Classloading issues in multiple PersistenceManagerFactories configurations

By Arnaud Thiefaine

February 2006

Document version: v100
Last revised: 06 February 2006

Copyright © 2000-2006 by Xcalia All rights reserved.

This publication pertains to Xcalia and any subsequent release until otherwise indicated in new edition or technical notes. Information in this document is subject to change without notice. The software described herein is furnished under a license agreement, and it may be used or copied only in accordance with the terms of the agreement.
All product and company names used herein may be trademarks or registered trademarks of their respective companies.

Xcalia
71, Rue Desnouettes
75015 PARIS
FRANCE

Introduction

This article provides different ways to integrate more than one PersistenceManagerFactory (PMF) in the same JVM environment (such as an application server).
Our basic example consists in integrating two web applications (each with its own PMF) in a same application server. We call these web applications App A and App B (with respectively PMF A and PMF B).

Classloading considerations

A classloader hierarchy can contain several levels. For instance, in an application server such as JONaS, the following levels can be found (from parents to children):

- system classloader: loads the classes from the JVM
- application server classloader: loads the common libraries of the application server
- application shared classloader: loads the libraries required by all the applications deployed in the application server; for instance RAR files used by all the applications will be placed at this classloader level
- application specific classloaders: loads the libraries required by a specific application (for instance packaged in an EAR file); the hierarchy of these classloaders is as follows:
    - EAR classloader: this classloader is responsible for loading all the classes of the EAR application; for instance, if there are RAR files, they will be loaded at this level
    - RAR classloader, if there is a RAR file embedded in the EAR
    - Web classloader: loads the classes of a web application, packaged in a WAR file

According to the chosen application server, different classloading strategies can be used such as:

- parent first: the classes are first sought at the highest level in the classloader hierarchy. This is the default and most common configuration.
- parent last: the classes are first sought at the lowest level in the classloader hierarchy

An article on the Server Side provides guidance to implement alternative classloading strategies. It explains how to use a fine filtering of classes in classloaders, or how to reduce the libraries memory footprint. It can be found at the following address:
http://www.theserverside.com/articles/article.tss?l=AdvancedClassLoading

App A and App B are distinct web applications, so their classes are not loaded with the same classloader. These distinct classloaders are at the same level in the classloader hierarchy: the application level. Since they are siblings, one doesn't know the classes of the other (and inversely).

Through the different web applications (App A and App B) are at the same classloader level, they have different classloaders, and their classes don't know of each other.

Multiple PMF instances and metadata registration

Let's now look at the heart of the problem. It is related to the metadata registration mechanism.

Metadata registration mechanism

The class `javax.jdo.spi.JDOImplHelper`, contained in the JDO API, provides a registry of metadata by class and its methods avoid the use of reflection at runtime. Persistence-capable classes register metadata with this class during class initialization (when they are loaded).

The class `JDOImplHelper` can notify some listeners when metadata are registered. In Xcalia Core, the PMF instances listen to the `JDOImplHelper`. Each PMF is notified with a `RegisterClassEvent` that contains the metadata of a given persistence-capable class.

Thus, when a new persistence-capable class is loaded, the `JDOImplHelper` is notified, and then each PMF receives the metadata about this class.

It should be noted that metadata carried in `RegisterClassEvents` are conformant to the JDO specification. However, Xcalia Core persistence-capable class metadata are richer than those of the JDO specification. When a PMF receives JDO metadata, it tries to load this richer metadata by looking for a .jdo file. By default this file is named package.jdo, but it can be changed with the `lido.metadataFileNames` property of the PMF. If the file doesn't contain metadata compatible with those of the `RegisterClassEvent`, a metadata not found error occurs.

