4. Working with Spring Data Repositories

The goal of the Spring Data repository abstraction is to significantly reduce the amount of boilerplate code required to implement data access layers for various persistence stores.

4.1. Core concepts

The central interface in the Spring Data repository abstraction is `Repository`. It takes the domain class to manage as well as the ID type of the domain class as type arguments. This interface acts primarily as a marker interface to capture the types to work with and to help you to discover interfaces that extend this one. The `CrudRepository` provides sophisticated CRUD functionality for the entity class that is being managed.

Example 3. `CrudRepository` **interface**

```java
public interface CrudRepository<T, ID> extends Repository<T, ID> {

  <S extends T> S save(S entity);      // (1)

  Optional<T> findById(ID primaryKey); // (2)

  Iterable<T> findAll();               // (3)

  long count();                        // (4)

  void delete(T entity);               // (5)

  boolean existsById(ID primaryKey);   // (6)

  // … more functionality omitted.
}
```

1. Saves the given entity.
2. Returns the entity identified by the given ID.
3. Returns all entities.
4. Returns the number of entities.
5. Deletes the given entity.
6. Indicates whether an entity with the given ID exists.

On top of the `CrudRepository`, there is a `PagingAndSortingRepository` abstraction that adds additional methods to ease paginated access to entities:

Example 4. `PagingAndSortingRepository` **interface**

```
public interface PagingAndSortingRepository<T, ID> extends CrudRepository<T, ID> {

  Iterable<T> findAll(Sort sort);

  Page<T> findAll(Pageable pageable);
}
```

To access the second page of `User` by a page size of 20, you could do something like the following:

```
PagingAndSortingRepository<User, Long> repository = // … get access to a bean
Page<User> users = repository.findAll(PageRequest.of(1, 20));
```

In addition to query methods, query derivation for both count and delete queries is available. The following list shows the interface definition for a derived count query:

Example 5. Derived Count Query

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

  long countByLastname(String lastname);
}
```

The following list shows the interface definition for a derived delete query:

Example 6. Derived Delete Query

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

  long deleteByLastname(String lastname);

  List<User> removeByLastname(String lastname);
}
```

4.2. Query methods

Standard CRUD functionality repositories usually have queries on the underlying datastore. With Spring Data, declaring those queries becomes a 3-step process:

1. Declare an interface extending Repository or one of its subinterfaces and type it to the domain class and ID type that it should handle, as shown in the following example:

```
interface PersonRepository extends Repository<Person, Long> { … }
```

2. Declare query methods on the interface.

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

  List<Person> findByLastname(String lastname);

}
```

3. Inject the repository instance and use it, as shown in the following example:

```
class SomeClient {


  private final PersonRepository repository;


  SomeClient(PersonRepository repository) {

    this.repository = repository;

  }


  void doSomething() {

    List<Person> persons = repository.findByLastname("Matthews");

  }

}
```

The sections that follow explain each step in detail:

4.4. Defining Query Methods

The repository proxy has two ways to derive a store-specific query from the method name:

- By deriving the query from the method name directly.
- By using a manually defined query.

Available options depend on the actual store.

4.4.2. Query Creation

The query builder mechanism built into Spring Data repository infrastructure is useful for building constraining queries over entities of the repository. The mechanism strips the prefixes `find…By`, `read…By`, `query…By`, `count…By`, and `get…By` from the method and starts parsing the rest of it. The introducing clause can contain further expressions, such as a `Distinct` to set a distinct flag on the query to be created. However, the first `By` acts as delimiter to indicate the start of the actual criteria. At a very basic level, you can define conditions on entity properties and concatenate them with `And` and `Or`. The following example shows how to create a number of queries:

Example 13. Query creation from method names

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

  List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);

  // Enables the distinct flag for the query
  List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);
  List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);

  // Enabling ignoring case for an individual property
  List<Person> findByLastnameIgnoreCase(String lastname);
  // Enabling ignoring case for all suitable properties
  List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);
  // Enabling static ORDER BY for a query
  List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
  List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
}
```

The actual result of parsing the method depends on the persistence store for which you create the query. However, there are some general things to notice:

