Spring Data JPA - Reference Documentation

1.0.0.M2

Preface iii
1. Project metadata iii
I. Reference Documentation 1
1. Repositories 2
1.1. Introduction 2
1.2. Core concepts 2
1.3. Query methods 3
1.3.1. Defining repository interfaces 4
1.3.2. Defining query methods 4
1.3.3. Creating repository instances 6
1.4. Custom implementations 7
1.4.1. Adding behaviour to single repositories 7
1.4.2. Adding custom behaviour to all repositories 9
2. JPA Repositories 11
2.1. Query methods 11
2.1.1. Query lookup strategies 11
2.1.2. Query creation 11
2.1.3. Using JPA NamedQueries 12
2.1.4. Using @Query 13
2.1.5. Using named parameters 14
2.1.6. Modifying queries 14
2.2. Specifications 14
2.3. Transactionality 16
2.3.1. Transactional query methods 17
2.4. Auditing 17
II. Appendix 20
A. Namespace reference 21
A.1. The `<repositories />` element 21
A.2. The `<repository />` element 21
B. Frequently asked questions 22
Glossary 23

Preface

1. Project metadata

- Version control - git://github.com/SpringSource/spring-data-jpa.git
- Bugtracker - https://jira.springsource.org/browse/DATAJPA
- Release repository - http://maven.springframework.org/release
- Milestone repsitory - http://maven.springframework.org/milestone
- Snapshot repsitory - http://maven.springframework.org/snapshot

Part I. Reference Documentation

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" />
```

Chapter 2. JPA Repositories

This chapter includes details of the JPA repository implementation.

2.1. Query methods

2.1.1. 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 Section 2.1.3, “Using JPA NamedQueries” for more information) or rather annotate your query method with `@Query` (see as for details).

Strategies

This strategy tries to find a declared query that can either be defined using JPA `@NamedQuery` means or Hades `@Query` annotation (see Section 2.1.3, “Using JPA NamedQueries” and Section 2.1.4, “Using @Query” for details). If no declared query is found execution of the query 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 named 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.

2.1.2. Query creation

Generally the query creation mechanism for JPA works as described in Section 1.3, “Query methods”. Here's a short example of what a JPA query method translates into:

Example 2.1. 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 like described in Section 1.3.2.2.1,

“Property expressions”. Here's an overview of the keywords supported for JPA and what a method containing that keyword essentially translates to.

Table 2.1. 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`
`Between`	`findByStartDateBetween`	`… where x.startDate between 1? and ?2`
`LessThan`	`findByAgeLessThan`	`… where x.age < ?1`
`GreaterThan`	`findByAgeGreaterThan`	`… where x.age > ?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`
`OrderBy`	`findByAgeOrderByLastnameDesc`	`… where x.age = ?1 order by x.lastname desc`
`Not`	`findByLastnameNot`	`… where x.lastname <> ?1`

2.1.3. 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 2.2. 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 not to need another config file to be edited, probably lowering maintenance cost. You pay for that benefit by the need to recompile your domain class for every new query declaration.

Example 2.3. 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 this named query all you need to do is to specify the `UserRepository` as follows:

Example 2.4. Query method declaration in UserRepository

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

  List<User> findByLastname(String lastname);

  User findByEmailAddress(String emailAddress);
}
```

Declaring this method we will try to resolve a call to this method 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.

2.1.4. Using @Query

Using named queries to declare queries for entities is a valid approach and works fine for a small number amount of queries. As the queries themselves are tied to a Java method to execute them you actually can bind them to the query executing methods using Spring Data JPA `@Query` annotation rather than annotating them to the domain class. This will free the domain class from persistence specific information and colocate the query to the repository interface.

Querys annotated to the query method will trump queries defined using `@NamedQuery` or named queries declared in in `orm.xml`.

