# Connection Handling

**by Armin Waibel, Martin Kalén**

## Table of contents

## 1. Introduction

In this section the connection handling within OJB is described. The connection management is implemented through two OJB interfaces:

- `org.apache.ojb.broker.accesslayer.ConnectionFactory`
- `org.apache.ojb.broker.accesslayer.ConnectionManagerIF`

## 2. ConnectionFactory

The `org.apache.ojb.broker.accesslayer.ConnectionFactory` interface implementation is a pluggable component (via the OJB.properties file - more about the OJB.properties file here) responsible for creation/lookup and release of connections.

```
public interface ConnectionFactory
{
    Connection lookupConnection(JdbcConnectionDescriptor jcd) throws
LookupException;

    void releaseConnection(JdbcConnectionDescriptor jcd, Connection con);

    void releaseAllResources();
}
```

To enable a specific *ConnectionFactory* implementation class in the *OJB.properties* file, set property *ConnectionFactoryClass*. Default:

```
ConnectionFactoryClass=org.apache.ojb.broker.accesslayer.ConnectionFactoryPooledImpl
```

OJB is shipped with several different implementation classes for use in different situations. The default implementation for example, will pool created Connection instances for increased performance (since instance creation normally makes a database server roundtrip and thus is costly).

To make it more easier to implement your own *ConnectionFactory* class, an abstract base class called `org.apache.ojb.broker.accesslayer.ConnectionFactoryAbstractImpl` exists, most shipped implementation classes inherit from this class.

> **Note:**
> All shipped implementations of *ConnectionFactory* with support for connection pooling will only use object pools for connections obtained directly from the JDBC DriverManager. If you are using a *DataSource* configuration, the JNDI DataSource is responsible for pooling.

### 2.1. ConnectionFactoryPooledImpl

A *ConnectionFactory* implementation using commons-pool to pool the Connection instances. On lookupConnection a Connection instance is borrowed from the object pool, and returned on the releaseConnection call. This implementation is used as *default* setting in the OJB.properties file.

This implementation allows a wide range of different settings, more info about the configuration properties can be found in the metadata repository *connection-pool* section.

### 2.2. ConnectionFactoryNotPooledImpl

Implementation that creates a new Connection instance on each lookupConnection call and closes (destroys) it on releaseConnection. All *connection-pool* settings are ignored by this implementation.

## 2.3. ConnectionFactoryManagedImpl

[@deprecated since OJB 1.0.4, now OJB automatic detect the running JTA-transaction and suppress critical method calls on the used connection]
Implementation specifically for use in *managed environments* like J2EE conformant application servers. In managed environments it is **mandatory** to use *DataSource* configuration, with Connection objects provided by the application server. OJB will not control Connection properties or transaction handling when using this implementation.

All *connection-pool* settings are ignored by this implementation.

## 2.4. ConnectionFactoryDBCPImpl

Implementation using commons-dbcp to pool the Connection instances. Since DBCP is using commons-pool internally, this implementation is very similar to *ConnectionFactoryPooledImpl*, but permits additional configuration for logging abandoned Connection instances (usable under development for detecting bad programming patterns).

This implementation allows a wide range of different settings, more info about the configuration properties can be found in the metadata repository *connection-pool* section.

# 3. ConnectionManager

The `org.apache.ojb.broker.accesslayer.ConnectionManagerIF` interface implementation is a pluggable component (via the OJB.properties file - more about the OJB.properties file here) responsible for managing the connection usage lifecycle and connection status (commit/rollback of connections).

```
public interface ConnectionManagerIF
{
    JdbcConnectionDescriptor getConnectionDescriptor();

    Platform getSupportedPlatform();

    boolean isAlive(Connection conn);

    Connection getConnection() throws LookupException;

    boolean isInLocalTransaction();

    void localBegin();

    void localCommit();

    void localRollback();

    void releaseConnection();

    void setBatchMode(boolean mode);

    boolean isBatchMode();

    void executeBatch();

    void executeBatchIfNecessary();

    void clearBatch();
}
```

The *ConnectionManager* is used by the *PersistenceBroker* to handle connection usage lifecycle.

# 4. Questions and Answers

## 4.1. How does OJB handle connection pooling?

OJB does connection pooling per default, except for datasources that are never pooled internally by OJB. Pooling of Connection instances when configuring OJB with DataSource lookup must be configured and performed in the DataSource provider.

The implementations of the `org.apache.ojb.broker.accesslayer.ConnectionFactory.java` interface are responsible for managing the connections in OJB. There are several implementations shipped with OJB called `org.apache.ojb.broker.accesslayer.ConnectionFactoryImpl.java`. There is, among others, a non-pooling implementation and an implementation using Commons DBCP API.

Configuration of the connection pooling is specified using the *connection-pool* element for each *jdbc-connection-descriptor*. The *connection-pool* element can be configured with properties for the specific *ConnectionFactory* implementation or JDBC driver used. For general information about the configuration, see the repository section or read the comments in repository.dtd.

## 4.2. Can I directly obtain a java.sql.Connection within OJB?

It is possible to obtain a Connection using the PB API and a `PersistenceBroker` instance. Example:

```
PersistenceBroker broker =
PersistenceBrokerFactory.createPersistenceBroker(myKey);
broker.beginTransaction();
// do something

Connection con = broker.serviceConnectionManager().getConnection();
// perform your connection action and do more
// close the created statement and result set

broker.commitTransaction();
broker.close();
```

After obtaining a Connection with `broker.serviceConnectionManager().getConnection()`, the connection can be used for any JDBC operations (except for transaction handling, more on this below). The user is responsible for cleanup of created Statement and ResultSet instances, so be sure to guard your call with a finally clause and close resources after use.

For read-only operations there is no need to start a PB transaction as in the example.

**Note:**
Do not commit any transactions on the Connection level, this should be left to OJB's PB API and will be performed automatically by calling PersistenceBroker commit-/abortTransaction methods.

**Note:**
Do not call `Connection.close()` on the obtained Connection, this should be left to OJB's *ConnectionFactory* and will be performed automatically when calling `broker.close()`.

If **no** transaction is running, it is possible to release a connection "by hand" after use by calling:

```
broker.serviceConnectionManager().releaseConnection();
```

This call performs cleanup operations on the used connection and pass the instance to the release method of *ConnectionFactory* (this will e.g. return the connection to pool or close it).

If you do not do any connection cleanup, the connection will at the latest be released when calling `broker.close()`.

Users who are interested in this section might also be interested in 'Is it possible to perform my own sql-queries in OJB?'.

## 4.3. When does OJB open/close a connection

This is dependent on the used OJB api. Generally OJB try to obtain a connection as late as possible and close (if a connection pool was used, OJB return the connection to the pool) the connection as soon as possible.

Using the PB-api the connection is obtained when `PersistenceBroker.beginTransaction()` was called or a query is executed.
On `PersistenceBroker.commitTransaction()` or `PersistenceBroker.abortTransaction()` call the connection was released. If no PB-tx is running, the connection will be released on `PersistenceBroker.close()` call.

Using the ODMG-api the connection is obtained when a query is executed or when the transaction commit. On leaving the commit method, the connection will be released.
All other top-level API should behave similar.