APPLY HIBERNATE TO MODEL AND PERSIST ASSOCIATIONS MAPPINGS IN DOCUMENT VERSION SYSTEM

Neetu Gupta

AIIT, Amity University, Sector- 125, Noida
neetugupta78@gmail.com

ABSTRACT

To implement any system using hibernate, a XML (Extensible Markup Language) mapping is defined that specifies how the various objects will persist in the database. The XML specifies mapping between classes and tables, properties and columns, association and foreign keys. In this paper we primarily focus on how different association among classes should be mapped in XML configuration file for persistence. A particular type of association should be specified in XML configuration file using valid elements and tags provided by hibernate. Such a defined XML document then create the valid schema, and the constraints between respective tables in the database to implement any system. Here we present how the different types of associations should be implemented in hibernate to have desired effect on the database. We also tested the use cases those may violate the constraints specified by XML configuration and studied the results.

Keywords

ORM (Object Relational Mapping), Hibernate, XML, Association

1. INTRODUCTION

The paper presents the implementations of different types of associations between classes in a document versioning system using Hibernate. In any system there exist different types of relationship between classes. ORM tool requires a mapping that specifies persistence of objects or classes as tables and properties of classes as columns. Similarly an association between classes should also be properly specified in XML mapping in order to persist it correctly in the database usually as foreign keys. Hibernate provides various elements and tags for defining different types of associations in XML configuration file.

We developed and tested a prototype of Document versioning system to model, understand and implement all the possible existing unidirectional associations. Our focus was to understand and test how each type of association should be specified in the XML configuration to have desired constraints implemented in the database. We studied the all possible different relationship types and how these can be mapped in XML to create desired schema and database entities on the other side in Hibernate. We have used Eclipse IDE (Interactive Development Environment) to develop and test the prototype. We also used hsqldb (HyperSQL DataBase) to simulate the database part required for the project.

2. PROPOSED SYSTEM FOR STUDYING ASSOCIATION

Versioning systems have always supported the document / code base management for any organization. Any such software always acts as a book keeping system for documents repository or for maintaining source code base of any developing software. It gives the facility to track the changes done in the particular document with other important associated information like user name that has changed the document, purpose for the change, and to see the exact changes done between two versions of a particular document. Based on entities discussed, the classes and associations between them are shown in fig. 1 and fig. 2 respectively.

2.1 Entities Involved The proposed system has the entities discussed in this section.

Document, an entity represents a document, a source code file, an image file etc. DocumentType an entity represents type like doc, jpg, htm etc. Version represents the changed version of the original document. User represents an authorized user who can create a new document or version. Tag, represent a series that have multiple documents say for a particular release. Contact, represents the contact of a user.

2.2 Implementation Details

To create a valid schema of the system, each class will have a source code java file and a mapping XML configuration file.

Source code. An entity is defined as a class and attributes as members of the class. A class will have one or more public constructors with a mandatory default constructor. Class will also define pair of get and set method for each property defined in it. Such a java code for Document class with basic attributes is as below. Similarly, we have implemented the classes for other entities as well.

```
 public class Document {
     private long documentId;
     private String name;
     public Document() {}      //default constructor
     public Document(String name) { this.name = name ;}
     public long getDocumentId() {return documentId;}
     public void setDocumentId(long documentId) {
      this.documentId = documentId;}
     public String getname() {           return this.name;}
     public void setname(String name) {this.name = name;}
}
```

Mapping File. To persist each class, we defined a XML based configuration file that maps a class as a table and its members as columns. A member can be defined Whereas an attribute could also be mapped with reference of other class representing the relationship between tables. One such XML configuration mapping file mapping Document class and its basic attributes in is presented here.

