ODMG-api Guide

by Armin Waibel

Table of contents

1. Introduction

The *ODMG API* is an implementation of the ODMG 3.0 Object Persistence API (http://www.odmg.org/) . The ODMG API provides a higher-level API and OQL query (../../docu/guides/query.html#odmg-oql) language based interface over the PersistenceBroker API (../../docu/guides/pb-guide.html) .

This document is not a ODMG tutorial (../../docu/tutorials/odmg-tutorial.html) (newbies please read the tutorial first) rather than a guide showing the specific usage and possible pitfalls in handling the ODMG-api and the proprietary extensions by OJB.

If you don't find an answer for a specific question, please have a look at the FAQ (../../docu/faq.html) and the other reference guides (../../docu/guides/summary.html) .

2. Specific Metadata Settings

To make OJB's *ODMG-api* implementation proper work some specific metadata settings needed in the repository mapping files (../../docu/guides/repository.html) .

All defined reference-descriptor (../../docu/guides/repository.html#reference-descriptor) and collection-descriptor (../../docu/guides/repository.html#collection-descriptor) need specific *auto-xxx* settings:

- auto-retrieve="true"
- auto_update="none"
- auto-delete="none"

So an example object mapping class-descriptor (../../docu/guides/repository.html#class-descriptor) look like:

```
<class-descriptor
    class="org.apache.ojb.odmg.shared.Master"
    table="MDTEST_MASTER"
    >
    <field-descriptor
        name="masterId"
        column="MASTERID"
        jdbc-type="INTEGER"
        primarykey="true"
        autoincrement="true"
        />
    <field-descriptor
        name="masterText"
        column="MASTER_TEXT"
        jdbc-type="VARCHAR"
        />
    <collection-descriptor
        name="collDetailFKinPK"
        element-class-ref="org.apache.ojb.odmg.shared.DetailFKinPK"
        proxy="false"
        auto-retrieve="true"
        auto-update="none"
        auto-delete="none"
        >
        <inverse-foreignkey field-ref="masterId"/>
    </collection-descriptor>
...
</class-descriptor>
```

A lot of mapping samples can be found in mappings for the OJB test suite (../../docu/testing/testsuite.html) . All mappings for the ODMG unit test are in `repository_junit_odmg.xml` file, which can be found under the *src/test* directory.

3. How to access ODMG-api

Obtain a `org.odmg.Implementation` instance first, then create a new `org.odmg.Database` instance and open this instance by setting the used jcd-alias (../../docu/guides/repository.html#jdbc-connection-descriptor) name:

```
Implementation odmg = OJB.getInstance();
Database database = odmg.newDatabase();
database.open("jcdAliasName#user#password", Database.OPEN_READ_WRITE);
```

The *user* and *password* separated by # hash only needed, when the user/passwd is not specified in the connection metadata (jdbc-connection-descriptor).

The jdbc-connection-descriptor (../../docu/guides/repository.html#jdbc-connection-descriptor) may look like:

```
<jdbc-connection-descriptor
                   jcd-alias="jcdAliasName"
                   ...
                   username="user"
                   password="password"
                   ...
     >
    ...
</jdbc-connection-descriptor>
```

With method call `OJB.getInstance()` always a **new** org.odmg.Implementation (../../api/org/odmg/Implementation.html) instance will be created and `odmg.newDatabase()` returns a new `Database` instance.

For best performance it's recommended to share the `Implementation`

(../../api/org/odmg/Implementation.html) instance across the application. To get the current open database from the `Implementation` instance, use method `Implementation.getDatabase(null)`

```
Implementation odmg = ....
// get current used database
Database database = odmg.getDatabase(null);
```

Or share the open `Database` instance as well.

See further in FAQ "Needed to put user/password of database connection in repository file?" (../../docu/faq.html#userPasswordNeeded) .

4. Configuration Properties

The OJB *ODMG-api* implementation has some adjustable properties and pluggable components. All configuration properties can be set in the OJB.properties (../../OJB.properties.txt) file.

Here are all properties used by OJB's *ODMG-api* implementation:

Property Name	Description
OqlCollectionClass	This entry defines the collection type returned from OQL queries (../../docu/guides/query.html#odmg-oql) . By default this value is set to a List. This will be good for most situations. If you need the additional features of the DList interface (DList itself is persistable, support of predicate) change setting to the DList implementation (See also property 'DListClass' entry). Using DLists for large resultsets may be bad for application performance. For these scenarios you can use ArrayLists or Vectors. Important note: The collection class to be used MUST implement the interface `org.apache.ojb.broker.ManageableCollection`.
ImplementationClass	Specifies the used base class for the *ODMG API* implementation. In managed environments a specific class is needed to potentiate JTA integration of OJB's ODMG implementation.
OJBTxManagerClass	Specifies the class for transaction management. In managed environments a specific class is needed to potentiate JTA integration of OJB's

