JPA Query Methods

JPA Query Methods

- Query Lookup Strategies
  - Declared Queries
- Query Creation
  - Annotation-based Configuration
- Using JPA Named Queries
  - XML Named Query Definition
  - Declaring Interfaces
- Using @Query
  - Applying a QueryRewriter
  - Using Advanced LIKE Expressions
  - Native Queries
- Using Sort
- Scrolling Large Query Results
- Using Named Parameters
- Using Expressions
- Other Methods
- Modifying Queries
  - Derived Delete Queries
- Applying Query Hints
  - Adding Comments to Queries
- Configuring Fetch- and LoadGraphs
- Scrolling
  - Scrolling using Offset
  - Scrolling using Keyset-Filtering

This section describes the various ways to create a query with Spring Data JPA.

Query Lookup Strategies

The JPA module supports defining a query manually as a String or having it being derived from the method name.

Derived queries with the predicates `IsStartingWith`, `StartingWith`, `StartsWith`, `IsEndingWith`, `EndingWith`, `EndsWith`, `IsNotContaining`, `NotContaining`, `NotContains`, `IsContaining`, `Containing`, `Contains` the respective arguments for these queries will get sanitized. This means if

the arguments actually contain characters recognized by `LIKE` as wildcards these will get escaped so they match only as literals. The escape character used can be configured by setting the `escapeCharacter` of the `@EnableJpaRepositories` annotation. Compare with Using Value Expressions.

Declared Queries

Although getting a query derived from the method name is quite convenient, one might face the situation in which either the method name parser does not support the keyword one wants to use or the method name would get unnecessarily ugly. So you can either use JPA named queries through a naming convention (see Using JPA Named Queries for more information) or rather annotate your query method with `@Query` (see Using `@Query` for details).

Query Creation

Generally, the query creation mechanism for JPA works as described in Query Methods. The following example shows what a JPA query method translates into:

Example 1. Query creation from method names

```
JAVA
public interface UserRepository extends Repository<User, Long> {

  List<User> findByEmailAddressAndLastname(String emailAddress, String lastname);
}
```

We create a query using the JPA criteria API from this, but, essentially, this translates into the following query: `select u from User u where u.emailAddress = ?1 and u.lastname = ?2`. Spring Data JPA does a property check and traverses nested properties, as described in Property Expressions.

The following table describes the keywords supported for JPA and what a method containing that keyword translates to:

Table 1. Supported keywords inside method names

Keyword	Sample	JPQL snippet
`Distinct`	`findDistinctByLastnameAndFirstname`	`select distinct … where x.lastname = ?1 and x.firstname = ?2`
`And`	`findByLastnameAndFirstname`	`… where x.lastname ?1 and x.firstname = ?2`

Keyword	Sample	JPQL snippet
`Or`	`findByLastnameOrFirstname`	`… where x.lastname ?1 or x.firstname = ?2`
`Is, Equals`	`findByFirstname, findByFirstnameIs, findByFirstnameEquals`	`… where x.firstname = ?1` (or `… where x.firstname IS NULL` if the argument is `null`)
`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, Null`	`findByAge(Is)Null`	`… where x.age is null`
`IsNotNull, NotNull`	`findByAge(Is)NotNull`	`… where x.age is no 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 `%`)

Keyword	Sample	JPQL snippet
`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` (or `… where x.lastname IS NOT NULL` if the argument is `null`)
`In`	`findByAgeIn(Collection<Age> ages)`	`… where x.age in ?1`
`NotIn`	`findByAgeNotIn(Collection<Age> ages)`	`… where x.age not i ?1`
`True`	`findByActiveTrue()`	`… where x.active = true`
`False`	`findByActiveFalse()`	`… where x.active = false`
`IgnoreCase`	`findByFirstnameIgnoreCase`	`… where UPPER(x.firstname) = UPPER(?1)`

NOTE

`In` and `NotIn` also take any subclass of `Collection` as a parameter as well as arrays or varargs. For other syntactical versions of the same logical operator, check Repository query keywords.

WARNING

WARNING

`DISTINCT` can be tricky and not always producing the results you expect. For example, `select distinct u from User u` will produce a complete different result than `select distinct u.lastname from User u`. In the first case, since you are including `User.id`, nothing will be duplicated, hence you'll get the whole table, and it would be of `User` objects.

However, that latter query would narrow the focus to just `User.lastname` and find all unique last names for that table. This would also yield a `List<String>` result set instead of a `List<User>` result set.

`countDistinctByLastname(String lastname)` can also produce unexpected results. Spring Data JPA will derive `select count(distinct u.id) from User u where u.lastname = ?1`. Again, since `u.id` won't hit any duplicates, this query will count up all the users that had the binding last name. Which would be the same as `countByLastname(String lastname)`!