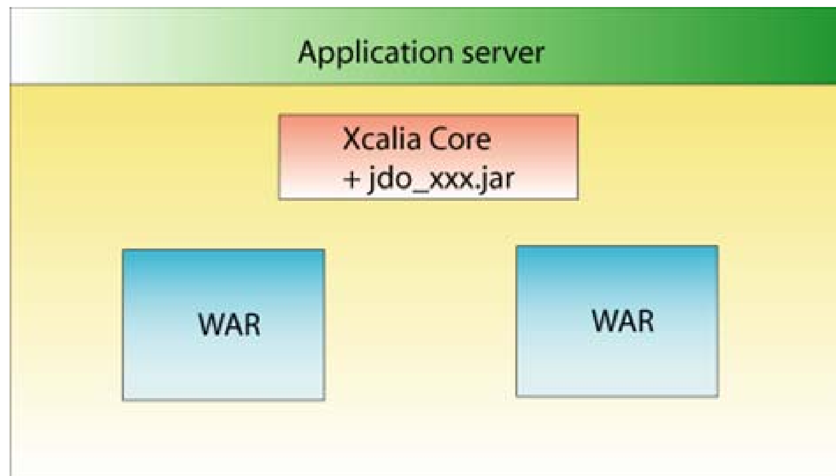
PMF notification issues

Let's follow on our example: PMF A and PMF B are registered to the same `JDOImplHelper`.
Each PMF runs with a distinct classloader (each in its own web application for instance).
When PMF A registers a persistence-capable class metadata, the `JDOImplHelper` sends it back to its listeners.
PMF A receives the metadata about a class that is known by its classloader, so no problem occurs.
PMF B receives the metadata about a class (from App A classloader) unknown by its classloader, so a problem occurs.

In fact, a problem occurs when a PMF receives an event which contains metadata about a class that its classloader can't retrieve. The following errors can be thrown: metadata not found, class not found, class cast exception.

Here is an example of an architecture where PMF notification issues occur. There are 3 classloaders involved:

- one for the `JDOImplHelper`, which is present in the libraries loaded by the application server (application shared level); please note that these libraries can be deployed as a RAR file
- one for each PMF instances (in the WAR files), they are children of the first classloader

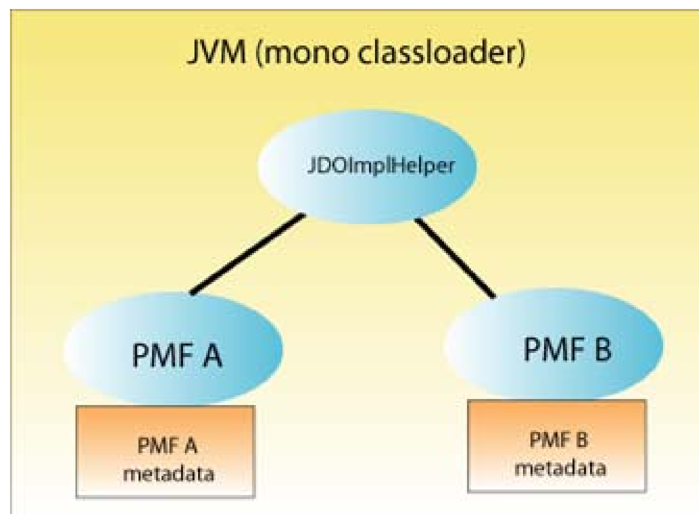


This architecture doesn't work and causes a "`No metadata has been registered for the class <className>`" error.

When PMF instances with different classloaders listen to the same `JDOImplHelper`, these PMF instances potentially listen to metadata that are incompatible with their classloader. This can produce errors.

Single classloader configurations

In this section, we treat the cases where multiple PMF instances share a same `JDOImplHelper`, but run with the same classloader.
In our examples there are two PMF instances, each with one metadata file that describes full metadata about persistence-capable classes.



All persistence-capable classes of the two PMF instances are loadable because they are in the same classloader.

```
PersistenceManagerFactory pmf1 = …  // PMF1 registers as a listener of the
// JDOImplHelper
```

```
Employee emp = new Employee();   // Employee notifies the JDOImplHelper,
// which in turn notifies PMF1 to load metadata of Employee1.

PersistenceManagerFactory pmf2 = …  // PMF1 registers as a listener of the
// JDOImplHelper, and receives notification of the Employee registered
persistent class, and so tries to load its metadata again.
```

However, problems can occur from conflicting class definitions in the metadata files.

Default metadata file names

By default, the two PMF instances use metadata loaded according to the JDO specification, i.e., tries to locate at the metadata file in a series of locations, some of which depend on the fully qualified class name. In this case, it will find a package.jdo in the class directory, and the needed metadata will be loaded.

So this case works.

Custom metadata file names

The two PMF instances use metadata file specified with the `lido.metadatafileNames` property. PMF A uses metadataA.jdo that describes the persistence-capable class A, and PMF B use metadataB.jdo that describes the persistence-capable class B.

When PMF B receives an event from the class A, it looks for its metadata in the file metadataB.jdo. And it does not find it so it produces a "Metadata not registered …" error;

However, there is a workaround designed for this case to work. It consists in providing the name of all the possibly required metadata file names in the `lido.metadataFileNames` property. In our example, it is defined as follows:

```
lido.metadataFileNames=metadataA.jdo; metadataB.jdo
```

Multiple classloader architectural solutions

We propose different ways to integrate multiple Xcalia Core PMF instances in a same runtime environment (JVM, application server, etc.).

