SQL extension

User documentation

Send comments about this document to: bpajot@bluexml.com

August 11, 2009

Contents

History

Date	Author (mail)	Log
04/08/09	B.Pajot (bpajot@bluexml.com)	- Initial version of the document

Getting Started

Database Configuration Prerequisites

This version was tested with Mysql (v.5.0.67). However, the architecture should support other databases since the used dialect is generic. Beta-testers should submit comments regarding experiments of this module on other database systems (DBS).

You have to define a schema[1] (here denoted <synchro_schema>) and a user (here denoted <synchro_user>) with all privileges on this schema. The default configuration uses a schema called « alf_synchro » with a user named "alfresco" (using a password "alfresco") ; this parameters must be used if the configuration is not changed.

> *Using the existing alfresco database to store synchronization data is not considered a good practice, since separating the original Alfresco schema from the synchronization one becomes difficult; re-synchronizing the original data will be more difficult particularly during the development and testing phase. Furthermore, future versions could decide to re-synchronize data by dropping and re-creating the synchronization database.*

Module Installation

The module should be deployed automatically when the sql option is selected on generation (the license has to be valid). Alternatively, the packaged amp can be retrieved and installed manually with help of the existing Alfresco facilities (script "apply_amps.sh" or with the jar utility).

Module Configuration

Once deployed, the configuration of the module is gathered in the configuration file "`synchronisation.properties`"[2] located in:

```
$TOMCAT/webapps/alfresco/➥
WEB-INF/classes/alfresco/module/com_bluexml_side_Integration_alfresco_sql/
```

The parameters are those of the synchronization database parameters:

- `synchrodb.username` : the username
- `synchrodb.password` : the associated password
- `synchrodb.driver` : the name of the driver as used in the jdbc configuration
- `synchrodb.url` : the jdbc access url

The default configuration uses the following parameters:

```
synchrodb.username=alfresco
synchrodb.password=alfresco
synchrodb.driver=org.gjt.mm.mysql.Driver
synchrodb.url=jdbc:mysql://localhost/alf_synchro
```

Debugging Information

Various debug information can be activated. The configuration is located in the file "`log4j.properties`" that can be found in the directory:

1 w.r.t. MySQL terminology. On other DBS, a schema can be called database.
2 Remark the British spelling

```
$TOMCAT/webapps/alfresco/➥
WEB-INF/classes/alfresco/module/com_bluexml_side_Integration_alfresco_sql/
```

Default configuration uses an “error” level on all the available loggers. Users who want deeper information on what really happens can set the level to “debug”. More, finer tuning can be performed knowing the name of the loggers; this developer information can be found by retrieving the sources of the module.

Model Generation

Database Object Mapping

Alfresco uses an object model (a.k.a. class/association model). In order to use a relational database like MySql to store object-like structure, an object-relational mapping (ORM) has to be performed. Instead of using classical ORM technology (e.g. Hibernate or JPA), we defined an ad-hoc mapping, essentially for performance reasons.

Core mapping

- Each object generates a table. The table contains at least two columns, one for the database id (“dbid”) of the alfresco-database element and one for the Alfresco identifier (“uuid”). The default name of the table is defined by the said “short-name” of the BlueXML full qualified-name. This short-name is constructed by keeping the sole name of the class discarding the package prefix.
- Each attribute of an object generates a column in the corresponding table. A mapping is defined between the types used in Alfresco (Java-based types) and the relational database types. A default generic mapping is performed; however, a custom dialect can be developed if necessary (refer to the corresponding developer section). The default name of the attribute is also the short-name, which is composed of the sole attribute name (discarding the package prefix and the name of the containing class).
- *Each association definition generates a table with two columns, one for the source id, and the other for the target id. An association id is also defined but currently not used. The name of the table is defined through this simple concatenation:*

  `<SourceClass>_[<SourceRole>_]<Association>_[<TargetRole>_]<TargetClass>`

  The names of the columns are either the name of the linked class, or if defined, the name of the role. Roles on both association ends are mandatory on reflexive associations; SIDE also defines a default role on oriented associations on the target side. One of the best-practices advocates to always define a role on both sides of the association.

Aspected classes

Regarding the Alfresco terminology, aspects enable to factorize specific behavior that can be reused across various classes. Aspect attributes are just considered as intrinsic attributes of the generated table. Similarly, aspect associations are just considered as normal associations and do not yield to a different behavior than normal associations. The default naming of the attributes and associations remains the same.

> *Aspects by themselves are not represented as separate tables. In the earlier tests, we provided this tables, however this mechanism was outside of the Alfresco aspect philosophy. More, having such an information was not helpful in the applications we developed on top of sql synchronization.*

Inheritance

Inheritance is one of the feature that brings the now well-known impedance mismatch in ORMs. The implementation we chose mainly relies on redundancy of the information; querying the database with

traditional SQL is thus made easier, providing an object-oriented querying language being out of th scope of the first implementation.

