Spring Data Commons - Reference Documentation

1.0.0.M7

Preface

The Spring Data Commons project applies core Spring concepts to the development of solutions using many non-relational data stores.

Part I. Reference

This part of the reference documentation details the ...

Chapter 1. Repositories

1.1. Introduction

Implementing a data access layer of an application has been cumbersome for quite a while. Too much boilerplate code had to be written. Domain classes were anemic and haven't been designed in a real object oriented or domain driven manner.

Using both of these technologies makes developers life a lot easier regarding rich domain model's persistence. Nevertheless the amount of boilerplate code to implement repositories especially is still quite high. So the goal of the repository abstraction of Spring Data is to reduce the effort to implement data access layers for various persistence stores significantly

The following chapters will introduce the core concepts and interfaces of Spring Data repositories.

1.2. Core concepts

The central interface in Spring Data repository abstraction is `Repository` (probably not that much of a surprise). It is typeable to the domain class to manage as well as the id type of the domain class and provides some sophisticated functionality around CRUD for the entity managed.

Example 1.1. Repository interface

```
public interface Repository<T, ID extends Serializable> {

    T save(T entity);                                                    ❶

    T findById(ID primaryKey);                                           ❷

    List<T> findAll();                                                   ❸

    Page<T> findAll(Pageable pageable);                                  ❹

    Long count();                                                        ❺

    void delete(T entity);                                               ❻

    boolean exists(ID primaryKey);                                       ❼

    // … more functionality omitted.
}
```

❶ Saves the given entity.
❷ Returns the entity identified by the given id.
❸ Returns all entities.
❹ Returns a page of entities.
❺ Returns the number of entities.
❻ Deletes the given entity.
❼ Returns whether an entity with the given id exists.

Usually we will have persistence technology specific sub-interfaces to include additional technology specific methods. We will now ship implementations for a variety of Spring Data modules that implement that interface.

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

Example 1.2. PagingAndSortingRepository

```
public interface PagingAndSortingRepository<T, ID extends Serializable> extends Repository<T, ID> {

    List<T> findAll(Sort sort);

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

Accessing the second page of User by a page size of 20 you could simply do something like this:

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

1.3. Query methods

Next to standard CRUD functionality repositories are usually query the underlying datastore. With Spring Data declaring those queries becomes a four-step process (we use the JPA based module as example but that works the same way for other stores):

1. Declare an interface extending the technology specific Repository sub-interface and type it to the domain class it shall handle.

```
public interface PersonRepository extends JpaRepository<User, Long> { … }
```

2. Declare query methods on the interface.

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

3. Setup Spring to create proxy instances for those interfaces.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://www.springframework.org/schema/data/jpa
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/jpa
    http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

  <repositories base-package="com.acme.repositories" />

</beans>
```

4. Get the repository instance injected and use it.

```
public class SomeClient {

  @Autowired
  private PersonRepoyitory repository;

  public void doSomething() {
    List<Person> persons = repository.findByLastname("Matthews");
```

```
}
```

At this stage we barely scratched the surface of what's possible with the repositories but the general approach should be clear. Let's go through each of these steps and and figure out details and various options that you have at each stage.

1.3.1. Defining repository interfaces

As a very first step you define a domain class specific repository interface to start with. It's got to be typed to the domain class and an ID type so that you get CRUD methods of the `Repository` interface tailored to it.

1.3.2. Defining query methods

1.3.2.1. Query lookup strategies

The next thing we have to discuss is the definition of query methods. There's roughly two main ways how the repository proxy is generally able to come up with the store specific query from the method name. The first option is to derive the quer from the method name directly, the second is using some kind of additionally created query. What detailed options are available pretty much depends on the actual store. However there's got to be some algorithm the decision which actual query to is made.

There's three strategies for the repository infrastructure to resolve the query. The strategy to be used can be configured at the namespace through the `query-lookup-strategy` attribute. However might be the case that some of the strategies are not supported for the specific datastore. Here are your options:

CREATE

This strategy will try to construct a store specific query from the query method's name. The general approach is to remove a given set of well-known prefixes from the method name and parse the rest of the method. Read more about query construction in ???.

USE_DECLARED_QUERY

This strategy tries to find a declared query which will be used for execution first. The query could be defined by an annotation somwhere or declared by other means. Please consult the documentation of the specific store to find out what options are available for that store. If the repository infrastructure does not find a declared query for the method at bootstrap time it will fail.

CREATE_IF_NOT_FOUND (default)

This strategy is actually a combination of the both mentioned above. It will try to lookup a declared query first but create a custom method name based query if no declared query was found. This is default lookup strategy and thus will be used if you don't configure anything explicitly. It allows quick query definition by method names but also custom tuning of these queries by introducing declared queries for those who need explicit tuning.

1.3.2.2. Query creation

The query builder mechanism built into Spring Data repository infrastructue is useful to build constraining queries over entities of the repository. We will strip the prefixes `findBy`, `find`, `readBy`, `read`, `getBy` as well as

`get` from the method and start parsing the rest of it. At a very basic level you can define conditions on entity properties and concatenate them with `AND` and `OR`.