The class `JDOImplHelper` is present in the library jdo_xxx.jar, where xxx represents the version number. This class conforms to the Singleton design pattern. However, it can be present multiple times in a runtime environment, as long as there is enough classloaders that have jdo_xxx.jar accessible.

That means that it is possible to use one `JDOImplHelper` for each needed PMF, which avoids the metadata compatibility issue. The first architectural solution uses this technique.

Another possibility to avoid metadata compatibility issue is to ignore the registering of the PMF instances to the `JDOImplHelper`. This is discussed in the third architectural solution.

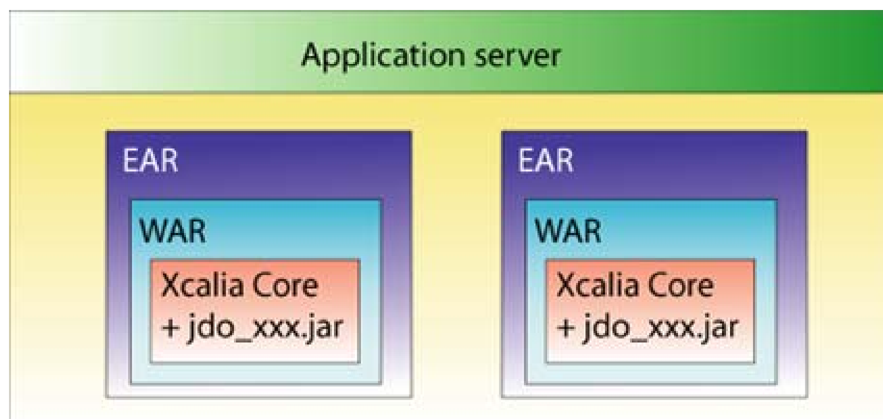
Solution 1: packaging Xcalia Core with each web application

In this kind of solution, Xcalia Core (which contains jdo_xxx.jar) is packaged with each web application.
Each application is provided in an EAR file, which contains the jars of the application and the jars of Xcalia Core.

There are two variants to this approach.

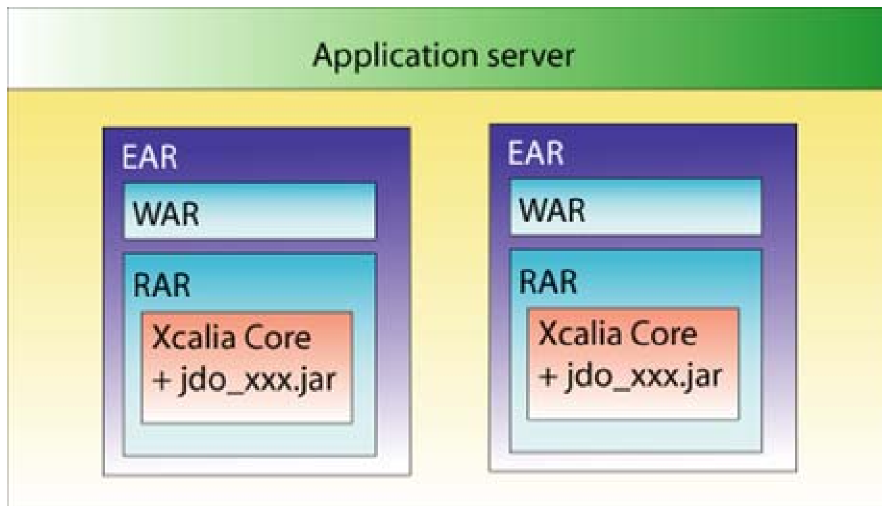
First approach

The EAR files contain a WAR file that contains the web application and the jars of Xcalia Core. The PMF is accessed with a `JDOHelper`.