What is the point of this query anyway? To find the number of people with a given last name? To find the number of *distinct* people with that binding last name? To find the number of *distinct last names*? (That last one is an entirely different query!) Using `distinct` sometimes requires writing the query by hand and using `@Query` to best capture the information you seek, since you also may be needing a projection to capture the result set.

Annotation-based Configuration

Annotation-based configuration has the advantage of not needing another configuration file to be edited, lowering maintenance effort. You pay for that benefit by the need to recompile your domain class for every new query declaration.

Example 2. Annotation-based named query configuration

JAVA

```
@Entity
@NamedQuery(name = "User.findByEmailAddress",
  query = "select u from User u where u.emailAddress = ?1")
public class User {

}
```

Using JPA Named Queries

NOTE

The examples use the `<named-query />` element and `@NamedQuery` annotation. The queries for these configuration elements have to be defined in the JPA query language. Of course, you can use `<named-native-query />` or `@NamedNativeQuery` too. These elements let you define the query in native SQL by losing the database platform independence.

XML Named Query Definition

To use XML configuration, add the necessary `<named-query />` element to the `orm.xml` JPA configuration file located in the `META-INF` folder of your classpath. Automatic invocation of named queries is enabled by using some defined naming convention. For more details, see below.

Example 3. XML named query configuration

```xml
<named-query name="User.findByLastname">
  <query>select u from User u where u.lastname = ?1</query>
</named-query>
```

The query has a special name that is used to resolve it at runtime.

Declaring Interfaces

To allow these named queries, specify the `UserRepository` as follows:

Example 4. Query method declaration in UserRepository

```java
public interface UserRepository extends JpaRepository<User, Long> {

  List<User> findByLastname(String lastname);

  User findByEmailAddress(String emailAddress);
}
```

Spring Data tries to resolve a call to these methods to a named query, starting with the simple name of the configured domain class, followed by the method name separated by a dot. So the preceding example would use the named queries defined earlier instead of trying to create a query from the method name.

Using @Query

Using named queries to declare queries for entities is a valid approach and works fine for a small number of queries. As the queries themselves are tied to the Java method that runs them, you can actually bind them directly by using the Spring Data JPA `@Query` annotation rather than annotating them to the domain class. This frees the domain class from persistence specific information and co-locates the query to the repository interface.

Queries annotated to the query method take precedence over queries defined using `@NamedQuery` or named queries declared in `orm.xml`.

The following example shows a query created with the `@Query` annotation:

Example 5. Declare query at the query method using `@Query`

```java
public interface UserRepository extends JpaRepository<User, Long> {

  @Query("select u from User u where u.emailAddress = ?1")
  User findByEmailAddress(String emailAddress);
}
```

Applying a QueryRewriter

Sometimes, no matter how many features you try to apply, it seems impossible to get Spring Data JPA to apply every thing you'd like to a query before it is sent to the `EntityManager`.

You have the ability to get your hands on the query, right before it's sent to the `EntityManager` and "rewrite" it. That is, you can make any alterations at the last moment. Query rewriting applies to the actual query and, when applicable, to count queries. Count queries are optimized and therefore, either not necessary or a count is obtained through other means, such as derived from a Hibernate `SelectionQuery`.

Example 6. Declare a QueryRewriter using @Query

```java
public interface MyRepository extends JpaRepository<User, Long> {

        @NativeQuery(value = "select original_user_alias.* from SD_USER
original_user_alias",
                queryRewriter = MyQueryRewriter.class)
        List<User> findByNativeQuery(String param);

        @Query(value = "select original_user_alias from User original_user_alias",
                queryRewriter = MyQueryRewriter.class)
        List<User> findByNonNativeQuery(String param);
}
```

This example shows both a native (pure SQL) rewriter as well as a JPQL query, both leveraging the same `QueryRewriter`. In this scenario, Spring Data JPA will look for a bean registered in the application context of the corresponding type.

You can write a query rewriter like this:

Example 7. Example `QueryRewriter`

```java
public class MyQueryRewriter implements QueryRewriter {

     @Override
     public String rewrite(String query, Sort sort) {
         return query.replaceAll("original_user_alias", "rewritten_user_alias");
```

```
    }
}
```

You have to ensure your `QueryRewriter` is registered in the application context, whether it's by applying one of Spring Framework's `@Component` -based annotations, or having it as part of a `@Bean` method inside an `@Configuration` class.

Another option is to have the repository itself implement the interface.

