

Generic Query (Hibernate)

Getting Started with gquery (Hibernate)

This is the getting started guide for the Generic Query (gquery) *Hibernate* component of the grepo framework. It's not supposed to be a complete reference manual - the goal is to show a basic usage and configuration scenario of grepo's gquery component (using hibernate). If you have problems understanding parts of this guide or the framework in general or if you have any suggestions, good ideas or if you have found potential bugs please let us know. So let's get started!

Version Information

This is the getting started guide for the Generic Query (gquery) *Hibernate* component of the grepo framework version 1.5.x.

Older versions of this guide:

- [Version 1.0](#)

- [Download the demo project](#)
 - [Let grepo tell you what's going on](#)
- [Demo application](#)
- [Using the grepo framework](#)
- [Creating the first repository](#)
 - [Using repository scanning](#)
- [Executing generic queries](#)
 - [Queries with named-parameters](#)
 - [How it works](#)
 - [Separating queries from Java code](#)
 - [How it works](#)
 - [Using dynamically generated queries](#)
- [Additional functionality](#)
 - [Paging](#)
 - [Result conversion](#)
 - [Implicit result conversion](#)
 - [Result validation](#)
 - [Tracking statistics](#)
- [Transaction Handling](#)
- [A word about conventions](#)

Download the demo project

The demo project for this guide can be checked out from our SVN repository as follows:

```
$ svn checkout
http://svn.codehaus.org/grepo/tags/demo-grep
o-query-hibernate-1.5.0
demo-grepo-query-hibernate
```

The demo project is a maven project and we highly recommend that you use maven to set up the project. If you don't want to use maven you can also set up the project manually. If you use maven and eclipse you can easily make an eclipse project using the following command in the demo project's root directory:

```
$ mvn eclipse:eclipse
```

You can now import the project in your eclipse workspace.

After you have imported the project you should now be able to run the *UserRepositoryTest* JUnit test. You can also run the test using maven from command line:

```
$ mvn test
```

Let grepo tell you what's going on

Grepo uses the [slf4j](#) logging api. If you set the logger level for the package *org.codehaus.grepo* to *DEBUG* in (*src/test/resources/log4j.xml*) like this:

```
<category name="org.codehaus.grepo">
  <priority value="DEBUG" />
</category>
```

Grepo should print out information similar to this:

```
15:11:51,211 DEBUG
[GenericQueryMethodInterceptor:56] -
Invoking method 'loadByUsername'
```

```
15:11:51,218 DEBUG
[QueryExecutorNamingStrategyImpl:62] -
Resolved executor name: load
15:11:51,225 DEBUG
[QueryExecutorFindingStrategyImpl:55] -
Found queryExecutor

'org.codehaus.grepo.query.hibernate.executor
.LoadQueryExecutor' for execution of method
    'loadByUsername'
15:11:51,241 DEBUG
[GenericRepositorySupport:105] - Executing
query without using transaction template
15:11:51,260 DEBUG
[QueryNamingStrategyImpl:86] - Resolved
named-query: demo.domain.User.ByUsername
15:11:51,265 DEBUG
[AbstractHibernateQueryExecutor:168] - Using
query: from User where username = ?
15:11:51,267 DEBUG
[AbstractHibernateQueryExecutor:456] -
Setting parameter '0' to 'max'
15:11:51,312 DEBUG
[DefaultHibernateRepository:155] - Query
result is

'demo.domain.User@24a09e41[id=0,username=max
]'
```

```
15:11:51,333 DEBUG
```

```
[GenericQueryMethodInterceptor:72] -
```

```
Invocation of method 'loadByUsername'  
took '0:00:00.010'
```

Demo application

The teeny-weeny demo application consists of one database table (*USERS*) which is mapped using Hibernate to one Java entity (*demo.domain.User*). Note that the object relational mapping is done using standard Hibernate mechanisms and is not grepo specific. The mapping can be found in *src/main/resources/META-INF/hibernate/User.hbm.xml*. The *USERS* table contains the following columns:

