Generic Procedure v1.0

Getting started with gprocedure

This is the getting started guide for the *Generic Procedure* (gprocedure) 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 gprocedure component. 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 Procedure* (gprocedure) component of the grepo framework version 1.0.x.

You can find the latest version of this guide here.

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-grepo-procedure-1.0.0
demo-grepo-procedure
```

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
```

Note: The demo application uses an Oracle database, so you need an Oracle instance running. Furthermore you need the appropriate jdbc driver in your classpath and you also will have to modify *src/main/resources/META-INF/spring/db-environment.xml* to configure the appropriate database connection settings. Additionally you need to setup the package used by the demo project by executing the sql scripts found in *src/main /resources/META-INF/oracle*. If you do not want to use Oracle, the demo project can easily be adapted to use different databases like Postgres etc...

You can now import the project in your eclipse workspace.

After you have imported the project you should now be able to run the *DemoProcedureTest* 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 commons-logging library. If you set the logger level for the package *org.codehaus.grepo* to *TRACE* in (*src/test/resources/log4j.xm*

/) like this:

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

Grepo should print out information similar to this:

```
00:06:00,822 TRACE [GenericProcedureMethodInterceptor:52] - Invoking method
'executeDemoProcedure'
00:06:00,835 TRACE [GenericProcedureRepositorySupport:96] - Executing
procedure using transaction template
00:06:00,840 TRACE [ProcedureCachingStrategyImpl:73] - Got procedure from
cache
                   key=executeDemoProcedure/grepo_demo.demo_procedure,
value=null
00:06:00,881 TRACE [ProcedureCompilationStrategyImpl:175] - Declaring
procedure param name=p_string,
                   type=org.springframework.jdbc.core.SqlParameter@78e45b5e
00:06:00,882 TRACE [ProcedureCompilationStrategyImpl:175] - Declaring
procedure param name=p_integer,
                   type=org.springframework.jdbc.core.SqlParameter@582ab653
00:06:00,883 TRACE [ProcedureCompilationStrategyImpl:175] - Declaring
procedure param name=p_result,

type=org.springframework.jdbc.core.SqlOutParameter@527f58ef
00:06:00,883 DEBUG [StoredProcedureImpl:156] - Compiled stored procedure.
Call string is
                   [{call grepo_demo.demo_procedure(?, ?, ?)}]
00:06:00,884 DEBUG [StoredProcedureImpl:342] - RdbmsOperation with SQL
[grepo_demo.demo_procedure] compiled
00:06:00,892 TRACE [ProcedureCompilationStrategyImpl:67] - Compiled stored
procedure:

org.codehaus.grepo.procedure.compile.StoredProcedureImpl@1b980630[sql=grep
o_demo.demo_procedure]
00:06:00,894 TRACE [ProcedureInputGenerationStrategyImpl:56] - Generated
input-map: {p_string=test, p_integer=1}
00:06:00,894 TRACE [GenericProcedureRepositoryImpl:83] - About to execute
procedure: grepo_demo.demo_procedure
00:06:00,895 TRACE [GenericProcedureRepositoryImpl:84] - Using input map:
{p_string=test, p_integer=1}
00:06:01,037 TRACE [GenericProcedureRepositoryImpl:90] - Procedure result
is '{p_result=p1:test p2:1}'
00:06:01,067 TRACE [GenericProcedureMethodInterceptor:72] - Invocation of
method 'executeDemoProcedure'
                   took '0:00:00.026'
```

Demo application

The teeny-weeny demo application consists of one database package (*GREPO_DEMO*) which contains one procedure and one function. The

spec and body for the package can be found in *src/main/resources/META-INF/oracle*. The package spec looks like:

```
CREATE OR REPLACE PACKAGE grepo_demo
IS
  PROCEDURE demo_procedure (
     p_string IN VARCHAR2,
     p_integer IN INTEGER,
     p_result OUT VARCHAR2);

  FUNCTION demo_function (
    p_string IN VARCHAR2,
    p_integer IN INTEGER)
    RETURN VARCHAR2;
END grepo_demo;
```