Example 8. Repository that provides the `QueryRewriter`

```java
public interface MyRepository extends JpaRepository<User, Long>, QueryRewriter {

        @Query(value = "select original_user_alias.* from SD_USER
original_user_alias",
                nativeQuery = true,
                queryRewriter = MyRepository.class)
        List<User> findByNativeQuery(String param);

        @Query(value = "select original_user_alias from User original_user_alias",
                queryRewriter = MyRepository.class)
        List<User> findByNonNativeQuery(String param);

        @Override
        default String rewrite(String query, Sort sort) {
            return query.replaceAll("original_user_alias", "rewritten_user_alias");
        }
}
```

Depending on what you're doing with your `QueryRewriter` , it may be advisable to have more than one, each registered with the application context.

NOTE

In a CDI-based environment, Spring Data JPA will search the `BeanManager` for instances of your implementation of `QueryRewriter` .

Using Advanced `LIKE` Expressions

The query running mechanism for manually defined queries created with `@Query` allows the definition of advanced `LIKE` expressions inside the query definition, as shown in the following example:

Example 9. Advanced `Like` expressions in @Query

```java
public interface UserRepository extends JpaRepository<User, Long> {

  @Query("select u from User u where u.firstname like %?1")
```

```
  List<User> findByFirstnameEndsWith(String firstname);
}
```

In the preceding example, the `LIKE` delimiter character (`%`) is recognized, and the query is transformed into a valid JPQL query (removing the `%`). Upon running the query, the parameter passed to the method call gets augmented with the previously recognized `LIKE` pattern.

Native Queries

Using the `@NativeQuery` annotation allows running native queries, as shown in the following example:

Example 10. Declare a native query at the query method using @Query

JAVA

```
public interface UserRepository extends JpaRepository<User, Long> {

  @NativeQuery(value = "SELECT * FROM USERS WHERE EMAIL_ADDRESS = ?1")
  User findByEmailAddress(String emailAddress);
}
```

NOTE

The `@NativeQuery` annotation is mostly a composed annotation for `@Query(nativeQuery=true)` but it also provides additional attributes such as `sqlResultSetMapping` to leverage JPA's `@SqlResultSetMapping(…)`.

NOTE

Spring Data can rewrite simple queries for pagination and sorting. More complex queries require either JSqlParser to be on the class path or a `countQuery` declared in your code. See the example below for more details.

Example 11. Declare native count queries for pagination at the query method by using @NativeQuery

JAVA

```
public interface UserRepository extends JpaRepository<User, Long> {

  @NativeQuery(value = "SELECT * FROM USERS WHERE LASTNAME = ?1",
    countQuery = "SELECT count(*) FROM USERS WHERE LASTNAME = ?1")
  Page<User> findByLastname(String lastname, Pageable pageable);
}
```

TIP

TIP

It is possible to disable usage of `JSqlParser` for parsing native queries although it is available on the class-path by setting `spring.data.jpa.query.native.parser=regex` via the `spring.properties` file or a system property.

Valid values are (case-insensitive):

- `auto` (default, automatic selection)
- `regex` (Use the builtin regex-based Query Enhancer)
- `jsqlparser` (Use JSqlParser)

A similar approach also works with named native queries, by adding the `.count` suffix to a copy of your query. You probably need to register a result set mapping for your count query, though.

Next to obtaining mapped results, native queries allow you to read the raw `Tuple` from the database by choosing a `Map` container as the method's return type. The resulting map contains key/value pairs representing the actual database column name and the value.

Example 12. Native query returning raw column name/value pairs

JAVA

```
interface UserRepository extends JpaRepository<User, Long> {

  @NativeQuery("SELECT * FROM USERS WHERE EMAIL_ADDRESS = ?1")
  Map<String, Object> findRawMapByEmail(String emailAddress);       1

  @NativeQuery("SELECT * FROM USERS WHERE LASTNAME = ?1")
  List<Map<String, Object>> findRawMapByLastname(String lastname);  2
}
```

1 Single `Map` result backed by a `Tuple`.

2 Multiple `Map` results backed by `Tuple`s.

NOTE

String-based Tuple Queries are only supported by Hibernate. Eclipselink supports only Criteria-based Tuple Queries.

Using Sort

Sorting can be done by either providing a `PageRequest` or by using `Sort` directly. The properties actually used within the `Order` instances of `Sort` need to match your domain model, which means they need to resolve to either a property or an alias used within the query. The JPQL defines this as a state field path expression.

NOTE

NOTE

Using any non-referenceable path expression leads to an `Exception`.

However, using `Sort` together with `@Query` lets you sneak in non-path-checked `Order` instances containing functions within the `ORDER BY` clause. This is possible because the `Order` is appended to the given query string. By default, Spring Data JPA rejects any `Order` instance containing function calls, but you can use `JpaSort.unsafe` to add potentially unsafe ordering.

