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: