5.2. Persisting entities

5.2.1. Saving entities

Saving an entity can be performed via the `CrudRepository.save(…)`-Method. It will persist or merge the given entity using the underlying JPA `EntityManager`. If the entity has not been persisted yet Spring Data JPA will save the entity via a call to the `entityManager.persist(…)` method, otherwise the `entityManager.merge(…)` method will be called.

Entity state detection strategies

Spring Data JPA offers the following strategies to detect whether an entity is new or not:

Table 3. Options for detection whether an entity is new in Spring Data JPA

Id-Property inspection (**default**)	By default Spring Data JPA inspects the identifier property of the given entity. If the identifier property is `null`, then the entity will be assumed as new, otherwise as not new.
Implementing `Persistable`	If an entity implements `Persistable`, Spring Data JPA will delegate the new detection to the `isNew(…)` method of the entity. See the JavaDoc for details.
Implementing `EntityInformation`	You can customize the `EntityInformation` abstraction used in the `SimpleJpaRepository` implementation by creating a subclass of `JpaRepositoryFactory` and overriding the `getEntityInformation(…)` method accordingly. You then have to register the custom implementation of `JpaRepositoryFactory` as a Spring bean. Note that this should be rarely necessary. See the JavaDoc for details.

5.3. Query methods

5.3.1. Query lookup strategies

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

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 NamedQueries for more information) or rather annotate your query method with `@Query` (see Using @Query for details).

5.3.2. Query creation

Generally the query creation mechanism for JPA works as described in Query methods. Here's a short example of what a JPA query method translates into:

Example 43. Query creation from method names

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

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

We will 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 will do a property check and traverse nested properties as described in Property expressions. Here's an overview of the keywords supported for JPA and what a method containing that keyword essentially translates to.

Table 4. Supported keywords inside method names

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`

Keyword	Sample	JPQL snippet
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)

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

5.3.3. Using JPA NamedQueries

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

XML named query definition

To use XML configuration simply add the necessary <named-query /> element to the orm.xml JPA configuration file located in 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 44. XML named query configuration

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

As you can see the query has a special name which will be used to resolve it at runtime.

Annotation configuration

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

Example 45. Annotation based named query configuration

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

}
```

Declaring interfaces

To allow execution of these named queries all you need to do is to specify the `UserRepository` as follows:

Example 46. Query method declaration in UserRepository

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

  List<User> findByLastname(String lastname);

  User findByEmailAddress(String emailAddress);
}
```

Spring Data will try 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 example here would use the named queries defined above instead of trying to create a query from the method name.

5.3.4. 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 executes them you actually can bind them directly using the Spring Data JPA `@Query` annotation rather than annotating them to the domain class. This will free the domain class from persistence specific information and co-locate the query to the repository interface.

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

Example 47. Declare query at the query method using @Query

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

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

Using advanced `LIKE` expressions

The query execution mechanism for manually defined queries using @Query allows the definition of advanced `LIKE` expressions inside the query definition.

Example 48. Advanced like-expressions in @Query

```
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 just shown sample `LIKE` delimiter character `%` is recognized and the query transformed into a valid JPQL query (removing the `%`). Upon query execution the parameter handed into the method call gets augmented with the previously recognized `LIKE` pattern.

Native queries

The `@Query` annotation allows to execute native queries by setting the `nativeQuery` flag to true.

Example 49. Declare a native query at the query method using @Query

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

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

Note, that we currently don't support execution of dynamic sorting for native queries as we'd have to manipulate the actual query declared and we cannot do this reliably for native SQL. You can however use native queries for pagination by specifying the count query yourself:

Example 50. Declare native count queries for pagination at the query method using @Query

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

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

This also works with named native queries by adding the suffix `.count` to a copy of your query. Be aware that you probably must register a result set mapping for your count query, though.

5.3.5. Using Sort

Sorting can be done be either providing a `PageRequest` or using `Sort` directly. The properties actually used within the `Order` instances of `Sort` need to match to 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 Using any non referenceable path expression leads to an Exception.

Using `Sort` together with `@Query` however allows you to sneak in non path checked `Order` instances containing *functions* within the `ORDER BY` clause. This is possible because the `Order` is just appended to the given query string. By default we will reject any `Order` instance containing function calls, but you can use `JpaSort.unsafe` to add potentially unsafe ordering.

Example 51. Using Sort and JpaSort

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

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

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

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

④ Valid `Sort` expression pointing to aliased function.

5.3.6. Using named parameters

By default Spring Data JPA will use position based parameter binding as described in all the samples above. This makes query methods a little error prone to 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.

Example 52. Using named parameters

```
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 that the method parameters are switched according to the occurrence in the query defined.

5.3.7. Using SpEL expressions

As of Spring Data JPA release 1.4 we support the usage of restricted SpEL template expressions in manually defined queries via `@Query`. Upon query execution these expressions are evaluated against

a predefined set of variables. We support the following list of variables to be used in a manual query.