	ODMG implementation.
cascadingDeleteOneToOne	If set *true* the cascading delete for 1:1 references will be enabled. This means that when an object A with 1:1 reference to B will be deleted, B will deleted too and all 1:1 references used by B and so on. When set *false* cascading delete is disabled for 1:1 references.
cascadingDeleteOneToN	If set *true* the cascading delete for 1:n references will be enabled. This means that when an object A with 1:n reference to B will be deleted, B will deleted too and all 1:n references used by B and so on. When set *false* cascading delete is disabled for 1:n references. In this case the referenced (*n-side*) objects will be *unlinked* (../../docu/guides/basic-technique.html#linking) - all FK values of the referenced objects will be *nullified*.
cascadingDeleteMToN	If set *true* the cascading delete for m:n references will be enabled. This means that when an object A with m:n reference to B will be deleted, B will deleted too and all m:n references used by B and so on. When set *false* cascading delete is disabled for m:n references. In this case only the m:n indirection table entries will be deleted, the referenced objects will be untouched.
ImplicitLocking	This property defines the *implicit locking* behavior. If set to *true* OJB implicitely locks objects to ODMG transactions after performing OQL queries or when do a single lock on an object using `Transaction#lock(...)` method. If implicit locking is used locking objects is recursive, that is associated objects are also locked. If ImplicitLocking is set to *false*, no locks are obtained in OQL queries and there is also no recursive locking when do single lock on an object.

LockAssociations	This property was only used when *ImplicitLocking* is enabled. It defines the behaviour for the OJB implicit locking feature. If set to *true* acquiring a write-lock on a given object x implies write locks on all objects associated to x. If set to *false*, in any case implicit read-locks are acquired. Acquiring a read- or write lock on x thus allways results in implicit read-locks on all associated objects.
DListClass	The used `org.odmg.DList` implementation class.
DArrayClass	The used `org.odmg.DArray` implementation class.
DMapClass	The used `org.odmg.DMap` implementation class.
DBagClass	The used `org.odmg.DBag` implementation class.
DSetClass	The used `org.odmg.DSet` implementation class.

5. OJB Extensions of ODMG

This section describes the propietary extension of the *ODMG-api* provided by OJB.

5.1. The ImplementationExt Interface

The OJB extension of the odmg Implementation (../../api/org/odmg/Implementation.html) interface is called ImplementationExt (../../api/org/apache/ojb/odmg/ImplementationExt.html) and provide additional methods missed in the standard class definition.

- get/setOqlCollectionClass
  Use this methods to change the used OQL query result class at runtime. Description can be found in *Configuration Properties* section and in javadoc of ImplementationExt (../../api/org/apache/ojb/odmg/ImplementationExt.html) .
- is/setImpliciteWriteLocks
  Use this methods to change the associated locking type at runtime when implicit locking is used. Description can be found in *Configuration Properties* section and in javadoc of ImplementationExt (../../api/org/apache/ojb/odmg/ImplementationExt.html) .

5.2. The TransactionExt Interface

The OJB extension of the odmg Transaction (../../api/org/odmg/Transaction.html) interface is called TransactionExt (../../api/org/apache/ojb/odmg/TransactionExt.html) and provide additional methods missed in the standard class definition.

- markDelete
  Description can be found in javadoc of TransactionExt (../../api/org/apache/ojb/odmg/TransactionExt.html) .
- markDirty
  Description can be found in javadoc of TransactionExt (../../api/org/apache/ojb/odmg/TransactionExt.html) .
- flush
  Description can be found in javadoc of TransactionExt (../../api/org/apache/ojb/odmg/TransactionExt.html) .
- is/setImplicitLocking
  Description can be found in javadoc of TransactionExt (../../api/org/apache/ojb/odmg/TransactionExt.html) .
- setCascadingDelete
  Description can be found in javadoc of TransactionExt (../../api/org/apache/ojb/odmg/TransactionExt.html) .
- getBroker()
  Returns the current used broker instance. Usage example is here.

5.3. The EnhancedOQLQuery Interface

The OJB extension of the odmg OQLQuery (../../api/org/odmg/OQLQuery.html) interface is called EnhancedOQLQuery (../../api/org/apache/ojb/odmg/oql/EnhancedOQLQuery.html) and provide additional methods missed in the standard class definition.

- create(String queryString, int startAtIndex, int endAtIndex)
  Description can be found in javadoc of EnhancedOQLQuery (../../api/org/apache/ojb/odmg/oql/EnhancedOQLQuery.html) .