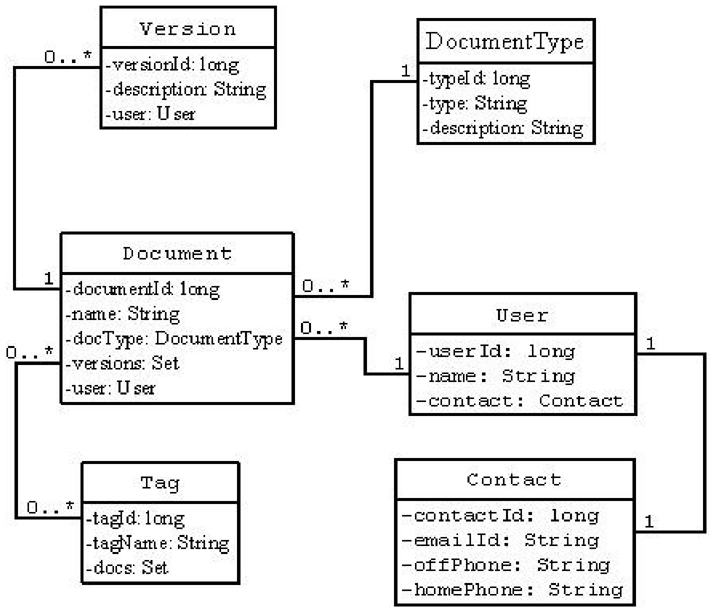

Fig. 1. Complete Class diagram of the system depicting the associations among classes

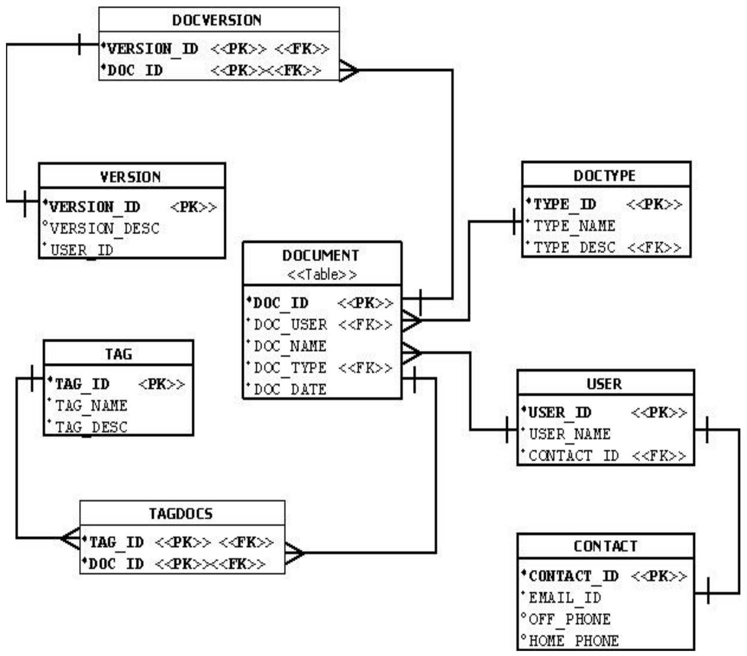

Fig. 2. Corresponding schema created on the database representing different association types

```
<? Xml version="1.0"?>
<hibernate-mapping>
<class name="docs.Document" table="DOCUMENT">
    <id name="documentId" column="DOC_ID">
        <generator class="native"/> </id>
       <property name="name" type="string" length="100" not-null="true"
       column="DOC_NAME"/>
</class>
</hibernate-mapping>
```

In discussed mapping file we have not yet included the attributes those create an association between classes. Refer to figure 1 such attributes for class (or say entity) *Document* are *type, version* and *user.* We discuss these attributes and related association in next section in detail.

3. ASSOCIATIONS AND THEIR IMPLEMENTATION

Here we discuss implementation of all possible types of unidirectional associations. We discuss how each association type is mapped into configuration file and various elements suggested in hibernate. We also discuss the affect of mappings on database.

3.1 Many-to-one unidirectional association Using Foreign Key.

Association from *Document* to *DocumentType* is a *many-to-one* kind of unidirectional association. For a *DocumentType* instance, there can be multiple *Document* instances, whereas, a *Document* can be of one *DocumentType* only. With the same explanation we can define association from Document to User as many-to-one (fig 3).