- The expressions are usually property traversals combined with operators that can be concatenated. You can combine property expressions with `AND` and `OR`. You also get support for operators such as `Between`, `LessThan`, `GreaterThan`, and `Like` for the property expressions. The supported operators can vary by datastore, so consult the appropriate part of your reference documentation.
- The method parser supports setting an `IgnoreCase` flag for individual properties (for example, `findByLastnameIgnoreCase(…)`) or for all properties of a type that supports ignoring case (usually `String` instances — for example, `findByLastnameAndFirstnameAllIgnoreCase(…)`). Whether ignoring cases is supported may vary by store, so consult the relevant sections in the reference documentation for the store-specific query method.
- You can apply static ordering by appending an `OrderBy` clause to the query method that references a property and by providing a sorting direction (`Asc` or `Desc`).

4.4.4. Special parameter handling

To handle parameters in your query, define method parameters as already seen in the preceding examples. Besides that, the infrastructure recognizes certain specific types like `Pageable` and `Sort`, to apply pagination and sorting to your queries dynamically. The following example demonstrates these features:

Example 14. Using `Pageable`, `Slice`, and `Sort` in query methods

```
Page<User> findByLastname(String lastname, Pageable pageable);

Slice<User> findByLastname(String lastname, Pageable pageable);

List<User> findByLastname(String lastname, Sort sort);

List<User> findByLastname(String lastname, Pageable pageable);
```

The first method lets you pass an `org.springframework.data.domain.Pageable` instance to the query method to

dynamically add paging to your statically defined query. A `Page` knows about the total number of elements and pages available. It does so by the infrastructure triggering a count query to calculate the overall number. As this might be expensive (depending on the store used), you can instead return a `Slice`. A `Slice` only knows about whether a next `Slice` is available, which might be sufficient when walking through a larger result set.
Sorting options are handled through the `Pageable` instance, too. If you only need sorting, add an `org.springframework.data.domain.Sort` parameter to your method. As you can see, returning a `List` is also possible. In this case, the additional metadata required to build the actual `Page` instance is not created (which, in turn, means that the additional count query that would have been necessary is not issued). Rather, it restricts the query to look up only the given range of entities.

Paging and Sorting

Simple sorting expressions can be defined by using property names. Expressions can be concatenated to collect multiple criterias into one expression.

Example 15. Defining sort expressions

```
Sort sort = Sort.by("firstname").ascending()
  .and(Sort.by("lastname").descending());
```

For a more type-safe way of defining sort expressions, start with the type to define the sort expression for and use method references to define the properties to sort on.

Example 16. Defining sort expressions using the type-safe API

```
TypedSort<Person> person = Sort.sort(Person.class);


TypedSort<Person> sort = person.by(Person::getFirstname).ascending()
  .and(person.by(Person::getLastname).descending());
```

If your store implementation supports Querydsl, you can also use the metamodel types generated to define sort expressions:

Example 17. Defining sort expressions using the Querydsl API

```
QSort sort = QSort.by(QPerson.firstname.asc())
  .and(QSort.by(QPerson.lastname.desc()));
```

4.4.5. Limiting Query Results

The results of query methods can be limited by using the `first` or `top` keywords, which can be used interchangeably. An optional numeric value can be appended to `top` or `first` to specify the maximum result size to be returned. If the number is left out, a result size of 1 is assumed. The following example shows how to limit the query size:

Example 18. Limiting the result size of a query with `Top` and `First`

```
User findFirstByOrderByLastnameAsc();

User findTopByOrderByAgeDesc();

Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);

Slice<User> findTop3ByLastname(String lastname, Pageable pageable);

List<User> findFirst10ByLastname(String lastname, Sort sort);

List<User> findTop10ByLastname(String lastname, Pageable pageable);
```

The limiting expressions also support the `Distinct` keyword. Also, for the queries limiting the result set to one instance, wrapping the result into with the `Optional` keyword is supported.

If pagination or slicing is applied to a limiting query pagination (and the calculation of the number of pages available), it is applied within the limited result.

