      //getters and setters
10.     // toString, hashCode and equals methods


11. }
```

```
1.  public interface CustomerRepository extends CrudRepository<Customer,
    Integer> {
2.
3.  }


4.
```

Now, if you need to fetch the details of customer based on emailId, how do you implement?

The methods provided by CrudRepository operate on primary key attribute whereas in this case, emailId is not the primary key.

So, there can be situations and requirements similar to the one discussed here where you will not have methods present in Spring Data repositories.

For implementing these type of requirements, Spring Data provides the following approaches:

- Query creation based on the method name
- Query creation using @Query annotation
- Query creation using @NamedQuery annotation

Query creation based on the method name

Query creation based on the method name

In this approach, we add methods in the interface which extends CrudRepository to implement custom requirements and Spring Data automatically generates the JPQL(Java Persistence Query Language) query based on the name of methods. These methods are called as **query methods** and are named according to some specific rules.

Some basic rules for naming these methods are as follows :

1. The method name should start with "**find...By**", "**get...By**", "**read...By**", "**count..By**" or "**query...By**" followed by search criteria. The search criteria is specified using attribute name of entity class and some specified keywords. For example, to search customer based on emailId, the following query method has to be added to repository interface :

```
1.  Customer findByEmailId(String emailId);
```

```
2.
```

When this method is called, it will be translated to the following JPQL query:

```
1. select c from Customer c where c.emailId = ?1
```

The number of parameters in query method must be equal to number of search conditions and they must be specified in the same order as mentioned in search conditions.

2. To fetch data based on more than one condition, you can concatenate entity attribute names using **And** and **Or** keywords to specify search criteria. For example, to search customers based on email address or name , the following method has to be added to repository interface :

```
1. List<Customer> findByEmailIdOrName(String emailId, String name);

2.
```

When the above method is called, it will be translated to following JPQL query:

```
1. select c from Customer c where c.emailId = ?1 or c.name=?2
```

3. Some other keywords that can be used inside query method names are Between, Is, Equals, Not, IsNot, IsNull, IsNotNull, LessThan, LessThanEqual, GreaterThan, GreaterThanEqual, After, Before, Like, etc.

The following table shows the use of above keywords :

Keyword	Method Name	Description
Between	findByDateofBirthBetween(LocalDate fromDate, LocalDate toDate)	Find customers whose date of birth is between specified dates
Equals	findByEmailIdEquals(String emailId)	Find customer with specified emailid
LessThan	findByDateOfBirthLessThan(LocalDate dateOfBirth)	Find customers born before specified dates
LessThanEqual	findByDateOfBirthLessThanEqual(LocalDate dateOfBirth)	Find customers born on or before specified date
GreaterThan	findByDateOfBirthGreaterThan(LocalDate dateOfBirth)	Find customers born after specified date
GreaterThanEqual	findByDateOfBirthGreaterThanEqual(LocalDate dateOfBirth)	Find customers born on or after specified date
After	findByDateOfBirthAfter(LocalDate dateOfBirth)	Find customers born after specified date
Before	findByDateOfBirthBefore(LocalDate dateOfBirth)	Find customers born before specified date
Like	findByNameLike(String pattern)	Find customers having with specified pattern
Null	findByEmailIdNull()	Find customers who have emailId
NotNull	findByEmailIdNotNull()	Find customers who don't have any emailId

4. To sort the results by a specified column, you can use **OrderBy** keyword. By default, results are arranged in ascending order. For example, the following method searches customers by name and also orders the result in ascending order of date of birth:

```
1. List<Customer> findByNameOrderByDateOfBirth(String name);
```

For sorting the results in descending order you can use Desc keyword as follows:

```
1. List<Customer> findByNameOrderByDateOfBirthDesc(String name);
```

5. To limit the number of results returned by a method, add **First** or **Top** keyword before the first 'By' keyword. To fetch more than one result, add the numeric value to the First and the Top keywords. For example, the following methods returns the first 5 customers whose emailId is given as method parameter:

```
1.  List<Customer> findFirst5ByEmailId(String emailId);
2.
3.  List<Customer> findTop5ByByEmailId(String emailId);
```

Query creation using @Query annotation

Query creation using @Query annotation

You have learnt how to create JPQL queries using name of query method but for complex requirements, query method names become too long which make them difficult to read and maintain. So, Spring Data provides an option to write custom JPQL queries on repository methods using **@Query** annotation.

For example, suppose you want to fetch customer name based on emailId, then this requirement can be implemented using @Query annotation as follows:

```
1.  public interface CustomerRepository extends CrudRepository<Customer,
    Integer> {
2.    //JPQL query
3.    @Query("SELECT c.name FROM Customer c WHERE c.emailId = :emailId")
4.    String findNameByEmailId(@Param("emailId") String emailId);
5.  }


6.
```

In above code, "SELECT c.name FROM Customer c WHERE c.emailId = :emailId" is JPQL query with named parameter "emailId". The @Param("emailId") parameter defines the named parameter in the argument list. You can also use positional parameter instead of named parameter as follows:

```
1.  public interface CustomerRepository extends CrudRepository<Customer,
    Integer> {
2.    //JPQL query
3.    @Query("SELECT c.name FROM Customer c WHERE c.emailId = ?1")
4.    String findNameByEmailId(String emailId);
5.  }


6.
```

Executing update and delete queries using @Modifying annotation

You can also execute update, delete or insert operations using @Query annotation. For this, query method has to be annotated with @Transactional and @Modifying annotation along with @Query annotation. For example, the following method updates the emailId of a customer based on customer's customerId.

```
1.  public interface CustomerRespository extends CrudRepository<Customer,
    Integer> {
2.    //JPQL query
3.      @Query("UPDATE Customer c SET c.emailId = :emailId WHERE c.customerId =
    :customerId")
4.    @Modifying
5.    @Transactional
6.    Integer updateCustomerEmailId(@Param("emailId") String
    updateCustomerByEmailId, @Param("customerId") Integer customerId);


7.  }
```

The @Modifying annotation triggers @Query annotation to be used for update or delete or insert operation instead of a select operation.

Query creation using @NamedQuery annotation

Query creation using @NamedQuery annotation

You have learnt how to generate query using query method name and how to define JPQL query using @Query annotation. In these approaches, the queries are scattered across different classes and the same queries may be written in different classes again and again which makes queries difficult to manage.

So, you can use named queries also with Spring Data. The named queries are queries with a name and are defined in entity classes using **@NamedQuery** annotation. Using Spring Data, default naming strategy of named query is to start with entity class name followed by dot (.) operator and then the name of the invoked repository method. For example, if CustomerEntity is entity class and findNameByEmailId is the name of repository method then query name will be as follows:

```
1. Customer.findNameByEmailId
```

Now, to associate JPQL query "SELECT c.name FROM CustomerEntity c WHERE c.emailId = :emailId" with the query name given above, the @NamedQuery annotation is used as follows:

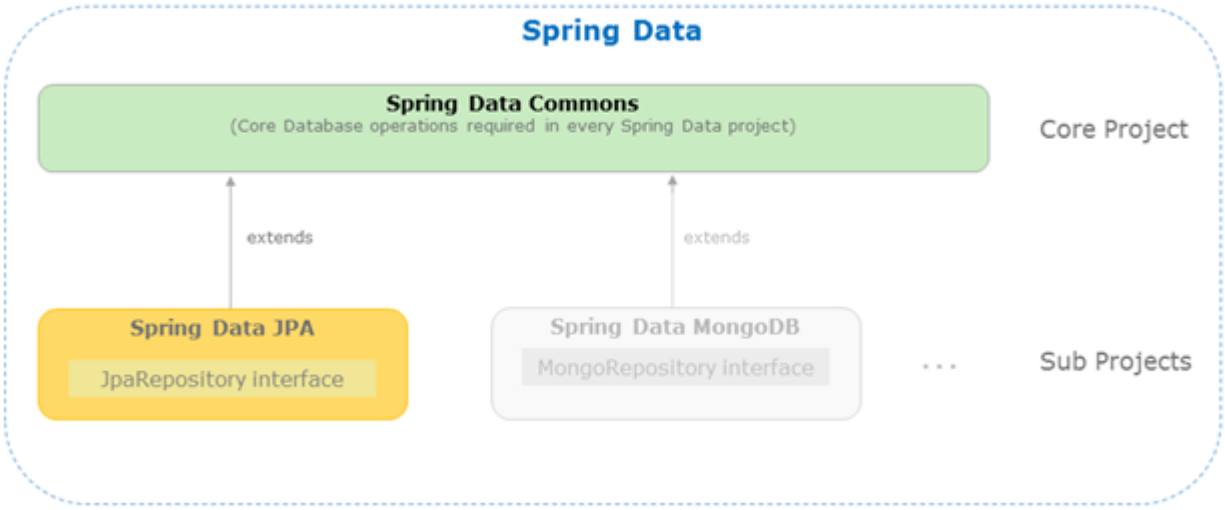
```
1. @Entity
2. @Table(name="customer")
3. @NamedQuery(name="Customer.findNameByEmailId", query="SELECT c.name FROM
   Customer c WHERE c.emailId = :emailId")
4. public class Customer {
5.   @Id
6.   private Integer customerId;
7.   private String emailId;
8.   private String name;
9.   private LocalDate dateOfBirth;
10.  @Enumerated(value=EnumType.STRING)
11.  private CustomerType customerType;
12.    // getter and setter methods

13. }
```

To use this named query in a repository, you need to add a query method in repository interface with the same name as defined in named query and the return type of method should be specified according to named query. For example, the following method has to be added to repository to invoke named query CustomerEntity.findNameByEmailId.