Example 1.3. Query creation from method names

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

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

The actual result of parsing that method will of course depend on the persistence store we create the query for. However there are some general things to notice. The expression are usually property traversals combined with operators that can be concatenated. As you can see in the example you can combine property expressions with And and Or. Beyond that you will get support for various operators like `Between`, `LessThan`, `GreaterThan`, `Like` for the property expressions. As the operators supported can vary from datastore to datastore please consult the according part of the reference documentation.

1.3.2.2.1. Property expressions

Property expressions can just refer to a direct property of the managed entity (as you just saw in the example above. On query creation time we already make sure that the parsed property is at a property of the managed domain class. However you can also traverse nested properties to define constraints on. Assume `Persons` have `Addresses` with `ZipCodes`. In that case a method name of

```
List<Person> findByAddressZipCode(ZipCode zipCode);
```

will create the property traversal `x.address.zipCode`. The resolution algorithm starts with interpreting the entire part (`AddressZipCode`) as property and checks the domain class for a property with that name (uncapitalized). If it succeeds it just uses that. If not it starts splitting up the source at the camel case parts from the right side into a head and a tail and tries to find the according property, e.g. `AddressZip` and `Code`. If we find a property with that head we take the tail and continue building the tree down from there. As in our case the first split does not match we move the split point to the left (`Address`, `ZipCode`).

Now although this should work for most cases, there might be cases where the algorithm could select the wrong property. Suppose our `Person` class has a `addressZip` property as well. Then our algorithm would match in the first split round already and essentially choose the wrong property and finally fail (as the type of `addressZip` probably has no code property). To resolve this ambiguity you can use _ inside your method name to manually define traversal points. So our method name would end up like so:

```
List<Person> findByAddress_ZipCode(ZipCode zipCode);
```

1.3.2.3. Special parameter handling

To hand parameters to your query you simply define method parameters as already seen in in examples above. Besides that we will recognizes certain specific types to apply pagination and sorting to your queries dynamically.

Example 1.4. Using Pageable and Sort in query methods

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

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

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

The first method allows you to pass a `Pageable` instance to the query method to dynamically add paging to your statically defined query. `Sorting` options are handed via the `Pageable` instance, too. If you only need sorting, simply add a `Sort` parameter to your method. As you also can see, simply returning a `List` is possible as well. We will then not retrieve the additional metadata required to build the actual `Page` instance but rather simply restrict the query to lookup only the given range of entities.

> **Note**
>
> To find out how many pages you get for a query entirely we have to trigger an additional count query. This will be derived from the query you actually trigger by default.

1.3.3. Creating repository instances

So now the question is how to create instances and bean definitions for the repository interfaces defined.

1.3.3.1. Spring

The easiest way to do so is by using the Spring namespace that is shipped with each Spring Data module that supports the repository mechanism. Each of those includes a repositories element that allows you to simply define a base packge Spring shall scan for you.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://www.springframework.org/schema/data/jpa
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/jpa
    http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

  <repositories base-package="com.acme.repositories" />

</beans:beans>
```

In this case we instruct Spring to scan com.acme.repositories and all it's sub packages for interfaces extending the appropriate `Repository` sub-interface (in this case `JpaRepository`). For each interface found it will register the presistence technology specific `FactoryBean` to create the according proxies that handle invocations of the query methods. Each of these beans will be registered under a bean name that is derived from the interface name, so an interface of `UserRepository` would be registered under `userRepository`. The `base-package` attribute allows to use wildcards, so that you can have a pattern of packages parsed.

Using filters

By default we will pick up every interface extending the persistence technology specific `Repository` sub-interface located underneath the configured base package and create a bean instance for it. However, you might want to gain finer grained control over which interfaces bean instances get created for. To do this we support the use of `<include-filter />` and `<exclude-filter />` elements inside `<repositories />`. The semantics are exactly equivalent to the elements in Spring's context namespace. For details see Spring reference documentation on these elements.

E.g. to exclude certain interfaces from instantiation as repository, you could use the following configuration:

Example 1.5. Using exclude-filter element

```
<repositories base-package="com.acme.repositories">
  <context:exclude-filter type="regex" expression=".*SomeRepository" />
</repositories>
```

This would exclude all interface ending on `SomeRepository` from being instantiated.

Manual configuration

If you'd rather like to manually define which repository instances to create you can do this with nested `<repository />` elements.