The following example uses `Sort` and `JpaSort`, including an unsafe option on `JpaSort`:

Example 13. Using `Sort` and `JpaSort`

JAVA

```
public interface UserRepository extends JpaRepository<User, Long> {

  @Query("select u from User u where u.lastname like ?1%")
  List<User> findByAndSort(String lastname, Sort sort);

  @Query("select u.id, LENGTH(u.firstname) as fn_len from User u where u.lastname
like ?1%")
  List<Object[]> findByAsArrayAndSort(String lastname, Sort sort);
}

repo.findByAndSort("lannister", Sort.by("firstname"));                1
repo.findByAndSort("stark", Sort.by("LENGTH(firstname)"));            2
repo.findByAndSort("targaryen", JpaSort.unsafe("LENGTH(firstname)")); 3
repo.findByAsArrayAndSort("bolton", Sort.by("fn_len"));               4
```

1 Valid `Sort` expression pointing to property in domain model.

2 Invalid `Sort` containing function call. Throws Exception.

3 Valid `Sort` containing explicitly *unsafe* `Order`.

4 Valid `Sort` expression pointing to aliased function.

Scrolling Large Query Results

When working with large data sets, scrolling can help to process those results efficiently without loading all results into memory.

You have multiple options to consume large query results:

1. Paging. You have learned in the previous chapter about `Pageable` and `PageRequest`.
2. Offset-based scrolling. This is a lighter variant than paging because it does not require the total result count.
3. Keyset-based scrolling. This method avoids the shortcomings of offset-based result retrieval by leveraging database indexes.

Read more on which method to use best for your particular arrangement.

You can use the Scroll API with query methods, Query-by-Example, and Querydsl.

NOTE

Scrolling with String-based query methods is not yet supported. Scrolling is also not supported using stored `@Procedure` query methods.

Using Named Parameters

By default, Spring Data JPA uses position-based parameter binding, as described in all the preceding examples. This makes query methods a little error-prone when refactoring regarding the parameter position. To solve this issue, you can use `@Param` annotation to give a method parameter a concrete name and bind the name in the query, as shown in the following example:

Example 14. Using named parameters

JAVA

```
public interface UserRepository extends JpaRepository<User, Long> {

  @Query("select u from User u where u.firstname = :firstname or u.lastname =
:lastname")
  User findByLastnameOrFirstname(@Param("lastname") String lastname,
                                 @Param("firstname") String firstname);
}
```

NOTE

The method parameters are switched according to their order in the defined query.

NOTE

As of version 4, Spring fully supports Java 8's parameter name discovery based on the `-parameters` compiler flag. By using this flag in your build as an alternative to debug information, you can omit the `@Param` annotation for named parameters.

Using Expressions

We support the usage of restricted expressions in manually defined queries that are defined with `@Query` . Upon the query being run, these expressions are evaluated against a predefined set of variables.

NOTE

If you are not familiar with Value Expressions, please refer to Value Expressions Fundamentals to learn about SpEL Expressions and Property Placeholders.

Spring Data JPA supports a variable called `entityName`. Its usage is `select x from #{#entityName} x`. It inserts the `entityName` of the domain type associated with the given repository. The `entityName` is resolved as follows: * If the domain type has set the name property on the `@Entity` annotation, it is used. * Otherwise, the simple class-name of the domain type is used.

The following example demonstrates one use case for the `#{#entityName}` expression in a query string where you want to define a repository interface with a query method and a manually defined query:

Example 15. Using SpEL expressions in repository query methods: entityName

JAVA

```
@Entity
public class User {

  @Id
  @GeneratedValue
  Long id;

  String lastname;
}

public interface UserRepository extends JpaRepository<User,Long> {

  @Query("select u from #{#entityName} u where u.lastname = ?1")
  List<User> findByLastname(String lastname);
}
```

To avoid stating the actual entity name in the query string of a `@Query` annotation, you can use the `#{#entityName}` variable.

NOTE

The `entityName` can be customized by using the `@Entity` annotation. Customizations in `orm.xml` are not supported for the SpEL expressions.

Of course, you could have just used `User` in the query declaration directly, but that would require you to change the query as well. The reference to `#entityName` picks up potential future remappings of the `User` class to a different entity name (for example, by using `@Entity(name = "MyUser")`).