```
1. public interface CustomerRespository extends CrudRepository<Customer,
   Integer>{
2.   String findNameByEmailId(@Param("emailId") String emailId);
```

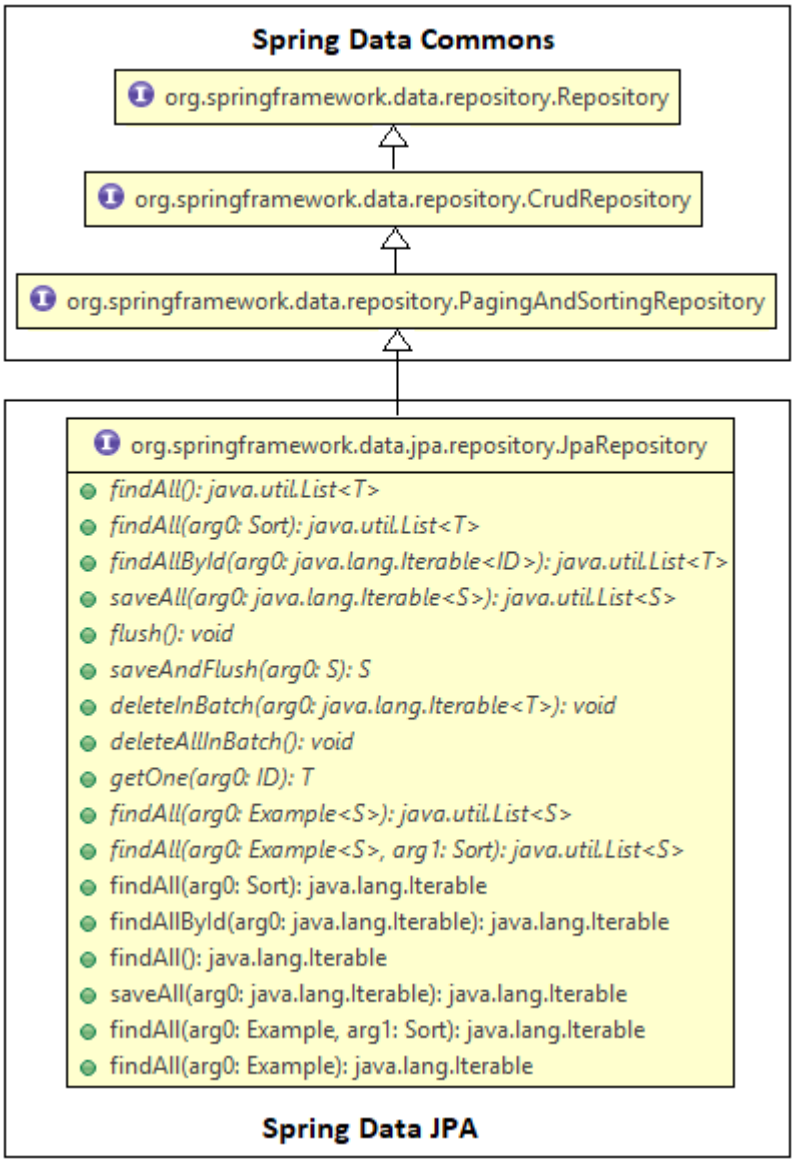
Spring Data JPA

Spring Data JPA is one of the sub project of the Spring Data which makes it easy to connect with relational databases using JPA based repositories by extending Spring Data repositories.



It provides the following interfaces:

- **JpaRepository<T, ID> interface**
  - It represents JPA specific repository.
  - It inherits methods of CrudRepository and PaginationAndSortingRepository. It also provides some additional methods for batch deletion of records and flushing the changes instantly to database.
  - JpaRepository is tightly coupled with JPA persistence technology. So use of this interface as base interface is not recommended. This is generally used used if JPA specific functionalities such as batch deletion and instant flushing of changes to database is required.



- **JpaSpecificationExecutor interface**
  - It is not a repository interface.
  - It provides methods that are used to fetch entities from database using JPA Criteria API.