```
<repositories base-package="com.acme.repositories">
  <repository id="userRepository" />
</repositories>
```

1.3.3.2. Standalone usage

You can also use the repository infrastructure outside of a Spring container usage. You will still need to have some of the Spring libraries on your classpath but you can generally setup repositories programatically as well. The Spring Data modules providing repository support ship a persistence technology specific RepositoryFactory that can be used as follows:

Example 1.6. Standalone usage of repository factory

```
RepositoryFactorySupport factory = … // Instantiate factory here
UserRepository repository = factory.getRepository(UserRepository.class);
```

1.4. Custom implementations

1.4.1. Adding behaviour to single repositories

Often it is necessary to provide a custom implementation for a few repository methods. Spring Data repositories easily allow provide custom repository code and integrate it with generic CRUD abstraction and query method functionality. To enrich a repository with custom functionality you have to define an interface and an implementation for that functionality first and let the repository interface you provided so far extend that custom interface.

Example 1.7. Interface for custom repository functionality

```
interface UserRepositoryCustom {

  public void someCustomMethod(User user);
}
```

Example 1.8. Implementation of custom repository functionality

```
class UserRepositoryImpl implements UserRepositoryCustom {

  public void someCustomMethod(User user) {
    // Your custom implementation
  }
}
```

Note that the implementation itself does not depend on Spring Data and can be a regular Spring bean. So you can either use standard dependency injection behaviour to inject references to other beans, take part in aspects and so on.

Example 1.9. Changes to the your basic repository interface

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

  // Declare query methods here
}
```

Let your standard repository interface extend the custom one. This makes CRUD and custom functionality available to clients.

Configuration

If you use namespace configuration the repository infrastructure tries to autodetect custom implementations by looking up classes in the package we found a repository using the naming conventions appending the namespace element's attribute `repository-impl-postfix` to the classname. This suffix defaults to `Impl`.

Example 1.10. Configuration example

```
<repositories base-package="com.acme.repository">
  <repository id="userRepository" />
</repositories>

<repositories base-package="com.acme.repository" repository-impl-postfix="FooBar">
  <repository id="userRepository" />
</repositories>
```

The first configuration example will try to lookup a class `com.acme.repository.UserRepositoryImpl` to act as custom repository implementation, where the second example will try to lookup `com.acme.repository.UserRepositoryFooBar`.

Manual wiring

The approach above works perfectly well if your custom implementation uses annotation based configuration and autowring entirely as will be trated as any other Spring bean. If your customly implemented bean needs some special wiring you simply declare the bean and name it after the conventions just descibed. We will then pick up the custom bean by name rather than creating an own instance.

Example 1.11. Manual wiring of custom implementations (I)

```
<repositories base-package="com.acme.repository">
  <repository id="userRepository" />
</repositories>

<beans:bean id="userRepositoryImpl" class="…">
  <!-- further configuration -->
</beans:bean>
```

This also works if you use automatic repository lookup without defining single `<repository />` elements.

In case you are not in control of the implementation bean name (e.g. if you wrap a generic repository facade around an existing repository implementation) you can explicitly tell the `<repository />` element which bean to use as custom implementation by using the `repository-impl-ref` attribute.

Example 1.12. Manual wiring of custom implementations (II)

```
<repositories base-package="com.acme.repository">
  <repository id="userRepository" repository-impl-ref="customRepositoryImplementation" />
</repositories>

<bean id="customRepositoryImplementation" class="…">
  <!-- further configuration -->
</bean>
```

1.4.2. Adding custom behaviour to all repositories

In other cases you might want to add a single method to all of your repository interfaces. So the approach just shown is not feasible. The first step to achieve this is adding and intermediate interface to declare the shared behaviour

Example 1.13. An interface declaring custom shared behaviour

```
public interface MyRepository<T, ID extends Serializable>
  extends JpaRepository<T, ID> {

  void sharedCustomMethod(ID id);
}
```

Now your individual repository interfaces will extend this intermediate interface to include the functionality declared. The second step is to create an implementation of this interface that extends the persistence technology specific repository base class which will act as custom base class for the repository proxies then.

Note

If you're using automatic repository interface detection using the Spring namespace using the interface just as is will cause Spring trying to create an instance of `MyRepository`. This is of course not desired as it just acts as indermediate between `Repository` and the actual repository interfaces

you want to define for each entity. To exclude an interface extending `Repository` from being instantiated as repository instance annotate it with `@NoRepositoryBean`.

Example 1.14. Custom repository base class

```
public class MyRepositoryImpl<T, ID extends Serializable>
  extends SimpleJpaRepository<T, ID> implements MyRepository<T, ID> {

  public void sharedCustomMethod(ID id) {
    // implementation goes here
  }
}
```

The last step to get this implementation used as base class for Spring Data repositores is replacing the standard `RepositoryFactoryBean` with a custom one using a custom `RepositoryFactory` that in turn creates instances of your `MyRepositoryImpl` class.

Example 1.15. Custom repository factory bean

```
public class MyRepositoryFactoryBean<T extends JpaRepository<?, ?>
  extends JpaRepositoryFactoryBean<T> {

  protected RepositoryFactorySupport getRepositoryFactory(…) {
    return new MyRepositoryFactory(…);
  }

  private static class MyRepositoryFactory extends JpaRepositoryFactory{

    public MyRepositoryImpl getTargetRepository(…) {
      return new MyRepositoryImpl(…);
    }

    public Class<? extends RepositorySupport> getRepositoryClass() {
      return MyRepositoryImpl.class;
    }
  }
}
```

Finally you can either declare beans of the custom factory directly or use the `factory-class` attribute of the Spring namespace to tell the repository infrastructure to use your custom factory implementation.

Example 1.16. Using the custom factory with the namespace

```
<repositories base-package="com.acme.repository"
  factory-class="com.acme.MyRepositoryFactoryBean" />
```