Another use case for the `#{#entityName}` expression in a query string is if you want to define a generic repository interface with specialized repository interfaces for a concrete domain type. To not repeat the definition of custom query methods on the concrete interfaces, you can use the entity name expression in the query string of the `@Query` annotation in the generic repository interface, as shown in the following example:

Example 16. Using SpEL expressions in Repository Query Methods: entityName with Inheritance

```java
@MappedSuperclass
public abstract class AbstractMappedType {
  …
  String attribute;
}

@Entity
public class ConcreteType extends AbstractMappedType { … }

@NoRepositoryBean
public interface MappedTypeRepository<T extends AbstractMappedType>
  extends Repository<T, Long> {

  @Query("select t from #{#entityName} t where t.attribute = ?1")
  List<T> findAllByAttribute(String attribute);
}

public interface ConcreteRepository
  extends MappedTypeRepository<ConcreteType> { … }
```

In the preceding example, the `MappedTypeRepository` interface is the common parent interface for a few domain types extending `AbstractMappedType`. It also defines the generic `findAllByAttribute(…)` method, which can be used on instances of the specialized repository interfaces. If you now invoke `findAllByAttribute(…)` on `ConcreteRepository`, the query becomes `select t from ConcreteType t where t.attribute = ?1`.

You can also use Expressions to control arguments may also be used to control method arguments. In these expressions the entity name is not available, but the arguments are. They can be accessed by name or index as demonstrated in the following example.

Example 17. Using Value Expressions in Repository Query Methods: Accessing Arguments

```java
@Query("select u from User u where u.firstname = ?1 and u.firstname=?#{[0]} and
u.emailAddress = ?#{principal.emailAddress}")
List<User> findByFirstnameAndCurrentUserWithCustomQuery(String firstname);
```

For `like`-conditions one often wants to append `%` to the beginning or the end of a String valued parameter. This can be done by appending or prefixing a bind parameter marker or a SpEL expression with `%`. Again the following example demonstrates this.

Example 18. Using Value Expressions in Repository Query Methods: Wildcard shortcut

```java
@Query("select u from User u where u.lastname like %:#{[0]}% and u.lastname like
%:lastname%")
```

```
List<User> findByLastnameWithSpelExpression(@Param("lastname") String lastname);
```

When using `like` -conditions with values that are coming from a not secure source the values should be sanitized so they can't contain any wildcards and thereby allow attackers to select more data than they should be able to. For this purpose the `escape(String)` method is made available in the SpEL context. It prefixes all instances of `_` and `%` in the first argument with the single character from the second argument. In combination with the `escape` clause of the `like` expression available in JPQL and standard SQL this allows easy cleaning of bind parameters.

Example 19. Using Value Expressions in Repository Query Methods: Sanitizing Input Values

JAVA

```
@Query("select u from User u where u.firstname like %?#{escape([0])}% escape ?#{escapeCharacter()}")
List<User> findContainingEscaped(String namePart);
```

Given this method declaration in a repository interface `findContainingEscaped("Peter_")` will find `Peter_Parker` but not `Peter Parker`. The escape character used can be configured by setting the `escapeCharacter` of the `@EnableJpaRepositories` annotation. Note that the method `escape(String)` available in the SpEL context will only escape the SQL and JPQL standard wildcards `_` and `%`. If the underlying database or the JPA implementation supports additional wildcards these will not get escaped.

Example 20. Using Value Expressions in Repository Query Methods: Configuration Properties

JAVA

```
@Query("select u from User u where u.applicationName = ?${spring.application.name:unknown}")
List<User> findContainingEscaped(String namePart);
```

You can refer in your query methods also to configuration property names including fallbacks if you wish to resolve a property from `Environment` during runtime. The property is being evaluated upon query execution. Typically, property placeholders resolve to String-like values.

Other Methods

Spring Data JPA offers many ways to build queries. But sometimes, your query may simply be too complicated for the techniques offered. In that situation, consider:

- If you haven't already, simply write the query yourself using `@Query`.
- If that doesn't fit your needs, consider implementing a custom implementation. This lets you register a method in your repository while leaving the implementation completely up to you. This gives you the ability to:

- Talk directly to the `EntityManager` (writing pure HQL/JPQL/EQL/native SQL or using the **Criteria API**)
- Leverage Spring Framework's `JdbcTemplate` (native SQL)
- Use another 3rd-party database toolkit.

- Another option is putting your query inside the database and then using either Spring Data JPA's `@StoredProcedure` annotation or if it's a database function using the `@Query` annotation and invoking it with a `CALL`.

These tactics may be most effective when you need maximum control of your query, while still letting Spring Data JPA provide resource management.

Modifying Queries

All the previous sections describe how to declare queries to access a given entity or collection of entities. You can add custom modifying behavior by using the custom method facilities described in Custom Implementations for Spring Data Repositories. As this approach is feasible for comprehensive custom functionality, you can modify queries that only need parameter binding by annotating the query method with `@Modifying`, as shown in the following example:

Example 21. Declaring manipulating queries

```java
@Modifying
@Query("update User u set u.firstname = ?1 where u.lastname = ?2")
int setFixedFirstnameFor(String firstname, String lastname);
```

Doing so triggers the query annotated to the method as an updating query instead of a selecting one. As the `EntityManager` might contain outdated entities after the execution of the modifying query, we do not automatically clear it (see the JavaDoc of `EntityManager.clear()` for details), since this effectively drops all non-flushed changes still pending in the `EntityManager`. If you wish the `EntityManager` to be cleared automatically, you can set the `@Modifying` annotation's `clearAutomatically` attribute to `true`.