On the specialized-class side, all the inherited attributes are part of the table. Inherited associations do not generate supplementary tables. Instead we chose to keep polymorphic tables. On the parent-class side, the structure definition is standard; however, stored data gather all the descendant tuples defined by the traditional inheritance process. The same process holds for descendant[3] associations.

Customizing the name-mapping

A name mapping can be enforced by the use of a simple "properties" file. Even if not considered as a good practice (this file is generated and should be managed through the editor), this method can be useful in particular situations.

The file named "`synchronisation-database-mapping.properties`" can be found in the following location:

```
$TOMCAT/webapps/alfresco/➥
WEB-INF/classes/alfresco/module/<SIDE model-extension module>/
```

where <SIDE model-extension module> defines the name of the generated model extension module (currently the name is `SIDE_ModelExtension_<top-package name>`).

The format of the file is quite intuitive. The following definition describes the default generated file:

```
class.name.<Class FQN>=<Class SN>
class.attribute.name.<Class FQN>.<Attribute FQN>=<Attribute SN>
association.name.<Association FQN>=<Association SN>
association.source.<Association FQN>=<Class SN>
association.source.alias.<Association FQN>=<Source Association End Role>
association.target.<Association FQN>=<Class SN>
association.target.alias.<Association FQN>=<Target Association End Role>
```

where FQN means "Full Qualified Name" and SN means "Short Name".

The "`alias`" property used on source and target associations is used to define an alias on the column name (initially, this property was mainly used on reflexive associations for which the source class name and the target class name were identical). These properties are optional; if not defined, the source and class names are used instead.

Known limitations and problems

Namespace limitations

The use of short names has a main limitation regarding conflicting names, when a class of the same name belongs to different packages of the same model. More, the problem also occurs when defining the same name in different related or unrelated model. This is however a limitation that can be managed locally by using custom name mapping.

The naming strategy is mainly due to database naming limitations. Most of the database systems provides a more or less restrictive limitation, MySQL using for example a limitation of 64 characters on the table name.

Current name mapping is generated automatically from the model. Using this technique can still lead to problems regarding the table name length (mainly on association table names). The current implementation supports automatic renaming of offending table names. Current strategy retrieves the table-name maximum length from the database meta-data and split the name at the appropriate size. This rough strategy can be

3 All the children of a common ancestor

customized by developers or integrators (customizing the injected beans). Another strategy only raises an error on offending table names, enabling the user to solve the problem manually.

Static mapping structure & redundancy

The current object-relational mapping cannot be customized. One may indeed want to choose a different structure in order to free some space on a huge repository. The ability to choose between foreign keys or association tables is a classical functionality of ORM systems (even if JPA uses an association table by default). Another one is related to the inheritance mapping, some forms avoiding data-redundancy (thus complexifying the writing of SQL queries).

Beyond the inheritance representation trick, the object to relational-database mapping raises other problems: one is the association orientation at the object level, which property cannot be maintained at relational level. Indeed, association tables are "orientation-agnostic". In fact, orientation is implicit with the name of the table (which is the name of the association). This is also another point of redundancy, since for a double-navigable association A<->B, both A->B and B->A association tables exist. The information in both tables is equivalent. Removing one of the tables could be possible, but writing queries with pure SQL would be less easy since we would have to know which table is available. This problem would be solved if a higher-level (object-oriented) language was used.

Database size and performance degradation

The technique we used replicates data from Alfresco repository to a relational database. Data is thus stored twice involving twice as much space. Moreover, replicating data also costs time, thus degrading global Alfresco writing performances.

One of the technique we used earlier in the test phase was the use of SQL views which provided a logical view of the data already existing in the Alfresco database. However two major issues raised: 1) view creation was related to a low-level database model, which was subject to change[4] and not well documented 2) performance on querying the views was terrible and thus unusable in real applications. The use of materialized views could have been a solution, however this functionality is only supported by few – mainly commercials – vendors, this choice being thus quite restrictive for future partners. Materialized views are based on some trigger technique at the database level; we chose to raise this method at a upper level where we have a knowledge of the Alfresco base objects (nodes and associations), thus providing a better integration.

Polymorphic association tables

The choice of polymorphic tables was made to remove partially redundancy. This choice has a little impact on keeping trace of the associations related to a particular class. If not impossible, this queries are not as so easy (efficient) as a count on a simple table. This kind of functionality seems hopelessly not so usual, this design choice having few impact on the use of the replicated database.

4 As it happened between major versions 2.x and 3.x