Table 5. Supported variables inside SpEL based query templates

Variable	Usage	Description
`entityName`	`select x from #{#entityName} x`	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 then it will be used. Otherwise the simple class-name of the domain type will be 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 with a manually defined query. In order not to have to state the actual entity name in the query string of a `@Query` annotation one can use the `#{#entityName}` Variable.

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

Example 53. Using SpEL expressions in repository query methods - entityName

```
@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);
}
```

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` will pick up potential future remappings of the User class to a different entity name (e.g. 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. In order not to have to 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.

Example 54. Using SpEL expressions in repository query methods - entityName with inheritance

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

5.3.8. Modifying queries

All the sections above describe how to declare queries to access a given entity or collection of entities. Of course you can add custom modifying behaviour by using facilities described in Custom implementations for Spring Data repositories. As this approach is feasible for comprehensive custom functionality, you can achieve the execution of modifying queries that actually only need parameter binding by annotating the query method with `@Modifying`:

Example 55. Declaring manipulating queries

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

This will trigger the query annotated to the method as 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 JavaDoc of `EntityManager.clear()` for details) since this will effectively drop all non-flushed changes still pending in the `EntityManager`. If you wish the

`EntityManager` to be cleared automatically you can set `@Modifying` annotation's `clearAutomatically` attribute to `true`.

Derived delete queries

Spring Data JPA also supports derived delete queries that allow you to avoid having to declare the JPQL query explicitly.

Example 56. Using a derived delete query

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

  void deleteByRoleId(long roleId);

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

Although the `deleteByRoleId(…)` method looks like it's basically producing the same result as the `deleteInBulkByRoleId(…)`, there is an important difference between the two method declarations in terms of the way they get executed. As the name suggests, the latter method will issue a single JPQL query (i.e. the one defined in the annotation) against the database. This means, even currently loaded instances of `User` won't see lifecycle callbacks invoked.

To make sure lifecycle queries are actually invoked, an invocation of `deleteByRoleId(…)` will actually execute a query and then deleting 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 executing the query and then calling `CrudRepository.delete(Iterable<User> users)` on the result and keep behavior in sync with the implementations of other `delete(…)` methods in `CrudRepository`.

5.3.9. 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 addtional count query triggered when applying pagination.

Example 57. Using QueryHints with a repository method

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

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

The just shown 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.

5.3.10. Configuring Fetch- and LoadGraphs

The JPA 2.1 specification introduced support for specifiying Fetch- and LoadGraphs that we also support via the `@EntityGraph` annotation which allows to reference a `@NamedEntityGraph` definition, that can be annotated on an entity, to be used to configure the fetch plan of the resulting query. The type (Fetch / Load) of the fetching can be configured via the `type` attribute on the `@EntityGraph` annotation. Please have a look at the JPA 2.1 Spec 3.7.4 for further reference.

Example 58. Defining a named entity graph on an entity.

```
@Entity
@NamedEntityGraph(name = "GroupInfo.detail",
  attributeNodes = @NamedAttributeNode("members"))
public class GroupInfo {

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

  …
}
```

Example 59. Referencing a named entity graph definition on an repository query method.

```
@Repository
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 via `@EntityGraph`. The provided `attributePaths` will be translated into the according `EntityGraph` without the need of having to explicitly add `@NamedEntityGraph` to your domain types.

Example 60. Using AD-HOC entity graph definition on an repository query method.

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

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

}
```

5.3.11. Projections

Spring Data Repositories usually return the domain model when using query methods. However, sometimes, you may need to alter the view of that model for various reasons. In this section, you will learn how to define projections to serve up simplified and reduced views of resources.

Look at the following domain model:

```
@Entity
public class Person {

  @Id @GeneratedValue
  private Long id;
  private String firstName, lastName;

  @OneToOne
  private Address address;
  …
}

@Entity
public class Address {

  @Id @GeneratedValue
  private Long id;
  private String street, state, country;

  …
}
```

This `Person` has several attributes:

- `id` is the primary key

- `firstName` and `lastName` are data attributes
- `address` is a link to another domain object

Now assume we create a corresponding repository as follows:

```
interface PersonRepository extends CrudRepository<Person, Long> {

  Person findPersonByFirstName(String firstName);
}
```

Spring Data will return the domain object including all of its attributes. There are two options just to retrieve the `address` attribute. One option is to define a repository for `Address` objects like this:

```
interface AddressRepository extends CrudRepository<Address, Long> {}
```

In this situation, using `PersonRepository` will still return the whole `Person` object. Using `AddressRepository` will return just the `Address`.

However, what if you do not want to expose `address` details at all? You can offer the consumer of your repository service an alternative by defining one or more projections.