4.4.8. Streaming query results

The results of query methods can be processed incrementally by using a Java 8 `Stream<T>` as return type. Instead of wrapping the query results in a `Stream` data store-specific methods are used to perform the streaming, as shown in the following example:

Example 23. Stream the result of a query with Java 8 `Stream<T>`

```
@Query("select u from User u")
Stream<User> findAllByCustomQueryAndStream();
```

```
Stream<User> readAllByFirstnameNotNull();

@Query("select u from User u")
Stream<User> streamAllPaged(Pageable pageable);
```

5.3.2. 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 57. Query creation from method names

```
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 3. 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`

Keyword	Sample	JPQL snippet
		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 not null
Like	findByFirstnameLike	… where x.firstname like ?1
NotLike	findByFirstnameNotLike	… where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	… where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	… where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	… where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	… where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	… where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	… where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> ages)	… where x.age not in ?1

Keyword	Sample	JPQL snippet
True	findByActiveTrue()	… where x.active = true
False	findByActiveFalse()	… where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	… where UPPER(x.firstame) = UPPER(?1)

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 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 61. 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 created with `@Query` allows the definition of advanced `LIKE` expressions inside the query definition, as shown in the following example:

Example 62. 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 preceding example, the `LIKE` delimiter character (`%`) is recognized, and the query is transformed into a valid JPQL query (removing the `%`). Upon query execution, the parameter passed to the method call gets augmented with the previously recognized `LIKE` pattern.

Native Queries

The `@Query` annotation allows for running native queries by setting the `nativeQuery` flag to true, as shown in the following example:

Example 63. 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);
}
```

5.3.5. Using Sort

Sorting can be done be 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.

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 65. 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", 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 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 66. 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);
}
```

5.3.8. 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 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 72. Declaring manipulating queries

```
@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 73. 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 basically produces 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 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(…)` executes 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 executing 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`.

5.3.11. Projections

Spring Data query methods usually return one or multiple instances of the aggregate root managed by the repository. However, it might sometimes be desirable to create projections based on certain attributes of those types. Spring Data allows modeling dedicated return types, to more selectively retrieve partial views of the managed aggregates.

Imagine a repository and aggregate root type such as the following example:

Example 78. A sample aggregate and repository

```
class Person {

  @Id UUID id;
  String firstname, lastname;
  Address address;

  static class Address {
    String zipCode, city, street;
```

```
  }
}

interface PersonRepository extends Repository<Person, UUID> {

  Collection<Person> findByLastname(String lastname);
}
```

Now imagine that we want to retrieve the person's name attributes only. What means does Spring Data offer to achieve this? The rest of this chapter answers that question.

Interface-based Projections

The easiest way to limit the result of the queries to only the name attributes is by declaring an interface that exposes accessor methods for the properties to be read, as shown in the following example:

Example 79. A projection interface to retrieve a subset of attributes

```
interface NamesOnly {

  String getFirstname();
  String getLastname();
}
```

The important bit here is that the properties defined here exactly match properties in the aggregate root. Doing so lets a query method be added as follows:

Example 80. A repository using an interface based projection with a query method

```
interface PersonRepository extends Repository<Person, UUID> {

  Collection<NamesOnly> findByLastname(String lastname);
}
```

The query execution engine creates proxy instances of that interface at runtime for each element returned and forwards calls to the exposed methods to the target object.

Projections can be used recursively. If you want to include some of the `Address` information as well, create a projection interface for that and return that interface from the declaration of `getAddress()`, as shown in the following example:

Example 81. A projection interface to retrieve a subset of attributes

```
interface PersonSummary {

  String getFirstname();
  String getLastname();
  AddressSummary getAddress();

  interface AddressSummary {
    String getCity();
  }
}
```

On method invocation, the `address` property of the target instance is obtained and wrapped into a projecting proxy in turn.

Closed Projections

A projection interface whose accessor methods all match properties of the target aggregate is considered to be a closed projection. The following example (which we used earlier in this chapter, too) is a closed projection:

Example 82. A closed projection

```
interface NamesOnly {

  String getFirstname();
  String getLastname();
}
```

If you use a closed projection, Spring Data can optimize the query execution, because we know about all the attributes that are needed to back the projection proxy. For more details on that, see the module-specific part of the reference documentation.