The `@Modifying` annotation is only relevant in combination with the `@Query` annotation. Derived query methods or custom methods do not require this annotation.

Derived Delete Queries

Spring Data JPA also supports derived delete queries that let you avoid having to declare the JPQL query explicitly, as shown in the following example:

Example 22. Using a derived delete query

```java
interface UserRepository extends Repository<User, Long> {

  void deleteByRoleId(long roleId);

  @Modifying
  @Query("delete from User u where u.role.id = ?1")
  void deleteInBulkByRoleId(long roleId);
}
```

Although the `deleteByRoleId(…)` method looks like it basically produces the same result as the `deleteInBulkByRoleId(…)`, there is an important difference between the two method declarations in terms of the way they are run. As the name suggests, the latter method issues a single JPQL query (the one defined in the annotation) against the database. This means even currently loaded instances of `User` do not see lifecycle callbacks invoked.

To make sure lifecycle queries are actually invoked, an invocation of `deleteByRoleId(…)` runs a query and then deletes the returned instances one by one, so that the persistence provider can actually invoke `@PreRemove` callbacks on those entities.

In fact, a derived delete query is a shortcut for running the query and then calling `CrudRepository.delete(Iterable<User> users)` on the result and keeping behavior in sync with the implementations of other `delete(…)` methods in `CrudRepository`.

NOTE

When deleting a lot of objects you will need to consider the performance implications to ensure sufficient memory availability. All resulting objects are loaded into memory before being deleted and are held in the session until flushing or completing the transaction.

Applying Query Hints

To apply JPA query hints to the queries declared in your repository interface, you can use the `@QueryHints` annotation. It takes an array of JPA `@QueryHint` annotations plus a boolean flag to potentially disable the hints applied to the additional count query triggered when applying pagination, as shown in the following example:

Example 23. Using QueryHints with a repository method

```java
public interface UserRepository extends Repository<User, Long> {

  @QueryHints(value = { @QueryHint(name = "name", value = "value")},
              forCounting = false)
  Page<User> findByLastname(String lastname, Pageable pageable);
}
```

The preceding declaration would apply the configured `@QueryHint` for that actually query but omit applying it to the count query triggered to calculate the total number of pages.

Adding Comments to Queries

Sometimes, you need to debug a query based upon database performance. The query your database administrator shows you may look VERY different than what you wrote using `@Query`, or it may look nothing like what you presume Spring Data JPA has generated regarding a custom finder or if you used query by example.

To make this process easier, you can insert custom comments into almost any JPA operation, whether its a query or other operation by applying the `@Meta` annotation.

Example 24. Apply `@Meta` annotation to repository operations

JAVA

```
public interface RoleRepository extends JpaRepository<Role, Integer> {

    @Meta(comment = "find roles by name")
    List<Role> findByName(String name);

    @Override
    @Meta(comment = "find roles using QBE")
    <S extends Role> List<S> findAll(Example<S> example);

    @Meta(comment = "count roles for a given name")
    long countByName(String name);

    @Override
    @Meta(comment = "exists based on QBE")
    <S extends Role> boolean exists(Example<S> example);
}
```

This sample repository has a mixture of custom finders as well as overriding the inherited operations from `JpaRepository`. Either way, the `@Meta` annotation lets you add a `comment` that will be inserted into queries before they are sent to the database.

It's also important to note that this feature isn't confined solely to queries. It extends to the `count` and `exists` operations. And while not shown, it also extends to certain `delete` operations.

IMPORTANT

While we have attempted to apply this feature everywhere possible, some operations of the underlying `EntityManager` don't support comments. For example, `entityManager.createQuery()` is clearly documented as supporting comments, but `entityManager.find()` operations do not.

Neither JPQL logging nor SQL logging is a standard in JPA, so each provider requires custom configuration, as shown the sections below.

Activating Hibernate comments

To activate query comments in Hibernate, you must set `hibernate.use_sql_comments` to `true`.

If you are using Java-based configuration settings, this can be done like this:

Example 25. Java-based JPA configuration

```java
@Bean
public Properties jpaProperties() {

    Properties properties = new Properties();
    properties.setProperty("hibernate.use_sql_comments", "true");
    return properties;
}
```

If you have a `persistence.xml` file, you can apply it there:

Example 26. `persistence.xml`-based configuration

```xml
<persistence-unit name="my-persistence-unit">

   ...registered classes...

    <properties>
        <property name="hibernate.use_sql_comments" value="true" />
    </properties>
</persistence-unit>
```

Finally, if you are using Spring Boot, then you can set it up inside your `application.properties` file:

Example 27. Spring Boot property-based configuration

```
spring.jpa.properties.hibernate.use_sql_comments=true
```

Activating EclipseLink comments

To activate query comments in EclipseLink, you must set `eclipselink.logging.level.sql` to `FINE`.