5.4. Access the PB-api within ODMG

As the PB-api (../../docu/guides/pb-guide.html) was used by OJB's *ODMG-api* implementation, thus it is possible to get access of the used `PersistenceBroker` instance using the extended Transaction interface class TransactionExt (../../api/org/apache/ojb/odmg/TransactionExt.html) :

```
Implementation odmg = ...;
```

```
TransactionExt tx = (TransactionExt) odmg.newTransaction();
tx.begin();
...
PersistenceBroker broker = tx.getBroker();
// do work with broker
...
tx.commit();
```

It's mandatory that the used *PersistenceBroker* instance **never** be closed with a `PersistenceBroker.close()` call or be committed with `PersistenceBroker.commitTransaction()`, this will be done internally by the ODMG implementation.

6. Notes on Using the ODMG API

6.1. Transactions

The ODMG API uses *object-level transactions*, compared to the PersistenceBroker *database-level transactions*. An ODMG `Transaction` (../../api/org/odmg/Transaction.html) instance contains all of the changes made to the object model within the context of that transaction, and will not commit them to the database until the ODMG `Transaction` is committed. At that point it will use a database transaction (the underlying PB-api) to ensure atomicity of its changes.

6.2. Locks

The ODMG specification includes several levels of locks and isolation. These are explained in much more detail in the Locking (../../docu/guides/lockmanager.html) documentation.

In the ODMG API, locks obtained on objects are locked within the context of a transaction. Any object modified within the context of a transaction will be stored with the transaction, other changes made to the same object instance by other threads, ignoring the lock state of the object, will also be stored - so take care of locking conventions.
The ODMG locking conventions (obtain a write lock before do any modifications on an object) ensure that an object can only be modified within the transaction.

It's possible to configure OJB's ODMG implementation to support implicit locking with *WRITE* locks. Then a write lock on an object forces OJB to obtain implicit write locks on all referenced objects. See configuration properties.

6.3. Persisting Non-Transactional Objects

Frequently, objects will be modified outside of the context of an ODMG transaction, such as a data access object in a web application. In those cases a persistent object can still be modified, but not directly through the *OMG ODMG specification*. OJB provides an extension to the ODMG specification for instances such as this. Examine this code:

```
public static void persistChanges(Product product)
{
    Implementation impl = OJB.getInstance();
    TransactionExt tx = (TransactionExt) impl.newTransaction();

    tx.begin();
    tx.markDirty(product);
    tx.commit();
}
```

In this function the product is modified outside the context of the transaction, and is then the changes are persisted within a transaction. The `TransactionExt.markDirty()` method indicates to the Transaction that the passed object has been modified, even if the Transaction itself sees no changes to the object.

7. Questions

7.1. I don't like OQL, can I use the PersistenceBroker Queries within ODMG

Yes you can! The ODMG implementation relies on PB Queries internally! Several users (including myself) are doing this.

If you have a look at the simple example below you will see how OJB Query objects can be used withing ODMG transactions.
The most important thing is to lock all objects returned by a query to the current transaction before starting manipulating these objects.
Further on do not commit or close the obtained PB-instance, this will be done by the ODMG transaction on `tx.commit() / tx.rollback()`.

```
TransactionExt tx = (TransactionExt) odmg.newTransaction();
tx.begin();
....
// cast to get intern used PB instance
PersistenceBroker broker = tx.getBroker();
...
// build query
QueryByCriteria query = ...
// perform PB-query
Collection result = broker.getCollectionByQuery(query);
// use result
...

tx.commit();
```

```
...
```

Note: Don't close or commit the used broker instance, this will be done by the odmg-api.

7.2. How to use multiple Databases

For each database define a jdbc-connection-descriptor (../../docu/guides/repository.html#jdbc-connection-descriptor) same way as described in the FAQ (../../docu/faq.html#multipleDB) .

Now it is possible to

- access the databases one after another, by closing the current used `Database` instance and by open a new one.

  ```
  // get current used database instance
  Database database = ...;
  // close it
  database.close();
  // open a new one
  database = odmg.newDatabase();
  database.open("jcdAliasName#user#password", Database.OPEN_READ_WRITE);
  ...
  ```

  The `Database.close()` call close the current used `Database` instance, after this it is possible to open a new database instance.

- use multiple databases in parallel, by creating a separate `Implementation` and `Database` instance for each jdbc-connection-descriptor (../../docu/guides/repository.html#jdbc-connection-descriptor) defined in the mapping metadata.

  ```
  Implementation odmg_1 = OJB.getInstance();
  Database database_1 = odmg.newDatabase();
  database.open("db_1#user#password", Database.OPEN_READ_WRITE);

  Implementation odmg_2 = OJB.getInstance();
  Database database_2 = odmg.newDatabase();
  database.open("db_2#user#password", Database.OPEN_READ_WRITE);
  ```

  Now it's possible to use both databases in parallel.

Note:

OJB does not provide distributed transactions by itself. To use distributed transactions, OJB have to be integrated in an j2ee conform environment (../../docu/guides/deployment.html#j2ee-server) (or made work with an JTA/JTS implementation).