Example 61. Simple Projection

```
interface NoAddresses {  ①

  String getFirstName(); ②

  String getLastName();  ③
}
```

This projection has the following details:

① A plain Java interface making it declarative.

② Export the `firstName`.

③ Export the `lastName`.

The `NoAddresses` projection only has getters for `firstName` and `lastName` meaning that it will not serve up any address information. The query method definition returns in this case `NoAdresses` instead of `Person`.

```
interface PersonRepository extends CrudRepository<Person, Long> {

  NoAddresses findByFirstName(String firstName);
}
```

Projections declare a contract between the underlying type and the method signatures related to the exposed properties. Hence it is required to name getter methods according to the property name of the underlying type. If the underlying property is named `firstName`, then the getter method must be named `getFirstName` otherwise Spring Data is not able to look up the source property. This type of projection is also called *closed projection*. Closed projections expose a subset of properties hence they can be used to optimize the query in a way to reduce the selected fields from the data store. The other type is, as you might imagine, an *open projection*.

Remodelling data

So far, you have seen how projections can be used to reduce the information that is presented to the user. Projections can be used to adjust the exposed data model. You can add virtual properties to your projection. Look at the following projection interface:

Example 62. Renaming a property

```
interface RenamedProperty {    ①

  String getFirstName();       ②

  @Value("#{target.lastName}")
  String getName();            ③
}
```

This projection has the following details:

① A plain Java interface making it declarative.

② Export the `firstName`.

③ Export the `name` property. Since this property is virtual it requires `@Value("#{target.lastName}")` to specify the property source.

The backing domain model does not have this property so we need to tell Spring Data from where this property is obtained. Virtual properties are the place where `@Value` comes into play. The `name` getter is annotated with `@Value` to use SpEL expressions pointing to the backing property `lastName`. You may have noticed `lastName` is prefixed with `target` which is the variable name pointing to the backing object. Using `@Value` on methods allows defining where and how the value is obtained.

Some applications require the full name of a person. Concatenating strings with `String.format("%s %s", person.getFirstName(), person.getLastName())` would be one possibility but this piece of code needs to be called in every place the full name is required. Virtual properties on projections leverage the need for repeating that code all over.

```
interface FullNameAndCountry {

  @Value("#{target.firstName} #{target.lastName}")
  String getFullName();

  @Value("#{target.address.country}")
  String getCountry();
}
```

In fact, `@Value` gives full access to the target object and its nested properties. SpEL expressions are extremly powerful as the definition is always applied to the projection method. Let's take SpEL expressions in projections to the next level.

Imagine you had the following domain model definition:

```
@Entity
public class User {

  @Id @GeneratedValue
  private Long id;
  private String name;

  private String password;
  …
}
```

IMPORTANT

This example may seem a bit contrived, but it is possible with a richer domain model and many projections, to accidentally leak such details. Since Spring Data cannot discern the sensitivity of such data, it is up to the developers to avoid such situations. Storing a password as plain-text is discouraged. You really should not do this. For this example, you could also replace `password` with anything else that is secret.

In some cases, you might keep the `password` as secret as possible and not expose it more than it should be. The solution is to create a projection using `@Value` together with a SpEL expression.

```
interface PasswordProjection {
  @Value("#{(target.password == null || target.password.empty) ? null : '******'}")
  String getPassword();
}
```

The expression checks whether the password is `null` or empty and returns `null` in this case, otherwise six asterisks to indicate a password was set.

5.4. Stored procedures

The JPA 2.1 specification introduced support for calling stored procedures via the JPA criteria query API. We Introduced the `@Procedure` annotation for declaring stored procedure metadata on a repository method.

Example 63. The definition of the pus1inout procedure in HSQL DB.

```
/;
DROP procedure IF EXISTS plus1inout
/;
CREATE procedure plus1inout (IN arg int, OUT res int)
BEGIN ATOMIC
 set res = arg + 1;
END
/;
```

Metadata for stored procedures can be configured via the `NamedStoredProcedureQuery` annotation on an entity type.

Example 64. StoredProcedure metadata definitions on an entity.

```
@Entity
@NamedStoredProcedureQuery(name = "User.plus1", procedureName = "plus1inout",
parameters = {
  @StoredProcedureParameter(mode = ParameterMode.IN, name = "arg", type = Integer
.class),
  @StoredProcedureParameter(mode = ParameterMode.OUT, name = "res", type =
Integer.class) })
public class User {}
```

Stored procedures can be referenced from a repository method in multiple ways. The stored procedure to be called can either be defined directly via the `value` or `procedureName` attribute of the `@Procedure` annotation or indirectly via the `name` attribute. If no name is configured the name of the repository method is used as a fallback.

Example 65. Referencing explicitly mapped procedure with name "plus1inout" in database.

```
@Procedure("plus1inout")
Integer explicitlyNamedPlus1inout(Integer arg);
```