Using the grepo framework

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

- grepo-core-<*VERSION*>.jar
- grepo-procedure-<*VERSION*>.jar

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

```
<import resource="classpath:META-INF/spring/grepo-procedure-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. This configuration is "standard" Spring and is not grepo specific. In the demo project the oracle datasource is 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="abstractProcedureRepository"

class="org.codehaus.grepo.procedure.repository.GenericProcedureRepositoryFactoryBean"
    abstract="true">
    <property name="dataSource" ref="dataSource" />
    <property name="transactionTemplate">
        <bean class="org.springframework.transaction.support.TransactionTemplate">
            <property name="transactionManager" ref="transactionManager" />
        </bean>
    </property>
</bean>
```

Here we define an *abstract* bean which will be used for all of our concrete procedure repositories. Its a good idea (although not required) to define an *abstract* bean because this makes configuration of concrete procedure repository beans simpler and furthermore you just have to modify the *abstract* bean if you need to change the basic configuration for your data access layer in the future. Note that the *abstract* bean above uses a really simple configuration approach (just a data source and transaction template are configured - 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 simplest configuration of the *abstract* bean would look like this (as you can see the only property required is the data source reference):

```
<bean id="abstractProcedureRepository"

class="org.codehaus.grepo.procedure.repository.GenericProcedureRepositoryFactoryBean"
    abstract="true">
    <property name="dataSource" ref="dataSource" />
</bean>
```

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

Creating the first procedure repository

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

```
package demo.repository;

import java.sql.Types;
import java.util.Map;

import org.codehaus.grepo.procedure.annotation.GenericProcedure;
import org.codehaus.grepo.procedure.annotation.In;
import org.codehaus.grepo.procedure.annotation.Out;

public interface DemoProcedure {

    @GenericProcedure(sql = "grepo_demo.demo_procedure")
    @Out(name = "p_result", sqlType = Types.VARCHAR)
    Map<String, String> executeDemoProcedure(
        @In(name = "p_string", sqlType = Types.VARCHAR) String param1,
        @In(name = "p_integer", sqlType = Types.INTEGER) int param2);

    @GenericProcedure(sql = "grepo_demo.demo_function", function = true)
    @Out(name = "p_result", sqlType = Types.VARCHAR)
    Map<String, String> executeDemoFunction(
        @In(name = "p_string", sqlType = Types.VARCHAR) String param1,
        @In(name = "p_integer", sqlType = Types.INTEGER) int param2);
}
```

Note that we define two method - one for the procedure *demo_procedure* and one for the function *demo_function*.

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

```
<bean id="demoProcedure" parent="abstractProcedureRepository">
    <property name="proxyInterface" value="demo.repository.DemoProcedure" />
</bean>
```

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

- *proxyInterface*: 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.DemoProcedure* interfaces which we have created previously.

Finished! We can now inject our *demoProcedure* bean wherever we need to execute the procedure or function provided by the *GREPO_DEMO* p ackage.

Defining procedures and functions

As you can see above we have defined a *Map* as the return type for both interface methods. This is the default return type of a procedure/function invocation. If you don't need to know the result of the procedure/function invocation just use *void* for the method's return type. Using *Map* as the return type is not always a decend choice because the calling code must be aware (hardcode) of the names (=keys in the *Map*) in order to retrieve the appropriate return values. In our examples we have (in fact) just one return parameter (that is _p_result_) and can return the value directly like this:

```
@GenericProcedure(sql = "grepo_demo.demo_procedure", returnParamName = "p_result")
@Out(name = "p_result", sqlType = Types.VARCHAR)
String executeDemoProcedure(
    @In(name = "p_string", sqlType = Types.VARCHAR) String param1,
    @In(name = "p_integer", sqlType = Types.INTEGER) int param2);

@GenericProcedure(sql = "grepo_demo.demo_function", returnParamName = "p_result",
function = true)
@Out(name = "p_result", sqlType = Types.VARCHAR)
String executeDemoFunction(
    @In(name = "p_string", sqlType = Types.VARCHAR) String param1,
    @In(name = "p_integer", sqlType = Types.INTEGER) int param2)
```

Note that we use the *returnParamName* property of the *GenericProcedure* annotation to tell grepo which parameter (of the *Map*) has to be returned. Also note that the method's return types have therefore changed from *Map* to *String*.