If you are using Java-based configuration settings, this can be done like this:

Example 28. Java-based JPA configuration

```java
@Bean
public Properties jpaProperties() {

    Properties properties = new Properties();
    properties.setProperty("eclipselink.logging.level.sql", "FINE");
    return properties;
}
```

If you have a `persistence.xml` file, you can apply it there:

Example 29. `persistence.xml`-based configuration

```xml
<persistence-unit name="my-persistence-unit">

   ...registered classes...

    <properties>
        <property name="eclipselink.logging.level.sql" value="FINE" />
    </properties>
</persistence-unit>
```

Finally, if you are using Spring Boot, then you can set it up inside your `application.properties` file:

Example 30. Spring Boot property-based configuration

```
spring.jpa.properties.eclipselink.logging.level.sql=FINE
```

Configuring Fetch- and LoadGraphs

The JPA 2.1 specification introduced support for specifying Fetch- and LoadGraphs that we also support with the `@EntityGraph` annotation, which lets you reference a `@NamedEntityGraph` definition. You can use that annotation on an entity to configure the fetch plan of the resulting query. The type (`Fetch` or `Load`) of the fetching can be configured by using the `type` attribute on the `@EntityGraph` annotation. See the JPA 2.1 Spec 3.7.4 for further reference.

The following example shows how to define a named entity graph on an entity:

Example 31. Defining a named entity graph on an entity.

```java
@Entity
@NamedEntityGraph(name = "GroupInfo.detail",
```

```
  attributeNodes = @NamedAttributeNode("members"))
public class GroupInfo {

  // default fetch mode is lazy.
  @ManyToMany
  List<GroupMember> members = new ArrayList<GroupMember>();

  …
}
```

The following example shows how to reference a named entity graph on a repository query method:

Example 32. Referencing a named entity graph definition on a repository query method.

JAVA

```
public interface GroupRepository extends CrudRepository<GroupInfo, String> {

  @EntityGraph(value = "GroupInfo.detail", type = EntityGraphType.LOAD)
  GroupInfo getByGroupName(String name);

}
```

It is also possible to define ad hoc entity graphs by using `@EntityGraph`. The provided `attributePaths` are translated into the according `EntityGraph` without needing to explicitly add `@NamedEntityGraph` to your domain types, as shown in the following example:

Example 33. Using ad-hoc entity graph definitions on a repository query method

JAVA

```
public interface GroupRepository extends CrudRepository<GroupInfo, String> {

  @EntityGraph(attributePaths = { "members" })
  GroupInfo getByGroupName(String name);

}
```

Scrolling

Scrolling is a more fine-grained approach to iterate through larger results set chunks. Scrolling consists of a stable sort, a scroll type (Offset- or Keyset-based scrolling) and result limiting. You can define simple sorting expressions by using property names and define static result limiting using the `Top` or `First` keyword through query derivation. You can concatenate expressions to collect multiple criteria into one expression.

Scroll queries return a `Window<T>` that allows obtaining the element's scroll position to fetch the next `Window<T>` until your application has consumed the entire query result. Similar to consuming a

Java `Iterator<List<…>>` by obtaining the next batch of results, query result scrolling lets you access the a `ScrollPosition` through `Window.positionAt(…)`.

```java
Window<User> users = repository.findFirst10ByLastnameOrderByFirstname("Doe",
ScrollPosition.offset());
do {

  for (User u : users) {
    // consume the user
  }

  // obtain the next Scroll
  users = repository.findFirst10ByLastnameOrderByFirstname("Doe",
users.positionAt(users.size() - 1));
} while (!users.isEmpty() && users.hasNext());
```

NOTE

The `ScrollPosition` identifies the exact position of an element with the entire query result. Query execution treats the position parameter *exclusive*, results will start *after* the given position.
`ScrollPosition#offset()` and `ScrollPosition#keyset()` as special incarnations of a `ScrollPosition` indicating the start of a scroll operation.

NOTE

The above example shows static sorting and limiting. You can define query methods alternatively that accept a `Sort` object define a more complex sorting order or sorting on a per-request basis. In a similar way, providing a `Limit` object allows you to define a dynamic limit on a per-request basis instead of applying a static limitation. Read more on dynamic sorting and limiting in the Query Methods Details.