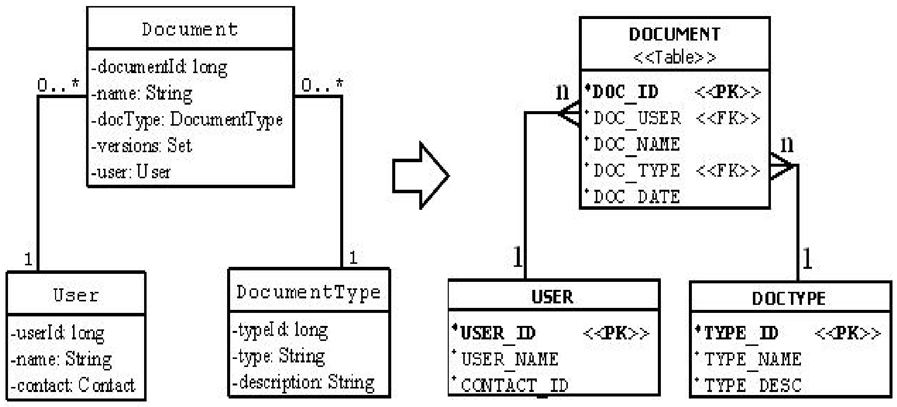

Fig. 3. Many-to-one unidirectional association using foreign key

This is mapped as foreign key constraint in the *Document* table as depicted in the fig. 3. To specify this mapping in *Document* class, we used the element <many-to-one> in corresponding mapping xml file Document.hbm.xml as

```
   <many-to-one  name=" docType" class="docs.DocumentType"
   column="DOC_TYPE" cascade="save-update"
         not-null="true" />
```

On defining this association, the object reference returned by *getDocType*() method of *Document* class is mapped to a foreign key column DOC_TYPE in the DOCUMENT table with reference to *DocumentType* class. The specified not-null attribute creates a not null constraint for column DOC_TYPE so we cannot have a DOCUMENT without a value from the domain for the field DOC_TYPE. Element cascade with "save-update" tells hibernate to navigate this association while transaction is committed and save the newly instantiated transient instances (*DocumentType* here) and persist changes to detached instances if any. Similarly, the association from Document to User is defined. Relevent java code and mapping would be

```
public class Document {
        ……
        private User user;
        public User getUser() {  return this.user; }
        public void setUser(User user) {this.user = user;}
        ……
}

<many-to-one name="user" class="docs.User"
        column="DOC_USER" cascade="save-update"
        not-null="true"
/>
```

An object reference returned by *getUser ()* is mapped to a foreign key column DOC_USER in the *DOCUMENT* table. Attribute *not-null* is true implementing not null constraint for *DOC_USER* and *cascade* is set to *save-update* to make transient object persistent. The application code to persist an instance of Document, User and DocumentType is like

```
   ………
1. tx = session.beginTransaction();
2. DocumentType doctype = new DocumentType ("txt", "MS  Word Document");
3. User user = new User ("Neetu");
4. Document doc1 = new Document("projectReport1", doctype, user);
5. session.save(doc1); // Persist Document instance
6. tx.commit();
```

Saving an instance of *Document* (line 5) will insert a row to *DOCUMENT* table and transient objects of *DOCTYPE* and *USER* due to *cascade* attribute set to *save-update*. Without setting the cascade attribute, we should be saving the instances of *DocType* and *User* independently (line 5 and 8 below) before saving the *Document* object like

3.2 One-to-many Unidirectional Using join table.

Unidirectional one-to-many association is not recommended using foreign key column. We used join table to implement the same. Mapping from class *Document* to *Version* is one-to-many. For a Document instance we can have multiple Version instances. To implement this, a join table *DOCVERSION* is created as shown in fig. 4.

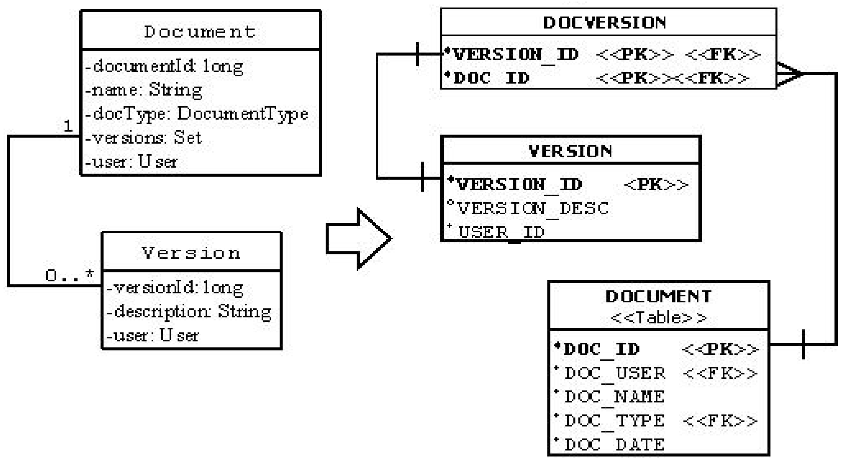

Fig. 4. One-to-many unidirectional using join table

We implemented the version in Document class using a collations type Set as

```
.............
private Set versions = new HashSet();
public Set getVersions () {return this.versions;}
public void setVersions (Set versions) {
        this.versions= versions; }
```