Depending on your needs there are several ways to define the procedure/function (*IN*, *INOUT*, *OUT*) parameters. Above we have used the most simple/intuitive way. You could also for instance define the parameters as follows:

```
@GenericProcedure(sql = "grepo_demo.demo_procedure", returnParamName = "p_result")
@InParams({
    @In(name = "p_string", sqlType = Types.VARCHAR),
    @In(name = "p_integer", sqlType = Types.INTEGER)})
@Out(name = "p_result", sqlType = Types.VARCHAR)
String executeDemoProcedure(
    @Param("p_string") String param1,
    @Param("p_integer") int param2);
```

Note that we define all (*IN*, *OUT*) parameters at the method level and use grepo's *Param* annotation to map method parameters to procedure *IN* (or *INOUT*) parameters.

How it works

You have to be aware that it is required to define all procedure/function parameters for a Spring *StoredProcedure* in the correct order. For our *demo_procedure* this means (due to the spec) that we have to define the parameters in the following order:

- *IN* p_string
- *IN* p_integer
- *OUT* p_result

You may wonder how grepo can know in which order the parameters have to be defined as we didn't tell the framework the appropriate order explicitly. The answer is conventions. For a procedure grepo assumes that parameters have to be defined in the order: *IN*, *INOUT*, *OUT*. For a function grepo assumes that parameters have to be defined in the order: *OUT*, *INOUT*, *IN*. If your package/function or repository method does not meet those conventions you may either consider writing a custom implementation of grepo's *org.codehaus.grepo.procedure.compile.ProcedureCompilationStrategy* interface or just use the *index* property of the grepo's *In*, *InOut*, *Out* annotations. The following example would not work (based on the procedure spec above):

```
@GenericProcedure(sql = "grepo_demo.demo_procedure", returnParamName = "p_result")
@Out(name = "p_result", sqlType = Types.VARCHAR)
String executeDemoProcedure(
    @In(name = "p_integer", sqlType = Types.INTEGER) int param2,
    @In(name = "p_string", sqlType = Types.VARCHAR) String param1);
```

Note that we have switched the order of *param2* and *param1* in the method. So grepo would define the parameters in the following order:

- *IN* p_intger
- *IN* p_string
- *OUT* p_result

As stated above it is required to define procedure/function parameters in correct order. A solution might be use the *index* property of grepo's *In* an notation as follows:

```
@GenericProcedure(sql = "grepo_demo.demo_procedure", returnParamName = "p_result")
@Out(name = "p_result", sqlType = Types.VARCHAR)
String executeDemoProcedure(
    @In(name = "p_integer", sqlType = Types.INTEGER, index=2) int param2,
    @In(name = "p_string", sqlType = Types.VARCHAR, index=1) String param1);
```

Additional functionality

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 *GenericProcedure* annotation, like this:

```
@GenericProcedure(sql = "grepo_demo.demo_procedure", resultConverter =
MyResultConverter.class)
@Out(name = "p_result", sqlType = Types.VARCHAR)
SomeObject executeDemoProcedure(
    @In(name = "p_string", sqlType = Types.VARCHAR) String param1,
    @In(name = "p_integer", sqlType = Types.INTEGER) int param2);
```

Note that in this case the *MyResultConverter* class would be responsible to convert the result (*Map*) of the procedure invocation to an instance of *SomeObject*.

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*.

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 *GenericProcedure* annotation like this:

```
@GenericProcedure(sql = "grepo_demo.demo_procedure", returnParamName = "p_result",
    resultValidator = MyResultValidator.class)
@Out(name = "p_result", sqlType = Types.VARCHAR)
String executeDemoProcedure(
    @In(name = "p_string", sqlType = Types.VARCHAR) String param1,
    @In(name = "p_integer", sqlType = Types.INTEGER) int param2);
```

Transaction Handling

You configure transaction handling for your repositories as you would do normally with plain Spring. However grepo also optionally supports transaction handling using spring's transaction template (see *abstractProcedureRepository* above). If you don't want grepo to handle transactions you just do not configure the *transactionTemplate* property. 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 *abstractProcedureRepository* bean with both templates defined:

```
<bean id="abstractProcedureRepository"

class="org.codehaus.grepo.procedure.repository.GenericProcedureRepositoryFactoryBean"
    abstract="true">
    <property name="dataSource" ref="dataSource" />
    <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>
</bean>
```

Note that you have to set the *isReadOnly* property of the *GenericProcedure* annotation to *true* in order to mark a procedure/function as read-only operation.