Example 2.5. 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);
}
```

2.1.5. Using named parameters

By default Sprign Data JPA will use position based parameter binding as described in all the samples above. This makes query methods a little error prone to refactorings 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 2.6. 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.

2.1.6. Modifying queries

All the sections before described how to declare queries to access a given entity or collection of entitites. Of course you can add custom modifying behaviour by using facilities described in ???. 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 2.7. 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 automatically clear it (see JavaDoc of `EntityManager.clear()` for details). This will effectively drop all non-flushed changes still pending in the `EntityManager`. If you don't wish the `EntityManager` to be cleared automatically you can set `@Modifying` annotation's `clearAutomatically` attribute to `false`;

2.2. Specifications

JPA 2 introduces a criteria API that can be used to build queries programatically. Writing a criteria you actually define the where-clause of a query for a query of the handled domain class. Taking another step back these criterias can be regarded as predicate over the entity that is verbalized by the JPA criteria API constraints.

Spring Data JPA now takes the concept of a specification from Eric Evans' book Domain Driven Design, that carries the same semantics and provides an API to define such `Specifications` using the JPA criteria API. Thus you find methods like this in `JpaRepository`:

```
List<T> readAll(Specification<T> spec);
```

The `Specification` interface now looks as follows:

```
public interface Specification<T> {
  Predicate toPredicate(Root<T> root, CriteriaQuery<?> query,
            CriteriaBuilder builder);
}
```

Okay, so what is the typical use case? `Specifications` can easily be used to build an extensible set of predicates on top of an entity that then can be combined and used with `JpaRepository` without the need of declaring a query (method) for every needed combination of those. Here's an example:

Example 2.8. Specifications for a Customer

```
public class CustomerSpecs {

  public static Specification<Customer> isLongTermCustomer() {
    return new Specification<Customer>() {
      Predicate toPredicate(Root<T> root, CriteriaQuery<?> query,
            CriteriaBuilder builder) {

         LocalDate date = new LocalDate().minusYears(2);
         return builder.lessThan(root.get(Customer_.createdAt), date);
      }
    };
  }


  public static Specification<Customer> hasSalesOfMoreThan(MontaryAmount value) {
    return new Specification<Customer>() {
      Predicate toPredicate(Root<T> root, CriteriaQuery<?> query,
            CriteriaBuilder builder) {

         // build query here
      }
    };
  }
}
```

Admittedly the amount of boilerplate leaves room for improvement (that will hopefully be reduced by Java 8 closures) but the client side becomes much nicer as you will see below. Besides that we have expressed some criteria on a business requirement abstraction level and created executable `Specifications`. So a client might use a `Specification` as follows:

Example 2.9. Using a simple Specification

```
List<Customer> customers = customerRepository.findAll(isLongTermCustomer());
```

Okay, why not simply creating a query for this kind of data access? You're right. Using a single `Specification` does not gain a lot of benefit over a plain query declaration. The power of `Specifications` really shines when you combine them to create new `Specification` objects. You can achieve this through the `Specifications` helper class we provide to build expressions like this:

Example 2.10. Combined Specifications

```
MonetaryAmount amount = new MonetaryAmount(200.0, Currencies.DOLLAR);
List<Customer> customers = customerRepository.readAll(
```

```
  where(isLongTermCustomer()).or(hasSalesOfMoreThan(amount)));
```

As you can see, `Specifications` offers some gluecode methods to chain and combine `Specifications`. Thus extending your data access layer is just a matter of creating new `Specification` implementations and combining them with ones already existing.

2.3. Transactionality

CRUD methods on repository instances are transactional by default. For reading 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 `Repository`. If you need to tweak transaction configuration for one of the methods declared in `Repository` simply redeclare the method in your repository interface as follows:

Example 2.11. Custom transaction configuration for CRUD

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

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

  // Further query method declarations
}
```

This will cause the `findAll()` method to be executed with a timeout of 10 seconds and without the `readOnly` flag.