Class-based Projections (DTOs)

Another way of defining projections is by using value type DTOs (Data Transfer Objects) that hold properties for the fields that are supposed to be retrieved. These DTO types can be used in exactly the same way projection interfaces are used, except that no proxying happens and no nested projections can be applied.

If the store optimizes the query execution by limiting the fields to be loaded, the fields to be loaded are determined from the parameter names of the constructor that is exposed.

The following example shows a projecting DTO:

Example 87. A projecting DTO

```
class NamesOnly {

  private final String firstname, lastname;

  NamesOnly(String firstname, String lastname) {

    this.firstname = firstname;
    this.lastname = lastname;
  }

  String getFirstname() {
    return this.firstname;
  }

  String getLastname() {
    return this.lastname;
  }

  // equals(…) and hashCode() implementations
}
```

5.7. Transactionality

By default, CRUD methods on repository instances are transactional. For read operations, the transaction configuration `readOnly` flag is set to `true`. All others are configured with a plain `@Transactional` so that default transaction configuration applies. For details, see JavaDoc of `SimpleJpaRepository`. If you need to tweak transaction configuration for one of the methods declared in a repository, redeclare the method in your repository interface, as follows:

Example 106. Custom transaction configuration for CRUD

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

  @Override
  @Transactional(timeout = 10)
  public List<User> findAll();

  // Further query method declarations
}
```

Doing so causes the `findAll()` method to run with a timeout of 10 seconds and without the `readOnly` flag.

Another way to alter transactional behaviour is to use a facade or service implementation that (typically) covers more than one repository. Its purpose is to define transactional boundaries for non-CRUD operations. The following example shows how to use such a facade for more than one repository:

Example 107. Using a facade to define transactions for multiple repository calls

```
@Service
class UserManagementImpl implements UserManagement {

  private final UserRepository userRepository;
  private final RoleRepository roleRepository;

  @Autowired
  public UserManagementImpl(UserRepository userRepository,
```

```
    RoleRepository roleRepository) {
    this.userRepository = userRepository;
    this.roleRepository = roleRepository;
  }

  @Transactional
  public void addRoleToAllUsers(String roleName) {

    Role role = roleRepository.findByName(roleName);

    for (User user : userRepository.findAll()) {
      user.addRole(role);
      userRepository.save(user);
    }
}
```

This example causes call to `addRoleToAllUsers(…)` to run inside a transaction (participating in an existing one or creating a new one if none are already running). The transaction configuration at the repositories is then neglected, as the outer transaction configuration determines the actual one used. Note that you must activate `<tx:annotation-driven />` or use `@EnableTransactionManagement` explicitly to get annotation-based configuration of facades to work. This example assumes you use component scanning.

5.7.1. Transactional query methods

To let your query methods be transactional, use `@Transactional` at the repository interface you define, as shown in the following example:

Example 108. Using @Transactional at query methods

```
@Transactional(readOnly = true)
public interface UserRepository extends JpaRepository<User, Long> {

  List<User> findByLastname(String lastname);
```

```
  @Modifying
  @Transactional
  @Query("delete from User u where u.active = false")
  void deleteInactiveUsers();
}
```

Typically, you want the `readOnly` flag to be set to `true`, as most of the query methods only read data. In contrast to that, `deleteInactiveUsers()` makes use of the `@Modifying` annotation and overrides the transaction configuration. Thus, the method runs with the `readOnly` flag set to `false`.

5.8. Locking

To specify the lock mode to be used, you can use the `@Lock` annotation on query methods, as shown in the following example:

Example 109. Defining lock metadata on query methods

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

  // Plain query method
  @Lock(LockModeType.READ)
  List<User> findByLastname(String lastname);
}
```

This method declaration causes the query being triggered to be equipped with a `LockModeType` of `READ`. You can also define locking for CRUD methods by redeclaring them in your repository interface and adding the `@Lock` annotation, as shown in the following example:

Example 110. Defining lock metadata on CRUD methods

```
interface UserRepository extends Repository<User, Long> {
  // Redeclaration of a CRUD method
  @Lock(LockModeType.READ);
  List<User> findAll();
}
```