Deployment

by Thomas Mahler, Armin Waibel, Stephen Ting, Christophe Lombart, Lucy Zhao

1. Introduction

This section enumerates all things needed to deploy OJB in standalone or servlet based applications and j2ee-container.

2. Things needed for deploying OJB

2.1. 1. The OJB binary jar archive

You need a `db-ojb-<version>.jar` file containing the compiled OJB library.
This jar files contains all OJB code neccessary in production level environments. It does not contain any test code. It also does not contain any configuration data. You'll find this file in the lib directory of the binary distribution. If you are working with the source distribution you can assemble the binary jar archive By calling

```
ant jar
```

This ant task generates the binary jar to the dist directory.

2.2. 2. Configuration data

OJB needs two kinds of configuration data:

1. Configuration of the OJB runtime environment. This data is stored in a file named `OJB.properties` . Learn more about this file here.
2. Configuration of the MetaData layer. This data is stored in file named `repository.xml` (and several included files). Learn more about this file here.

Note:
These configuration files are read in through ClassLoader resource lookup and must therefore be placed on the classpath.

2.3. 3. Additional jar archives

OJB depends on several other jar archives. These jar files are shipped in the `db-ojb-<version>/lib` directory. These jar files are listed here.

Some of these jar files are only used during build-time and need not to be be deployed in runtime environments.
Apart from wasting disk space they do no harm. If you don't care you just take all jars from `db-ojb-<version>/lib`.
If you do care, here is the list of jars you can omit during runtime:

- `ant.jar`
- `antlr.debug.jar`
- `antlr_compiletime.jar`
- `junit.jar`
- `optional.jar`
- `xalan.jar`
- `ejb.jar`

- `servlet.jar`
- `jakarta-regexp-xxx.jar`
- `torque-xxx.jar`
- `velocity-xxx.jar`

2.4. 4. Don't forget the JDBC driver

The repository.xml defines JDBC Connections to your runtime databases. To use the declared JDBC drivers the respective jar archives must also be present in the classpath. Refer to the documentation of your databases.

In the following sections I will describe how to deploy these items for specific runtime environments.

3. Deployment in standalone applications

Deploying OJB for standalone applications is most simple. If you follow these four steps your application will be up in a few minutes.

1. Add `db-ojb-<version>.jar` to the classpath
2. place `OJB.properties` and `repository.xml` files on the classpath
3. Add the additional runtime jar archives to the classpath.
4. Add your JDBC drivers jar archive to the classpath.

4. Deployment in servlet based applications

Generally speaking the four steps described in the previous section have to be followed also in Servlet / JSP based environments.
The exact details may differ for your specific Servlet container, but the general concepts should be quite similar.

1. Deploy `db-ojb-<version>.jar` with your servlet applications WAR file.
   The WAR format specifies that application specific jars are to be placed in a directory `WEB-INF/lib`. Place `db-ojb-<version>.jar` to this directory.
2. Deploy `OJB.properties` and `repository.xml` with your servlet applications WAR file.
   The WAR format specifies that Servlet classes are to be placed in a directory `WEB-INF/classes`. The OJB configuration files have to be in this directory.
3. Add the additional runtime jar archives to `WEB-INF/lib` too.
4. Add your JDBC drivers jar archive to `WEB-INF/lib`.

By executing `ant war` you can generate a sample servlet application assembled to a valid WAR file. The resulting `ojb-servlet.war` file is written to the dist directory. You can deploy this WAR file to your servlet engine or unzip it to have a look at its directory structure.
you can also use the target `war` as a starting point for your own deployment scripts.

5. Deployment in EJB based applications

The above mentioned guidelines concerning jar files and placing of the OJB.properties and the repository.xml are valid for EJB environments as well. But apart from these basic steps you'll have to perform some additional configurations to integrate OJB into a managed environment.

The instructions to make OJB running within your application server should be similar for all server. So the following instructions for JBoss should be useful for all user. E.g. most `OJB.properties` file settings are the same for all application server.

There are some topics you should examine very carefully:

- **Connection handling:** Lookup DataSource from your AppServer, only these connections will be enlisted in running transactions

- **Caching:** Do you need distributed caching?
- **Locking:** Do you need distributed locking (when using odmg-api)?

5.1. Configure OJB for managed environments considering as JBoss example

The following steps describe how to configure OJB for managed environments and deploy on a ejb conform Application Server (JBoss) on the basis of the shipped ejb-examples. In managed environments OJB needs some specific properties.

5.1.1. 1. Adapt OJB.properties file

If the PB-api is the only persistence API being used (no ODMG nor JDO) and it is **only** being used in a managed environment, it is strongly recommended to use a special PersistenceBrokerFactory class, which enables PersistenceBroker instances to participate in the running JTA transaction (e.g. this makes PBStateListener proper work in managed environments and enables use of 'autoSync' property in ObjectCacheDefaultImpl):

```
PersistenceBrokerFactoryClass=org.apache.ojb.broker.core.PersistenceBrokerFactorySyncImpl
```

Note:

Don't use this setting in conjunction with any other top-level api (e.g. ODMG-api).

Your `OJB.properties` file need the following additional settings to work within managed environments (apply to **all** used api):

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

...
# set used application server TM access class
JTATransactionManagerClass=
org.apache.ojb.otm.transaction.factory.JBossTransactionManagerFactory
```

A specific *ConnectionFactory* implementation was used to by-pass all forbidden method calls in managed environments.

The *JTATransactionManagerClass* set the used implementation class for transaction manager lookup, necessary for for `javax.transaction.TransactionManager` lookup to participate in running *JTA transaction* via `javax.transaction.Synchronization` interface.

The ODMG-api needs some additional settings for use in managed environments (only needed when odmg-api was used):

```
...
# only needed for odmg-api
ImplementationClass=org.apache.ojb.odmg.ImplementationJTAImpl

...
# only needed for odmg-api
OJBTxManagerClass=org.apache.ojb.odmg.JTATxManager
```

The *ImplementationClass* specify the ODMG base class implementation. In managed environments a specific implementation is used, able to participate in *JTA transactions*.

The *OJBTxManagerClass* specify the used OJBTxManager implementation to manage the transaction synchronization in managed enviroments.

Note:

Currently OJB integrate in managed environments via `javax.transaction.Synchronization` interface. When the *JCA adapter* is finished (work in progress) integration will be more smooth.

5.1.2. 2. Declare datasource in the repository (repository_database) file and do additional configuration

Do only use `DataSource` from the application server to connect to your database (Local used connections do not participate in JTA transaction).

> **Note:**
>
> We strongly recommend to use JBoss 3.2.2 or higher of the 3.x series of JBoss. With earlier versions of 3.x we got Statement/Connection resource problems when running some ejb stress tests. As workaround we introduce a jboss specific attribute *eager-release* for version before 3.2.2, but it seems that this attribute can cause side-effects. Again, this problem seems to be fixed in 3.2.2.

Define OJB to use a DataSource:

```
<!-- Datasource example -->
<jdbc-connection-descriptor
    jcd-alias="default"
    default-connection="true"
    platform="Sapdb"
    jdbc-level="2.0"
    jndi-datasource-name="java:DefaultDS"
    username="sa"
    password=""
    eager-release="false"
    batch-mode="false"
    useAutoCommit="0"
    ignoreAutoCommitExceptions="false"
>
    <object-cache class="org.apache.ojb.broker.cache.ObjectCacheDefaultImpl">
        <attribute attribute-name="timeout" attribute-value="900"/>
        <attribute attribute-name="autoSync" attribute-value="true"/>
     </object-cache>

    <sequence-manager className="org.apache.ojb.broker.util.sequence.SequenceManagerNextValImpl">
    </sequence-manager>

</jdbc-connection-descriptor>
```

The attribute `useAutoCommit="0"` is mandatory in managed environments, because it's in most cases not allowed to change autoCommit state.

> **Note:**
>
> In managed environments you can't use the default sequence manager (SequenceManagerHighLowImpl) of OJB. For alternative sequence manager implemetation see here.

5.1.3. [2b. How to deploy ojb test hsqldb database to jboss]

If you use hsql database for testing you can easy setup the DB on jboss. After creating the database in OJB test directory with `ant prepare-testdb`, take the generated `.../target/test/OJB.script` file and rename it to `default.script`. Then replace the jboss default.script file in `.../jboss-3.x.y/server/default/db/hypersonic` with this file.

5.1.4. 3. Include all OJB configuration files in classpath

Include the all needed OJB configuration files in your classpath:

- OJB.properties
- repository.dtd
- repository.xml
- repository_internal.xml
- repository_database.xml,

- repository_ejb.xml (if you want to run the ejb examples)

To deploy the ejb-examples beans we include all configuration files in a ejb jar file - more info about this see below.

The repository.xml for the ejb-example beans look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This is a sample metadata repository for the ObjectBridge
System. Use this file as a template for building your own
mappings-->

<!-- defining entities for include-files -->
<!DOCTYPE descriptor-repository SYSTEM "repository.dtd" [
<!ENTITY database SYSTEM "repository_database.xml">
<!ENTITY internal SYSTEM "repository_internal.xml">
<!ENTITY ejb SYSTEM "repository_ejb.xml">
]>


<descriptor-repository version="1.0"
            isolation-level="read-uncommitted">

    <!-- include all used database connections -->
                            &database;

    <!-- include ojb internal mappings here -->
                            &internal;

    <!-- include mappings for the EJB-examples -->
                            &ejb;

</descriptor-repository>
```

5.1.5. 4. Enclose all libraries OJB depend on

In most cases it is recommended to include all libraries OJB depend on in the application .ear/.sar or ejb .jar file to make OJB run and (re-)deployable. Here are the libraries needed to make the ojb sample session beans run on JBoss:

- The jakarta commons libraries files (all commons-xxx.jar) from OJB /lib directory
- The antlr jar file (antlr-xxx.jar) from OJB /lib directory
- jakarta-regexp-xxx.jar from OJB /lib directory
- [jakarta turbine jcs.jar from OJB /lib directory, only if ObjectCacheJCSImpl was used]

(This was tested with jboss 3.2.2)

5.1.6. 5. Take care of caching

Very important thing is cache synchronization with the database. When using the ODMG-api or PB-api (with special PBF (see 1.) setting) it's possible to use all `ObjectCache` implementations as long as OJB doesn't run in a clustered mode. When the `ObjectCacheDefaultImpl` cache implementation was used it's recommended to enable the *autoSync* mode.
In clustered environments (OJB run on different AppServer nodes) you need a distributed ObjectCache or you should use a local/empty cache like

```
ObjectCacheClass=org.apache.ojb.broker.cache.ObjectCachePerBrokerImpl
```

or

```
ObjectCacheClass=org.apache.ojb.broker.cache.ObjectCacheEmptyImpl
```

The cache is pluggable, so you can write your own ObjectCache implementation to accomplish your expectations.

More info you can find in clustering and ObjectCache topic.

5.1.7. 6. Take care of locking

If the used api supports *Object Locking* (e.g. ODMG-api, PB-api does not), in clustered environments (OJB run on different AppServer nodes) a distributed lock management is mandatory.

5.1.8. 7. Put all together

Now put all files together. We keep the examples as simple as possible, thus we deploy only a ejb .jar file. Below you can find a short instruction how to pack an ejb application .ear file including OJB.

Generate the ejb-examples described below or build your own ejb .jar file including all beans, ejb-jar.xml and appServer dependend files. Then add all OJB configuration files, the db-ojb jar file and all libraries OJB depends on into this ejb .jar file. The structure of the ejb .jar file should now look like this:

```
/OJB.properties
/repository.dtd
/repository.xml
/all used repository-XYZ.xml
/META-INF
.../Manifest.mf
.../ejb-jar.xml
.../jboss.xml

/all ejb classes

/db-ojb-1.X.jar
/all used libraries
```

5.1.9. 7b. Example: Deployable jar

For example the jar-file used to test the *ejb-examples* shipped with OJB, base on the *db-ojb-XY-beans.jar* file. This jar was created when the *ejb-examples* target was called.

The generated jar contains only the ejb-classes and the deployment-descriptor. We have to add additional jars (all libraries used by OJB) and files (all configuration files) to make it deployable. The deployable *db-ojb-XY-beans.jar* should look like this:

```
/OJB.properties
/repository.dtd
/repository.xml
/repository_database.xml
/repository_ejb.xml
/repository_internal.xml
/META-INF
.../Manifest.mf
.../ejb-jar.xml
.../jboss.xml

/org
.../apache (all ejb classes)

/db-ojb-1.X.jar

/antlr-XXX.jar
/commons-beanutils-XXX.jar
/commons-collections-XXX.jar
/commons-dbcp-XXX.jar
/commons-lanf-XXX.jar
/commons-logging-XXX.jar
/commons-pool-XXX.jar
/jakarta-regexp-XXX.jar
```

Please pay attention on the configuration settings to make OJB work in managed environments (especially the OJB.properties settings).

> **Note:**
> This example isn't a real world production example. Normally you will setup one or more enterprise archive files (.ear files) to bundle one or more complete J2EE (web) applications. More about how to build an *J2EE application* using OJB see here.

The described example should be re-deployable/hot-deployable in JBoss.
If you will get any problems, please let me know. All suggestions are welcome!

5.1.10. 8. How to access OJB API?

In managed environments it is possible to access OJB in same way used in non-managed environments:

```
// PB-api
PersistenceBroker broker = PersistenceBrokerFactory.create...;

//ODMG-api
Implementation odmg = OJB.getInstance();
```

But it is also possible to bind OJB api access classes to JNDI and lookup the the api entry classes via JNDI.

5.1.11. 9. OJB logging within JBoss

Jboss use log4j as standard logging api.
In summary, to use log4j logging with OJB within jBoss:
1) in OJB.properties set

```
LoggerClass=org.apache.ojb.broker.util.logging.Log4jLoggerImpl
```

There is no need for a separate log4j.properties file of OJB-specific log4j settings (in fact the OJB.properties setting LoggerConfigFile is ignored). Instead, the jBoss log4j configuration file must be used:

2) in JBOSS_HOME/server/default/conf/log4j.xml,
define appenders and add categories to add or filter logging of desired OJB packages, following the numerous examples in that file. For example,

```
<category name="org.apache.ojb">
    <priority value="DEBUG" />
    <appender-ref ref="CONSOLE"/>
    <appender-ref ref="FILE"/>
</category>

<category name="org.apache.ojb.broker.metadata.RepositoryXmlHandler">
    <priority value="ERROR" />
    <appender-ref ref="CONSOLE"/>
    <appender-ref ref="FILE"/>
</category>
```

5.2. Build example beans

5.2.1. Generate the sample session beans

The OJB source distribution was shipped with a bunch of sample session beans and client classes for testing. Please recognize that we don't say that these examples show "best practices" of using OJB within enterprise java beans - it's only one way to make it work.

The source code of the sample beans is stored in directory

`[db-ojb]/src/ejb/org/apache/ojb/ejb`
To generate the sample beans call

```
ant ejb-examples
```

This ant target copies the bean sources to `[db-ojb]/target/srcejb` generates all needed bean classes and deployment descriptor (by using xdoclet) to the same directory, compiles the sources and build an ejb .jar file called `[db-ojb]/dist/db-ojb-XXX-beans.jar`. Test clients for the generated beans included in the `[db-ojb]/dist/db-ojb-XXX-client.jar`.

To run xdoclet properly the following xdoclet jar files needed in `[db-ojb]/lib` directory (xdoclet version 1.2xx or higher):

```
xdoclet-xxx.jar
xdoclet-ejb-module-xxx.jar
xdoclet-jboss-module-xxx.jar
xdoclet-jmx-module-xxx.jar
xdoclet-web-module-xxx.jar
xdoclet-xjavadoc-module-xxx.jar
```

If you using a different application server than JBoss, you have to modifiy the *xdoclet* ant target in `[db-ojb]/build-ejb-examples.xml` to force xdoclet to generate the appServer specific files. See xdoclet documentation for further information.

5.2.2. How to run test clients for PB / ODMG - api

If the "extended ejb .jar" file was successfully deployed we need a test client to invoke the ejb-examples. As said above, the *ejb-examples* target generates a test client jar too. It's called `[db-ojb]/dist/db-ojb-XXX-client.jar` and contains junit based test clients for the PB-/ODMG-api.
The main test classes are:

- org.apache.ojb.ejb.AllODMGTests
- org.apache.ojb.ejb.AllPBTests

OJB provide an ant target to run the client side bean tests. Include all needed appServer libraries in `[db-ojb]/lib` (e.g. for JBoss jbossall-client.jar do the job, beside the "j2ee jars"). To run the PB-api test clients (access running JBoss server with default settings) call

```
ant ejb-examples-run -Dclient.class=org.apache.ojb.ejb.AllPBTests
```

To run the test clients on an arbitrary appServer pass the JNDI properties for naming context initalisation too, e.g.

- -Djava.naming.factory.initial="org.jnp.interfaces.NamingContextFactory"
- -Djava.naming.provider.url="jnp://localhost:1099"
- -Djava.naming.factory.url.pkgs="org.jboss.naming:org.jnp.interfaces"

Then the target call may looks like

```
ant ejb-examples-run -Dclient.class=org.apache.ojb.ejb.AllPBTests
 -Djava.naming.factory.initial="org.jnp.interfaces.NamingContextFactory"
  -Djava.naming.provider.url="jnp://localhost:1099"
   -Djava.naming.factory.url.pkgs="org.jboss.naming:org.jnp.interfaces"
```

5.3. Packing an .ear file

Here is an example of the .ear package structure. It is redeployable without having to restart JBoss.

5.3.1. The Package Structure

The package structure of the *.ear* file should look like:

```
/ejb.jar/
...EJBs
...META-INF/
......ejb-jar.xml
......jboss.xml
......MANIFEST.MF

/web-app.war/
...JSP
...WEB-INF/
......web.xml

/META-INF/
...application.xml
/ojb.jar
/[ojb required runtime jar]

/OJB.properties
/repository.dtd
/respository_internal.xml
/repository.xml
/repository_database1.xml
/repository_app1.xml
/repository_database2.xml
/repository_app2.xml
```

5.3.2. Make OJB API Resources available

There are two approaches to use OJB api in the ejb.jar file:

1. To create a Manifest.mf file with classpath attribute that include all the runtime jar required by OJB (Very important to include all required jar). The sample below works fine:

```
Class-Path: db-ojb-1.0.rc6.jar antlr-2.7.3.jar commons-beanutils.jar
commons-collections.jar commons-dbcp-1.1.jar commons-lang-2.0.jar
commons-logging.jar commons-pool-1.1.jar
jakarta-regexp-1.3.jar
```

Note:

If you to include the jar file under a directory of the ear file, says `/lib/db-ojb-1.0.rc6.jar` and etc. At the classpath attribute it will be something like: `Class-Path: ./lib/db-ojb-1.0.rc6.jar and etc` (The "." in front is important)

2. To add the required jar file as a "java" element in the application.xml file:

```
<module>
    <java>antlr-2.7.3.jar</java>
</module>
<module>
    <java>commons-beanutils.jar</java>
</module>
<module>
    <java>commons-collections.jar</java>
</module>
<module>
    <java>commons-dbcp-1.1.jar</java>
</module>
<module>
    <java>commons-lang-2.0.jar</java>
</module>
<module>
    <java>commons-logging.jar</java>
</module>
<module>
    <java>commons-pool-1.1.jar</java>
</module>
```

```
<module>
    <java>db-ojb-1.0.rc6.jar</java>
</module>
```

> **Note:**
> To use this approach, all the library had to be in the root of the ear.

(This was tested on Jboss 3.2.3)

5.4. Make OJB accessible via JNDI

Current bean examples do directly use OJB main classes, but it's also possible to make OJB accessible via JNDI and use a JNDI-lookup to access OJB api's in your beans.
To make the OJB api's accessible via JNDI, you can bind them to JNDI. How to do this depends on the used environment. The main classes/method to bind are:

- Class `org.apache.ojb.broker.core.PersistenceBrokerFactoryFactory` for PB-api. Make method `PersistenceBrokerFactoryFactory.instance()` accessible.
- Class `org.apache.ojb.odmg.OJB` for ODMG-api. Make method `OJB.getInstance()` accessible.

5.4.1. JBoss

In JBoss you can use `mbean` classes. `org.apache.ojb.jboss.PBFactory` and `org.apache.ojb.jboss.ODMGFactory` are mbean implementations bind PB-api and ODMG-api main classes to JNDI.
Let JBoss know about the new mbeans, so declare them in a `jboss-service.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>

<server>
    <mbean code="org.apache.ojb.jboss.PBFactory"
        name="DefaultDomain:service=PBAPI,name=ojb/PBAPI">
        <depends>jboss.jca:service=RARDeployer</depends>
        <attribute name="JndiName">ojb/PBAPI</attribute>
    </mbean>

    <mbean code="org.apache.ojb.jboss.ODMGFactory"
    name="DefaultDomain:service=ODMG,name=ojb/defaultODMG">
        <depends>jboss.jca:service=RARDeployer</depends>
        <attribute name="JndiName">ojb/defaultODMG</attribute>
    </mbean>
</server>
```

5.4.2. Other Application Server

In other application server you can do similar steps to bind OJB main api classes to JNDI. For example in Weblogic you can use *startup class* implementation (see below).

5.5. Instructions for Weblogic

1. Add the OJB jar files and depedencies into the Weblogic classpath

2. As usual create the connection pool and the datasource.

3. Prepare the OJB.properties file. Should be similar to jboss. Expect the following entry:

```
...
# Weblogic Transaction Manager Factory
JTATransactionManagerClass=
org.apache.ojb.broker.transaction.tm.WeblogicTransactionManagerFactory
```

4. Modify the connection information in the repository.xml (specify the datasource name). SequenceManager implementation depends on the used DB, more info see here:

```
<jdbc-connection-descriptor
jcd-alias="default"
default-connection="true"
platform="Sapdb"
jdbc-level="2.0"
jndi-datasource-name="datasource_demodb"
eager-release="false"
batch-mode="false"
useAutoCommit="0"
ignoreAutoCommitExceptions="false"
>

<sequence-manager
className="org.apache.ojb.broker.util.sequence.SequenceManagerNextValImpl">
<attribute attribute-name="grabSize" attribute-value="20"/>
</sequence-manager>
</jdbc-connection-descriptor>
```

Note:

The following step is only neccessary if you want to bind OJB main api classes to JNDI.

[5.] Compile the following classes (see at the end of this section) and add them to the weblogic classpath. This allows to access the PB-api via JNDI lookup. Register via the weblogic console the startup class (see `OjbPbStartup` class below). The JNDI name and the OJB.properties file path can be specified as parameters in this startup class.

To use the ODMG-api you have to write a similar startup class. This shouldn't be too complicated. Take a look in `org.apache.ojb.jboss` package (dir `src/connector/main`). Here you could find the jboss mbeans. All you have to do is bound a similar class to JNDI in weblogic.
Implement `ODMGJ2EEFactory` Interface in your class bound this class to JNDI (in the ejb-examples the beans try to lookup the `Implementation` instance via `"java:/ojb/defaultODMG"`). Your ODMGFactory class should implement this method

```
    public Implementation getInstance()
    {
        return OJBJ2EE_2.getInstance();
    }
```

Write a session bean similar to those provided for the JBOSS samples. It is also possible to use the ejb-example beans (doing minor modifications when the JNDI lookup should be used).

Webolgic startup class
Write an OJB startup class to make OJB accessible via JNDI can look like (I couldn't test this sample class, so don't know if it will work ;-)):

```
package org.apache.ojb.weblogic;

import javax.naming.*;

import org.apache.ojb.broker.core.PersistenceBrokerFactoryFactory;
import org.apache.ojb.broker.core.PersistenceBrokerFactoryIF;

import weblogic.common.T3ServicesDef;
import weblogic.common.T3StartupDef;
import java.util.Hashtable;

/**
* This startup class created and binds an instance of a
* PersistenceBrokerFactoryIF into JNDI.
*/
public class OjbPbStartup
        implements T3StartupDef, OjbPbFactory, Serializable
{
    private String defaultPropsFile = "org/apache/ojb/weblogic/OJB.properties";

    public void setServices(T3ServicesDef services)
    {
    }

    public PersistenceBrokerFactoryIF getInstance()
    {
        return PersistenceBrokerFactoryFactory.instance();
    }

    public String startup(String name, Hashtable args)
            throws Exception
    {

        try
        {
            String jndiName = (String) args.get("jndiname");
            if(jndiName == null || jndiName.length() == 0)
                jndiName = OjbPbFactory.DEFAULT_JNDI_NAME;

            String propsFile = (String) args.get("propsfile");
            if(propsFile == null || propsFile.length() == 0)
            {
                System.setProperty("OJB.properties", defaultPropsFile);
            }
            else
            {
                System.setProperty("OJB.properties", propsFile);
            }

            InitialContext ctx = new InitialContext();
            bind(ctx, jndiName, this);

            // return a message for logging
            return "Bound OJB PersistenceBrokerFactoryIF to " + jndiName;
        }
        catch(Exception e)
        {
            e.printStackTrace();
            // return a message for logging
            return "Startup Class error: impossible to bind OJB PB factory";
        }
    }

    private void bind(Context ctx, String name, Object val)
            throws NamingException
    {
        Name n;
        for(n = ctx.getNameParser("").parse(name); n.size() > 1; n = n.getSuffix(1))
        {
            String ctxName = n.get(0);
            try
            {
                ctx = (Context) ctx.lookup(ctxName);
            }
            catch(NameNotFoundException namenotfoundexception)
            {
                ctx = ctx.createSubcontext(ctxName);
            }
        }
        ctx.bind(n.get(0), val);
    }
}
```

The used OjbPbFactory interface:

```
package org.apache.ojb.weblogic;

import org.apache.ojb.broker.core.PersistenceBrokerFactoryIF;

public interface OjbPbFactory
{
    public static String DEFAULT_JNDI_NAME = "PBFactory";
    public PersistenceBrokerFactoryIF getInstance();
}
```