Another possibility to alter transactional behaviour is using a facade or service implementation that typically covers more than one repository. Its purpose is to define transactional boundaries for non-CRUD operations:

Example 2.12. 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.readAll()) {
      user.addRole(role);
      userRepository.save(user);
    }
}
```

This will cause call to `addRoleToAllUsers(…)` to run inside a transaction (participating in an existing one or create a new one if none already running). The transaction configuration at the repositories will be neglected then as the outer transaction configuration determines the actual one used. Note that you will have to activate `<tx:annotation-driven />` explicitly to get annotation based configuration at facades working. The example above assumes you're using component scanning.

2.3.1. Transactional query methods

To let your query methods be transactional simply use `@Transactional` at the repository interface you define.

Example 2.13. 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 will use the `readOnly` flag set to true as most of the query methods will be reading ones. In contrast to that `deleteInactiveUsers()` makes use of the `@Modifying` annotation and overrides the transaction configuration. Thus the method will be executed with `readOnly` flag set to false.

> **Note**
>
> It's definitely reasonable to use transactions for read only queries as we can mark them as such by setting the `readOnly` flag. This will not act as check that you do not trigger a manipulating query nevertheless (although some databases reject e.g. `INSERT` or `UPDATE` statements inside a transaction set to be read only) but gets propagated as hint to the underlying JDBC driver to do performance optimizations. Furthermore Spring will do some optimizations to the underlying JPA provider. E.g. when used with Hibernate the flush mode is set to `NEVER` when you configure a transaction as read only which causes Hibernate to skip dirty checks that gets quite noticeable on large object trees.

2.4. Auditing

Most applications will require some auditability for entities allowing to track creation date and user and modification date and user. Spring Data JPA provides facilities to add this audition information to entity transparently by AOP means. To take part in this functionality your domain classes have to implement a more advanced interface:

Example 2.14. `Auditable` interface

```
public interface Auditable<U, ID extends Serializable>
        extends Persistable<ID> {

    U getCreatedBy();
```

```
    void setCreatedBy(U createdBy);

    DateTime getCreatedDate();

    void setCreated(Date creationDate);

    U getLastModifiedBy();

    void setLastModifiedBy(U lastModifiedBy);

    DateTime getLastModifiedDate();

    void setLastModified(Date lastModifiedDate);
}
```

As you can see the modifying entity itself only has to be an entity. Mostly this will be some sort of User entity, so we chose U as parameter type.

Note

To minimize boilerplate code Spring Data JPA offers `AbstractPersistable` and `AbstractAuditable` base classes that implement and preconfigure entities. Thus you can decide to only implement the interface or enjoy more sophisticated support by extending the base class.

General auditing configuration

Spring Data JPA ships with an entity listener that can be used to trigger capturing auditing information. So first you have to register the `AuditingEntityListener` inside your `orm.xml` to be used for all entities in your persistence contexts:

Example 2.15. Auditing configuration orm.xml

```
<persistence-unit-metadata>
  <persistence-unit-defaults>
    <entity-listeners>
      <entity-listener class="….data.jpa.domain.support.AuditingEntityListener" />
    </entity-listeners>
  </persistence-unit-defaults>
</persistence-unit-metadata>
```

Now activating auditing functionlity is just a matter of adding the Spring Data JPA `auditing` namespace element to your configuration:

Example 2.16. Activating auditing in the Spring configuration

```
<jpa:auditing auditor-aware-ref="yourAuditorAwareBean" />
```

As you can see you have to provide a bean that implements the `AuditorAware` interface which looks as follows:

Example 2.17. `AuditorAware` interface

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