COLUMN	PROPERTIES
ID	not null, primary key
FIRSTNAME	nullable
LASTNAME	nullable
USERNAME	not null, unique
EMAIL	nullable, unique

Using the grepo framework

In order to use the gquery component with Hibernate you need the following grepo artifacts (jars) in your project's classpath:

- grepo-core-<VERSION>.jar
- grepo-query-commons-<VERSION>.jar
- grepo-query-hibernate-<VERSION>.jar

Somewhere in your Spring application context (xml) you have to import the default Hibernate configuration of the grepo gquery component.

```
<import  
resource="classpath:META-INF/grepo/grepo-que  
ry-hibernate-default.cfg.xml" />
```

In the demo project this is done in *src/main/resources/META-INF/spring/application-context.xml*. Note that you may not need to import that file if you decide to setup grepo with special/custom configuration - you could for instance configure the required "grepo" beans directly in your application context.

Furthermore you need to define a datasource and a Hibernate session factory. This configuration is "standard" Spring/Hibernate and is not grepo specific. In the demo project the hsql datasource and session factory are

configured in `src/main/resources/META-INF/spring/db-environment.xml`.

The next step is to define a basic configuration for the data access layer. This configuration may look like this:

```
<bean id="repositoryFactory"

class="org.codehaus.grepo.query.hibernate.re
pository.HibernateRepositoryFactoryBean"
abstract="true" />
```

Using the grepo namespace the configuration above looks like:

```
<gquery:repository-factory
id="repositoryFactory" />
```

Here we define an *abstract* factory bean which will be used to create all instances of our concrete repositories (DAOs). Its a good idea (although not required) to use an *abstract* factory bean because this makes configuration of concrete repository beans simpler and furthermore you just have to modify the *abstract* factory bean if you need to change the basic configuration for your data access layer in the future. Note that the *abstract* factory bean above uses a really simple configuration approach (the rest of the configuration will be done by the framework behind the scenes). For custom/special needs the *abstract* bean definition (properties) may be more comprehensive. The following is a simple example of an *abstract* bean which sets the *sessionFactory* property accordingly:

```
<bean id="repositoryFactory"

class="org.codehaus.grepo.query.hibernate.re
pository.HibernateRepositoryFactoryBean"
abstract="true">
  <property name="sessionFactory"
ref="sessionFactory" />
</bean>
```

Using the grepo namespace the configuration above looks like:

```
<gquery:repository-factory
id="repositoryFactory">
  <property name="sessionFactory"
ref="sessionFactory" />
</gquery:repository-factory>
```

That's it, you are now ready to build your data access layer using the grepo framework!

Creating the first repository

Now its time to create the generic repository for the *User* entity. For this we just have to define an empty interface (*demo.repository.UserRepository*), which looks like:

```
package demo.repository;

import
org.codehaus.grepo.query.hibernate.repository
y.ReadWriteHibernateRepository;
import demo.domain.User;
import
org.springframework.stereotype.Repository;

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

Note that we want a read-write repository (this means that we want a repository with basic CRUD operations) and thus have to extend from *org.codehaus.grepo.query.hibernate.repository.ReadWriteHibernateRepository*. For a read-only repository we would extend from *org.codehaus.grepo.query.hibernate.repository.ReadOnlyHibernateRepository* instead. The first generic type is the Java entity (*User*) and the second one is the type for the primary key property of the entity (*Long*).

The next step is to configure a repository bean in our Spring application context:

```
<bean id="userRepository"
parent="repositoryFactory">
  <property name="proxyClass"
value="demo.repository.UserRepository" />
</bean>
```

Using the grepo namespace the configuration above looks like:

```
<gquery:repository id="userRepository"
factory="repositoryFactory"

proxy-class="demo.repository.UserRepository"
/>
```

The *userRepository* bean uses the *repositoryFactory* bean as parent and thus inherits the basic configuration from the parent. We have to set (at least) one property:

- *proxyClass*: This property must be set to the interface which should be "proxied". In our case we provide the fully qualified name of the *demo.repository.UserRepository* interfaces which we have created previously.

Using repository scanning

You can also use grepo's repository-scan feature (instead of configuring your repository beans manually) in order to automatically detect concrete repository beans. This functionality is similar (based on) Spring's component-scan feature.

Using the grepo namespace the configuration looks like:

```
<gquery:repository-scan
base-package="demo.repository"
factory="repositoryFactory" />
```

Note: That the repository interfaces (by default) have to be annotated with Spring's *Repository* annotation in order to be detected by the repository-scanner.

Finished! We can now inject our *userRepository* bean wherever we need basic CRUD operations for the *User* entity.

Executing generic queries

We can now add various methods to our *UserRepository* interface in order to execute queries (either HQL or SQL). In our application we might want to load a user from database via *username* property. Therefore we would add the following method to our *UserRepository* interface:

```
@GenericQuery(query = "from User where  
username = ?")  
User getByUsername(String username);
```

And that's it! We can now inject our *userRepository* bean and use the *getByUsername* method to fetch *User* entities from database by *username*. As you can see we use grepo's *GenericQuery* annotation to tell the framework that the method has to be executed (or handled) dynamically (remember that we didn't have to provide an implementation of our *UserRepository* interface). The *query* property provides the HQL query to be executed when the method gets invoked. We can also execute SQL queries:

```
@GenericQuery(query = "select * from USERS  
where username = ?", isNativeQuery = true)  
@HibernateQueryOptions(entityClasses =  
@EntityClass(clazz = User.class))  
User getByUsername(String username);
```

Here we have to use grepo's *HibernateQueryOptions* annotation to tell the framework that an instance of *User* has to be returned.

Queries with named-parameters

We could also use Hibernate's named-parameter style for placeholders in our query (instead of the JDBC style which uses question marks as placeholders):

```
@GenericQuery(query = "from User where  
username = :un")  
User getByUsername(@Param("un") String  
username);
```

Note that we also have to annotate the method's parameters with grepo's *Param* annotation. This is necessary to associate method parameters with named-parameters in the query. Even though using named-parameters requires

a little more configuration in your interface (because of the *Param* annotation), there are several reasons why someone would (and should) prefer named-parameter in queries:

- The sequence and number of query-parameters does not have to match the method's parameters. (Using question marks as placeholders generally requires sequence and number of query-parameters to match the method's parameters).
- You can use one method parameter for multiple query parameters.
- Using named-parameter queries you can provide a *java.util.Collection* as method parameter and use it for *IN-Clauses* in your query. For instance:

```
@GenericQuery(query = "from User where  
username IN (:list)")  
List<User> findByUsernames(@Param("list")  
Collection<String> usernames);
```

How it works

The attentive reader may wonder how grepo can return the appropriate types. The *getByUsername* method returns an instance of *User* while *findByUsernames* returns a list of *User* entities. The answer is conventions. Grepo is configured with a set of *org.codehaus.grepo.query.commons.executor.QueryExecutors*. As the name implies a *QueryExecutor* is responsible for executing queries. In the default configuration grepo uses the following implementations of the *QueryExecutor* interface:

- *org.codehaus.grepo.query.hibernate.executor.GetQueryExecutor*: This implementation executes a query using the *uniqueResult* method of the Hibernate API and thus returns *null* or the instance. If multiple records were found, Hibernate throws an unchecked exception.
- *org.codehaus.grepo.query.hibernate.executor.LoadQueryExecutor*: Same as *GetQueryExecutor* with the difference that this executor never returns *null* - if no records were found then an *NoResultException* will be thrown.
- *org.codehaus.grepo.query.hibernate.executor.ListQueryExecutor*: This implementation executes a query using the *list* method of the Hibernate API and thus returns instances of *java.util.List*. This executor never returns *null* - if no records were found an empty list is returned.
- *org.codehaus.grepo.query.hibernate.executor.UpdateQueryExecutor*: This implementation executes *update-* or *delete-queries*, that is queries that *update* or *delete* records in database. It uses the *executeUpdate* method of the Hibernate API and thus returns an *int* indicating how many records were affected by the (update- or delete-) operation.
- *org.codehaus.grepo.query.hibernate.executor.IterateQueryExecutor*: This implementation executes a query using the *iterate* method of the Hibernate API and thus returns an instance of *java.util.Iterator*.
- *org.codehaus.grepo.query.hibernate.executor.ScrollQueryExecutor*: This implementation executes a query using the *scroll* method of the Hibernate API and thus returns an instance of *org.hibernate.ScrollableResults*.

So obviously grepo used the *ListQueryExecutor* for the *findByUsernames* and the *GetQueryExecutor* for the *getByUsername* method. But how did grepo know which executor has to be used for which method (Note that we didn't tell grepo which executor has to be used). The answer is again conventions. The framework uses a *org.codehaus.grepo.query.commons.executor.ExecutorFindingStrategy* which is responsible for finding the appropriate executor for a given generic method. In its default configuration grepo uses *org.codehaus.grepo.query.commons.executor.QueryExecutorFindingStrategyImpl*. This implementation uses a *org.codehaus.grepo.query.commons.executor.QueryExecutorNamingStrategy* which is responsible to retrieve an executor name for a given method. Furthermore the *QueryExecutorFindingStrategyImpl* has a registry (basically a *Map*) which maps executor names to executor classes. The or

`g.codehaus.grepo.query.commons.executor.QueryExecutorNamingStrategyImpl` (which is the one grepo uses in its basic configuration) resolves executor names according to the following rules:

If the name of the method matches the pattern

```
^(is|has|find|get|load|scroll|iterate|delete|update)
```

then the executor name is the matching prefix. This means that if the method starts with *is*, *has*, *get*, *load*, *scroll*, *iterate*, *delete* or *update* this rule will be applied.

If the method name does not match the pattern above, then *null* will be returned, meaning that the strategy didn't find the appropriate executor name for this method.

Feel free to provide your own implementation if desired. Note also that it is always possible to write a custom *QueryExecutor* without grepo being aware of that implementation. So you can use grepo's default configuration and write your own *QueryExecutor* and tell it to use this executor like this:

```
@GenericQuery(queryExecutor =  
MyCustomQueryExecutor.class)  
List<User> findUsingCustomExecutor();
```

Using this approach is straight forward, because you tell grepo what executor has to be used and thus the framework doesn't use the *QueryExecutorNamingStrategy* at all. That's good if you have only a few methods which require a special/custom executor. If you want to use your custom executors more frequently then it's probably better to configure the framework accordingly.

Separating queries from Java code

You may not want to have the queries directly in your Java code. Grepo uses Hibernate's concept of named queries to achieve that. This would make your repository methods even easier:

```
@GenericQuery  
User getByUsername(String username);
```

In our mapping file (`src/main/resources/META-INF/hibernate/User.hbm.xml`) we would define the the HQL query like this:

```
<query name="demo.domain.User.ByUsername">
    from User where username = ?
</query>
```

An appropriate SQL query would look like:

```
<sql-query
name="demo.domain.User.ByUsername">
    <return class="User" />
    select * from USERS where username = ?
</sql-query>
```

How it works

You may now wonder how grepo is able to find the correct query to execute (you will most likely have several named queries defined within your Hibernate session factory). The answer is conventions. Grepo uses a *org.codehaus.grepo.query.commons.executor.QueryExecutorFactory* which is responsible for creating *QueryExecutor* instances. Grepo's default implementation is *org.codehaus.grepo.query.commons.executor.QueryExecutorFactoryImpl* which uses a *org.codehaus.grepo.query.commons.naming.QueryNamingStrategy*. You could write your own implementation and tell grepo to use that strategy instead. Grepo's default implementation is *org.codehaus.grepo.query.commons.naming.QueryNamingStrategyImpl* which resolves query names for generic methods according to the following rules:

If the *queryName* property is set for the *GenericQuery* annotation, then this value is used as the query name. For instance:

```
@GenericQuery(queryName = "myqueryname")
User getByUsername(String username);
```

```
<query name="myqueryname">
    from User where username = ?
</query>
```

If the name of the method matches the pattern

```
^(is|has|find|get|load|scroll|iterate|delete|update)
```

then the query name is composed of the fully qualified entity class name and the method name (with removed prefix). This means that if the method starts with *is*, *has*, *get*, *load*, *scroll*, *iterate*, *delete* or *update* this rule will be applied. The example above shows how the query for the *getByUsername* method was resolved to "*demo.domain.User.ByUsername*". Note that the matched prefix is removed.

If the name of the method does not match the pattern above, then the query name is composed of the fully qualified entity class name and the complete method name (without removed prefix).

Using dynamically generated queries

So far we have only used static queries. But you may also want to generate your queries dynamically depending on the given method (input) parameters. In our application we may want a method which finds users via their *firstname* and/or *lastname*. For this we can either just implement the method ourselves (meaning not annotating the method with *GenericQuery*) or we could write an implementation of the *org.codehaus.grepo.query.hibernate.generator.HibernateQueryGenerator* interface. A simple implementation which generates a HQL query looks like:

```
package demo.repository;

import org.apache.commons.lang.StringUtils;
import
org.codehaus.grepo.query.commons.aop.QueryMethodParameterInfo;
import
org.codehaus.grepo.query.hibernate.generator
.AbstractHibernateQueryGenerator;

public class UserSearchQueryGenerator
extends AbstractHibernateQueryGenerator {

    public String
generate(QueryMethodParameterInfo qmpi) {
    String firstname =
```

```
qmpi.getParameterByParamName("fn",
String.class);
    String lastname =
qmpi.getParameterByParamName("ln",
String.class);

    StringBuilder query = new
StringBuilder("from User where 1=1");
        if
(StringUtils.isNotEmpty(firstname)) {
            query.append(" AND firstname =
:fn");
        }
        if
(StringUtils.isNotEmpty(lastname)) {
            query.append(" AND lastname =
:ln");
        }
```

```
        return query.toString();
    }
}
```

In our *UserRepository* we would define a method like this:

```
@GenericQuery(queryGenerator =
UserSearchQueryGenerator.class)
List<User> findUsersByName(
    @Param("fn") String firstname,
    @Param("ln") String lastname);
```

Note that it is required to generate queries with named-parameter style (as JDBC style does not work). If you want to generate a SQL (native) query instead of HQL you would just use *org.codehaus.grepo.query.hibernate.generator.AbstractHibernateNativeQueryGenerator* as the base class for your generator. Furthermore you can also use the Hibernate Criteria API like this:

```
package demo.repository;

import org.apache.commons.lang.StringUtils;
import
org.codehaus.grepo.query.commons.aop.QueryMe
thodParameterInfo;
import
org.codehaus.grepo.query.hibernate.generator
.CriteriaGenerator;
import
org.hibernate.criterion.DetachedCriteria;
import org.hibernate.criterion.Restrictions;
import demo.domain.User;

public class UserSearchCriteriaGenerator
```

```
implements CriteriaGenerator {

    public DetachedCriteria
generate(QueryMethodParameterInfo qmpi) {
        String firstname =
qmpi.getParameterByParamName("fn",
String.class);
        String lastname =
qmpi.getParameterByParamName("ln",
String.class);

        DetachedCriteria criteria =
DetachedCriteria.forClass(User.class);
        if
(StringUtils.isEmpty(firstname)) {

criteria.add(Restrictions.eq("firstname",
firstname));
        }
        if
(StringUtils.isEmpty(lastname)) {

criteria.add(Restrictions.eq("lastname",
lastname));
        }
    }
}
```

```
        return criteria;
    }
}
```

The method would be defined in the *UserRepository* as follows:

```
@GenericQuery
@HibernateQueryOptions(criteriaGenerator =
    UserSearchCriteriaGenerator.class)
List<User> findUsersByName(
    @Param("fn") String firstname,
    @Param("ln") String lastname);
```

Additional functionality

Paging

Hibernate offers *paging* functionality which can be used with grepo's *MaxResults* and *FirstResult* annotations like this:

```
@GenericQuery(query = "from User")
List<User> findUsers(@FirstResult int
    firstResult, @MaxResults int maxResults);
```

It's also possible to use the *firstResult* and *maxResults* properties of the *GenericQuery* annotation.

Result conversion

The framework supports result conversion functionality. For this grepo uses implementations of the *org.codehaus.grepo.core.converter.ResultConverter* interface. *ResultConverters* can be used to convert the result of a query invocation and may be configured for methods using grepo's *GenericQuery* annotation, like this:

```
@GenericQuery(resultConverter =  
MyResultConverter.class)  
int getBySomething(String something);
```

Hibernate also offers a similar feature known as *result transformation*. You can use this feature using grepo's *HibernateQueryOption* annotation by configuring the *resultTransformer* property accordingly.

Implicit result conversion

Furthermore grepo supports so called implicit result conversion, which means that in some cases (if necessary) the result will be converted automatically (without configuration). The framework uses a *org.codehaus.grepo.core.converter.ResultConverterFindingService*. Grepo's default implementation *org.codehaus.grepo.core.converter.ResultConverterFindingServiceImpl* uses a registry (basically a *Map*) to map (*return*) types to *ResultConverter* classes. The *ResultConverterFindingService* checks if conversion is required (for instance if query result is not compatible with method return type) and uses the *registry* in order to retrieve the appropriate converter to use. In its default configuration grepo knows the following *ResultConverters*:

- *org.codehaus.grepo.core.converter.ResultToBooleanConverter*: This implementation is used to convert objects to instances of *java.lang.Boolean*.
- *org.codehaus.grepo.core.converter.ResultToLongConverter*: This implementation is used to convert objects to instances of *java.lang.Long*.
- *org.codehaus.grepo.core.converter.ResultToIntegerConverter*: This implementation is used to convert objects to instances of *java.lang.Integer*.

Suppose we need functionality for our demo application which allows to register new users. As we have seen the *email* property is unique and thus must not already exist in the database. We could use the following method so we can check for existing email-addresses:

```
@GenericQuery(query = "select count(id) from  
User where email = ?")  
boolean isExistingEmail(String email);
```

Note that the query returns *0* or *1* (because *email* is unique) and the method's return type is *boolean*. Executing this method (and having grepo logger level set to *DEBUG*) the following will be logged:

```
15:43:30,212 DEBUG
[DefaultHibernateRepository:155] - Query
result is '0'
15:43:30,222 DEBUG
[ResultConverterFindingStrategyImpl:97] -
Found converter

'org.codehaus.grepo.core.converter.ResultToB
ooleanConverter' for conversion from
        'java.lang.Long' to
'boolean'
15:43:30,223 DEBUG
[ResultConversionServiceImpl:51] - Doing
conversion for result
        (class=java.lang.Long,
value=0) with converter

'org.codehaus.grepo.core.converter.ResultToB
ooleanConverter'
15:43:30,227 DEBUG
[ResultConversionServiceImpl:57] -
Conversion result is 'false'
```

Result validation

The framework supports result validation functionality. For this grepo uses implementations of the *org.codehaus.grepo.core.validator.ResultValidator* interface. *ResultValidators* can be used to validate the result of a method invocation. Note that result validation is performed after result conversion. *ResultValidators* can be configured for methods using grepo's *GenericQuery* annotation like this:

```
@GenericQuery(resultValidator =  
MyResultValidator.class)  
int getBySomething(String something);
```

Tracking statistics

The framework supports tracking statistics using grepo's Generic Statistics (gstatistics) component. The first step is to configure the gstatistics component accordingly. Using grepo's default configuration you have to import the default configuration of the grepo statistics component somewhere in your Spring application context:

```
<import  
resource="classpath:META-INF/grepo/grepo-sta  
tistics-default.cfg.xml" />
```

Furthermore you have to enable statistics. For instance this can be done for concrete repositories as follows:

```
<bean id="userRepository"  
parent="repositoryFactory">  
    <property name="proxyClass"  
value="demo.repository.UserRepository" />  
    <property name="statisticsEnabled"  
value="true" />  
</bean>
```

Using the grepo namespace the configuration above looks like:

```
<gquery:repository id="userRepository"  
factory="repositoryFactory"  
  
proxy-class="demo.repository.UserRepository"  
>  
    <property name="statisticsEnabled"  
value="true" />  
</gquery:repository>
```

You can also configure the *repositoryFactory* to enable statistics for all repository beans which use the factory as follows:

```
<bean id="repositoryFactory"  
class="org.codehaus.grepo.query.hibernate.re  
pository.HibernateRepositoryFactoryBean"  
abstract="true">  
    <property name="statisticsEnabled"  
value="true" />  
</bean>
```

Using the grepo namespace the configuration above looks like:

```
<gquery:repository-factory  
id="repositoryFactory">  
    <property name="statisticsEnabled"  
value="true" />  
</gquery:repository-factory>
```

See the [Generic Statistics getting started guide](#) for more information.

Transaction Handling

By default grepo does not handle transactions at all (this should be done by your service layer really) - you configure transaction handling for your repository objects (DAOs) as you would do normally with plain Spring/Hibernate. However grepo also optionally supports transaction handling using spring's transaction template. If you want grepo to handle transactions, you just configure the *transactionTemplate* property:

```
<gquery:repository-factory
id="repositoryFactory">
  <property name="transactionTemplate">
    <bean
class="org.springframework.transaction.support.
TransactionTemplate">
      <property
name="transactionManager"
ref="transactionManager" />
    </bean>
  </property>
</gquery:repository-factory>
```

Additionally you can configure the *readOnlyTransactionTemplate* property. Doing so grepo will use the read-only template for executing read-only operations. If no read-only template is defined, then grepo will use the *transactionTemplate* (if configured) for both read- and write-operations. Here is an example of the *repositoryFactory* bean with both templates defined:

```

<gquery:repository-factory
id="repositoryFactory">
  <property name="transactionTemplate">
    <bean
class="org.springframework.transaction.support.TransactionTemplate">
      <property
name="transactionManager"
ref="transactionManager" />
    </bean>
  </property>
  <property
name="readOnlyTransactionTemplate">
    <bean
class="org.springframework.transaction.support.TransactionTemplate">
      <property
name="transactionManager"
ref="transactionManager" />
      <property name="readOnly"
value="true" />
    </bean>
  </property>
</gquery:repository-factory>

```

A word about conventions

The grepo framework was designed around the [convention over configuration](#) paradigm. You configure the grepo framework with your guidelines (if the default configuration does not meet your needs) for your data access layer and grepo will then apply those rules transparently. For instance, we have seen that grepo can guarantee that all methods which execute *"list-queries"* have to start with the prefix *"list"* and also have to return an instance of *java.util.List*. If you want to break the rules you could for instance configure a method as follows:

```
@GenericQuery(queryExecutor =  
ListQueryExecutor.class)  
List<User> getByUsernames(@Param("list")  
Collection<String> usernames);
```

Note that the method name does not meet the conventions. The method starts with "get" but actually a "list-query" is required because of the method's return type (*java.util.List*). Here you can see the power of the convention over configuration paradigm and grepo in general. Breaking the rules is still possible but you have to know what you're doing and furthermore have additional configuration overhead.