Hibernate element <many-to-many> is used with unique constraint set to true. Unique property is used to map the "one" side of one-to-many resulting in the uniqueness constraint to makes sure that a particular Version instance is not associated with multiple instances of Document. Corresponding update the configuration file Document.hbm.XML metadata using <set> element as

```
<set Name = "versions" table="DOCVERSIONS">
<key column = "DOC_ID"/>
<many-to-many column="VERSION_ID"
      class ="docs.Version" unique="true" />
</set>
```

We tested application to create a new version for an already existing Document object. Line 5, 7 saves two objects of type Version, to add two instances in VERSION table. Line 10, 11 will attach two versions to a Document doc1that results in saving two rows in table DOCVERSION.

```
1.    ..........
2.    tx = session.beginTransaction ();
3.    Version ver1 = new Version ("verison1", user);
4.    session.save (ver1);
5.    Version ver2 = new Version ("verison2", user);
6.    session.save (ver2);
7.    // Load the already saved Document instance
8.    Document doc1 = (Document) session.load
(Document.class, doc1Id);
```

```
9.    // Add version instance ver1, ver2 to Document doc1
10.   doc1.getVersions().add(ver1);
11.   doc1.getVersions().add(ver2);
```

We tested unique constraint attaching ver1 to another document doc2

```
Document doc2 = (Document)session.load
(Document.class, doc2Id);
doc2.getVersions ().add(ver1);
```

It throws an exception saying violation of integrity constraint. Result is that we can save multiple Version instances with a single Document but not otherwise.

3.3 One-to-one unidirectional using foreign key.

For one-to-one association type, we introduced Contact class in the system. An instance of Contact type stores contact for a User instance. We cannot have more than one Contact for a User. This defines association from User to Contact as one-to-one.

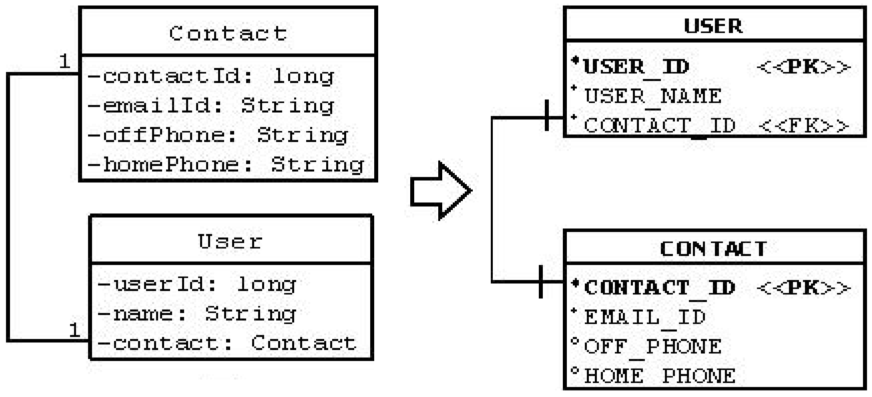

Fig. 5. One-to-one using foreign key

We implemented this using a foreign key column *CONTACT_ID* in *USER* table as shown in fig 5. We used element many-to-one with unique attribute set to true to enforce unique constraint on the User side as well on Contact object as

```
<many-to-one name="contact" class="docs.Contact"
      column="CONTACT_ID" cascade ="save-update" not-null="true" unique = "true" />
```

We tested application code that create a Contact instance and associate the same with an User instance as

```
1.  ....................
2.  tx = session.beginTransaction ();
3.  Contact contact1 =
            new Contact("xyz@abc.com", "12345", "12345");
4.  User user1 = new User ("Neetu");
5.  user.setContact (contact);
```

```
6. session.save (user1);
7. tx.commit();
```

Persisting an object of type User (line 6) automatically saves the transient Contact instance contact1 into CONTACT table. Attempt to attach contact1 with user2 object generates an integrity violation error.

```
User user2 = new User ("user1");
User2.setContact (contact1);
```