```
    T getCurrentAuditor();
}
```

Usually you will have some kind of authentication component in your application that tracks the user currently working with the system. This component should be `AuditorAware` and thus allow seemless tracking of the auditor.

Part II. Appendix

Appendix A. Namespace reference

A.1. The `<repositories />` element

The `<repositories />` element acts as container for `<repository />` elements or can be left empty to trigger auto detection[1] of repository instances. Attributes defined for `<repositories />` act are propagated to contained `<repository />` elements but can be overridden of course.

Table A.1. Attributes

Name	Description
`base-package`	Defines the package to be used to be scanned for repository interfaces extending `*Repository` (actual interface is determined by specific Spring Data module) in auto detection mode. All packages below the configured package will be scanned, too. In auto configuration mode (no nested `<repository />` elements) wildcards are also allowed.
`repository-impl-postfix`	Defines the postfix to autodetect custom repository implementations. Classes whose names end with the configured postfix will be considered as candidates. Defaults to `Impl`.
`query-lookup-strategy`	Determines the strategy to be used to create finder queries. See ??? for details. Defaults to `create-if-not-found`.

A.2. The `<repository />` element

The `<repository />` element can contain all attributes of `<repositories />` except `base-package`. This will result in overriding the values configured in the surrounding `<repositories />` element. Thus here we will only document extended attributes.

Table A.2. Attributes

`id`	Defines the id of the bean the repository instance will be registered under as well as the repository interface name.
`custom-impl-ref`	Defines a reference to a custom repository implementation bean.

[1] see ???

Appendix B. Frequently asked questions

B.1. Common

B.1.1.

I'd like to get more detailled logging information on what methods are called inside `JpaRepository`, e.g. How can I gain them?

You can make use of `CustomizableTraceInterceptor` provided by Spring:

```
<bean id="customizableTraceInterceptor" class="
  org.springframework.aop.interceptor.CustomizableTraceInterceptor">
  <property name="enterMessage" value="Entering $[methodName]($[arguments])"/>
  <property name="exitMessage" value="Leaving $[methodName](): $[returnValue]"/>
</bean>

<aop:config>
  <aop:advisor advice-ref="customizableTraceInterceptor"
    pointcut="execution(public * org.sfw.data.jpa.repository.JpaRepository+.*(..))"/>
</aop:config>
```

B.2. Infrastructure

B.2.1.

Currently I have implemented a repository layer based on `HibernateDaoSupport`. I create a `SessionFactory` by using Spring's `AnnotationSessionFactoryBean`. How do I get Hades DAOs working in this environment.

You have to replace `AnnotationSessionFactoryBean` with the `LocalContainerEntityManagerFactoryBean`. Supposed you have registered it under `entityManagerFactory` you can reference it in you repositories based on `HibernateDaoSupport` as follows:

Example B.1. Looking up a SessionFactory from an HibernateEntityManagerFactory

```
<bean class="com.acme.YourDaoBasedOnHibernateDaoSupport">
  <property name="sessionFactory">
    <bean factory-bean="entityManagerFactory"
      factory-method="getSessionFactory" />
  </property>
</bean>
```

B.3. Auditing

B.3.1.

I want to use Spring Data JPA auditing capabilities but have my database already set up to set modification and creation date on entities. How to prevent Hades to set the date programmatically.

Just use the `set-dates` attribute of the `auditing` namespace element to false.

Glossary

A

AOP	Aspect oriented programming

C

Commons DBCP	Commons DataBase Connection Pools - Library of the Apache foundation offering pooling implementations of the `DataSource` interface.
CRUD	Create, Read, Update, Delete - Basic persistence operations

D

DAO	Data Access Object - Pattern to separate persisting logic from the object to be persisted
Dependency Injection	Pattern to hand a component's dependency to the component from outside, freeing the component to lookup the dependant itself. For more information see http://en.wikipedia.org/wiki/Dependency_Injection.

E

EclipseLink	Object relational mapper implementing JPA - http://www.eclipselink.org

H

Hibernate	Object relational mapper implementing JPA - http://www.hibernate.org

J

JPA	Java Persistence Api

S

Spring	Java application framework - http://www.springframework.org