`WindowIterator` provides a utility to simplify scrolling across `Window`s by removing the need to check for the presence of a next `Window` and applying the `ScrollPosition`.

```java
WindowIterator<User> users = WindowIterator.of(position ->
repository.findFirst10ByLastnameOrderByFirstname("Doe", position))
  .startingAt(ScrollPosition.offset());

while (users.hasNext()) {
  User u = users.next();
  // consume the user
}
```

Scrolling using Offset

Offset scrolling uses similar to pagination, an Offset counter to skip a number of results and let the data source only return results beginning at the given Offset. This simple mechanism avoids large re-

sults being sent to the client application. However, most databases require materializing the full query result before your server can return the results.

Example 34. Using `OffsetScrollPosition` with Repository Query Methods

```java
interface UserRepository extends Repository<User, Long> {

  Window<User> findFirst10ByLastnameOrderByFirstname(String lastname,
OffsetScrollPosition position);
}

WindowIterator<User> users = WindowIterator.of(position ->
repository.findFirst10ByLastnameOrderByFirstname("Doe", position))
  .startingAt(OffsetScrollPosition.initial());  1
```

1 Start with no offset to include the element at position `0`.

CAUTION

There is a difference between `ScrollPosition.offset()` and `ScrollPosition.offset(0L)`. The former indicates the start of scroll operation, pointing to no specific offset whereas the latter identifies the first element (at position `0`) of the result. Given the *exclusive* nature of scrolling, using `ScrollPosition.offset(0)` skips the first element and translate to an offset of `1`.

Scrolling using Keyset-Filtering

Offset-based requires most databases require materializing the entire result before your server can return the results. So while the client only sees the portion of the requested results, your server needs to build the full result, which causes additional load.

Keyset-Filtering approaches result subset retrieval by leveraging built-in capabilities of your database aiming to reduce the computation and I/O requirements for individual queries. This approach maintains a set of keys to resume scrolling by passing keys into the query, effectively amending your filter criteria.

The core idea of Keyset-Filtering is to start retrieving results using a stable sorting order. Once you want to scroll to the next chunk, you obtain a `ScrollPosition` that is used to reconstruct the position within the sorted result. The `ScrollPosition` captures the keyset of the last entity within the current `Window`. To run the query, reconstruction rewrites the criteria clause to include all sort fields and the primary key so that the database can leverage potential indexes to run the query. The database needs only constructing a much smaller result from the given keyset position without the need to fully materialize a large result and then skipping results until reaching a particular offset.

WARNING

WARNING

Keyset-Filtering requires the keyset properties (those used for sorting) to be non-nullable. This limitation applies due to the store specific `null` value handling of comparison operators as well as the need to run queries against an indexed source. Keyset-Filtering on nullable properties will lead to unexpected results.

Using `KeysetScrollPosition` with Repository Query Methods

```
JAVA
interface UserRepository extends Repository<User, Long> {

  Window<User> findFirst10ByLastnameOrderByFirstname(String lastname,
KeysetScrollPosition position);
}

WindowIterator<User> users = WindowIterator.of(position ->
repository.findFirst10ByLastnameOrderByFirstname("Doe", position))
  .startingAt(ScrollPosition.keyset());  1
```

1 Start at the very beginning and do not apply additional filtering.

Keyset-Filtering works best when your database contains an index that matches the sort fields, hence a static sort works well. Scroll queries applying Keyset-Filtering require to the properties used in the sort order to be returned by the query, and these must be mapped in the returned entity.

You can use interface and DTO projections, however make sure to include all properties that you've sorted by to avoid keyset extraction failures.

When specifying your `Sort` order, it is sufficient to include sort properties relevant to your query; You do not need to ensure unique query results if you do not want to. The keyset query mechanism amends your sort order by including the primary key (or any remainder of composite primary keys) to ensure each query result is unique.

Copyright © 2005 - 2025 Broadcom. All Rights Reserved. The term "Broadcom" refers to Broadcom Inc. and/or its subsidiaries.
Terms of Use • Privacy • Trademark Guidelines • Thank you • Your California Privacy Rights • Cookie Settings

Apache®, Apache Tomcat®, Apache Kafka®, Apache Cassandra™, and Apache Geode™ are trademarks or registered trademarks of the Apache Software Foundation in the United States and/or other countries. Java™, Java™ SE, Java™ EE, and OpenJDK™ are trademarks of Oracle and/or its affiliates. Kubernetes® is a registered trademark of the Linux Foundation in the United States and other countries. Linux® is the registered trademark of Linus Torvalds in the United States and other countries. Windows® and Microsoft® Azure are registered trademarks of Microsoft Corporation. "AWS" and "Amazon Web Services" are trademarks or registered trademarks of Amazon.com Inc. or its affiliates. All other trademarks and copyrights are property of their respective owners and are only mentioned for informative purposes. Other names may be trademarks of their respective owners.