Caused by: java.sql.SQLException: Unique constraint violation: SYS_CT_472 in statement [insert into USER (USER_ID, USER_NAME, CONTACT_ID) values (null, ?, ?)] at org.hsqldb.jdbc.Util.throwError(Unknown Source)at rg.hsqldb.jdbc.jdbcPreparedStatement.executeUpdate(Unknown Source) at org.hibernate.id.insert.AbstractSelectingDelegate.performInsert(AbstractSelectingDelegate.java: 33)

3.4 Many-to-many Unidirectional.

Association between *Tag* and *Document* is of many-to-many type. For an instance of *Tag*, there will be multiple instances of *Document* and vice versa. We successfully implemented this association using the join table as depicted in fig. 6.
We created a join table *TAGDOCS* to persist Collection *docs* added in *Tag* class. For now, we only implemented it unidirectional from *Tag* to *Document* as

```
private Set docs = new HashSet();
public Set getDocs() {return this.docs;}
public void setDocs(Set docs) {this.docs = docs;}
……………..
```

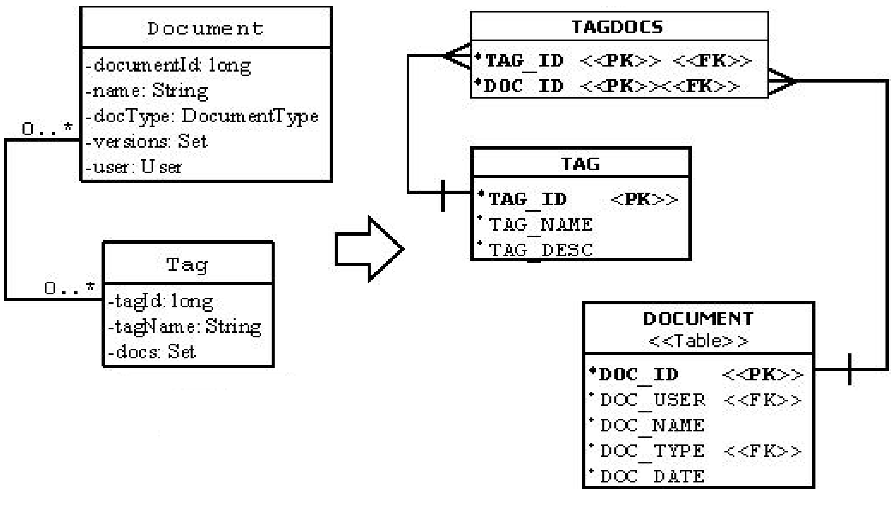

Fig 6. Many-to-many unidirectional association and mapping using join table

This association is defined using *many-to-many* property without setting unique to true opposite to the case of one-to-many. We have used the property <Set> to persist the collection to create join table *TAGDOCS*. Added the following mapping in XML configuration file of Tag class

```
 <set name = "docs" table="TAGDOCS">
   <key column = "TAG_ID"/>
   <many-to-many column="DOC_ID" class ="docs.Document"/>
  </set>
```

The related java code to associate a *Document* instance to a *Tag* is tested. A Tag object is persisted in *TAGS* table (line 4) and it is associated with already saved *Document* objects with doc1and doc2 with tag object.

```
1.  ...........
2.  tx = session.beginTransaction();
3.  Tag tag = new Tag("3.0");
4.  session.save(tag);
5.  tag.getDocs().add(doc1);
6.  tag.getDocs().add(doc2);
7.  tx.commit();
8.  ............
```

4. CONCLUSIONS AND FUTURE SCOPE

We have implemented and tested a prototype of Document versioning system using ORM hibernate to understand how different types of possible associations among classes can be implemented to generate a valid schema and database. We also studied how to incorporate the constraints like uniqueness, cascading persistence of multiple objects by defining a suitable XML in hibernate. We tested the code that creates and update database using hibernate query. In future work, we will incorporate the bidirectional associations in system wherever possible and will study and test the hibernate implementation required. The complete source code of the system is available with the author.

ACKNOWLEDGMENT

The writer thanks Mr. Ashutosh Gupta, the reviewer of the paper for his continues support towards understanding hibernate and troubleshooting during setting up the eclipse, hsqldb framework and narrow down other technical issues.

REFERENCES

[1] Hibernate, http://www.hibernate.org
[2] Christian Bauer, Gavin King, "Hibernate in Action", Manning
[3] Bauer, C, King, G 2006 Java Persistence with Hibernate, Manning, Manning 2007
[4] Dia, https://live.gnome.org/Dia
[5] http://www.eclipse.org/
[6] http://hsqldb.org