Second approach

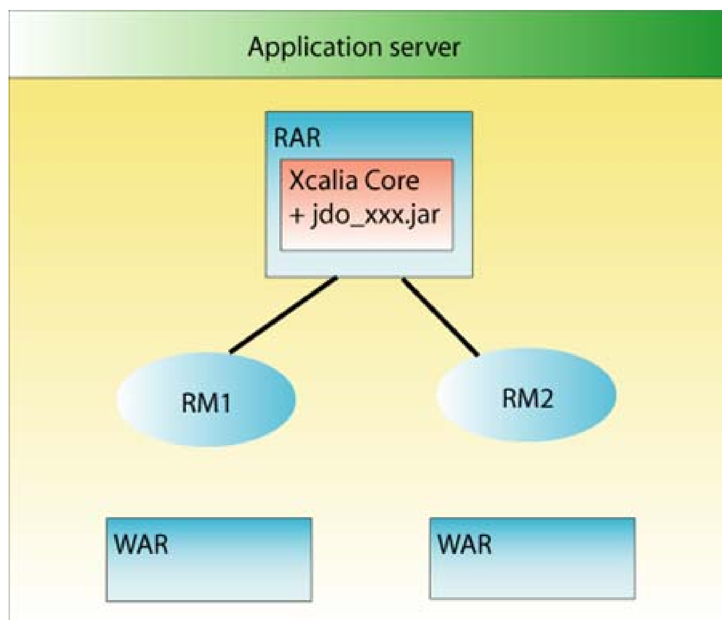
The EAR files contain:
- the WAR of the web application
- a RAR connector that contains the jars of Xcalia Core. The PMF is accessed through JNDI as a `ConnectionFactory`



These approaches work well because each web application embeds its own `JDOImplHelper`.

Solution 2: ignoring RegisterClassEvents

An architecture similar to that of the second solution is implemented.
The variant is that the WAR files don't embed jdo_xxx.jar.



As a consequence, the different PMF instances share the same `JDOImplHelper` (present in the RAR file), and that can cause metadata compatibility issues.

In order to prevent this, it is chosen that the PMF instances ignore metadata registration notifications. This is done by setting the following property in lido.properties:

```
lido.jdo.registerClassEvents = disabled
```

This solution however brings a significant hindrance: some metadata about persistence-capable classes are lost at runtime, including those concerning the current PMF. In this mode, one must specify the persistent class explicitly to the Xcalia Core APIs in order to ensure metadata loading. For instance:

```
Employee emp = new Employee ();      // Employee class notifies the
// JDOImplHelper, but the PMF is no longer notified on such a loading (and
// thus doesn't known about its metadata)

Query q = pm.newQuery(Employee.class, filter);
result = q.execute();   // Xcalia Core load Employee's metadata
```

Information on class hierarchy may be lost. For instance, persistence metadata provide information on class hierarchy in a top-down direction. For a given class, metadata describe its inheriting classes. If that information is missing, queries about a given class won't retrieve instances from its subtypes.

```
Manager mgr = new Manager ();      // Manager class notifies the
// JDOImplHelper, but the PMF is still no longer notified on such a loading
// (and thus doesn't known about its metadata)

Query q = pm.newQuery(Employee.class, filter);
result = q.execute ();    // Xcalia Core only query the datastore for
// employees

pm.newObjectIdInstance(Manager.class, "12"); // Xcalia Core loads metadata
// for the Manager persistent class

Query q = pm.newQuery(Employee.class, filter);
result = q.execute ();   // Xcalia Core only query the datastore for both
// employees and managers
```

It should be noted that subclass issue may also occur in enabled mode: it can happen if the loading of the class Employee didn't imply the loading of Employee and Manger metadata (i.e. if their metadata are not defined in the same file).

Resource manager reference release issue

When an application module (either EJB-JAR or WAR) is undeployed, the connector keeps references to the classes of this module. Each time the module is redeployed, the connector loads new references to the applicative classes, thus causing memory leaks.

Conclusion

In this article, we discussed about persistence metadata retrieval when an enterprise application deals with multiple PMF instances.
We proposed several architectural solutions in order to manage multiple PMF instance and to avoid metadata compatibility issues.
Some of the solutions bring drawbacks, such has loss of information contained in metadata, or memory leaks.

It seems that the best approaches consist in packaging a PMF instance and a `JDOImplHelper` in the same classloader. When one PMF is associated with one `JDOImplHelper`, metadata compatibility issues are avoided.

The issues encountered result from a more general problem. The applicative modules contain applications classes that go up to a shared, technical connector and its classloader. However, the technical connector deals with applicative classes by keeping references to them, which causes problems. Technical components and application components should be clearly separated.

For the moment, a neat separation can be achieved by putting the code that manages applicative objects (for example the `JDOImplHelper`) in the applicative module, and not in a shared, technical space.

It should also be noted that the metadata loading issues are not likely to be encountered with standards such as SDO (Service Data Objects) or EJB3.0.
With SDO, data are not represented by classes, but in a standardized manner, so there are no Classloading problems. Xcalia Core implements the SDO specification as an alternative to JDO.
With EJB3.0, the metadata are tightly-coupled with the persistent classes, so metadata retrieval issues don't occur.

The incoming product Xcalia Intermediation Server will both a neat separation between technical and applicative aspects related to PMF instances, and a solution to classloading issues, since persistence-capable classes won't be present server-side.