



Invoking a JCA service through Xcalia Core for Services

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 sample makes use of Xcalia Core for Services in a service-oriented architecture (SOA) deployed in an application container. This architecture is separated into two tiers: the service provider tier and the presentation (service consumer) tier.

The service tier consists in a JCA Connector. It hides some customer data.

The presentation tier is implemented as a Java servlet, which makes use of the JDO API in order to transparently access some data hidden in the service tier.

Xcalia Core for Services provides the magic that enables the mapping of POJO classes (in the presentation tier) to the JCA Connector (in the service tier).

The technologies, standards and architectures covered in this sample are as follows: Xcalia Core for Services, Xcalia Intermediation Platform, JDO, servlets, JCA, Jalist, SOA.

Getting started

Prerequisites

In order to run this sample, you first need to have Xcalia Core (version 4.1 or greater), and Xcalia Intermediation Platform (version 4.0 or greater) properly installed and configured. You also must have a valid licence enabling the service mapping. As the default licence coming Xcalia Core just covers the object-relational mapper, you must request an evaluation licence of the service mapper at sales@xcalia.com.

An Ant installation (version 1.6) is also required but there is one packaged with Xcalia Core.

Installing the sample

Unpack the sample in a directory of your choice. Let's call it `${sample.root.dir}`.

You need to provide a valid Xcalia Core license file in the `${sample.root.dir}/etc/` directory.

You also must modify the `build.properties` file and indicate the path to your Xcalia Core and Xcalia Intermediation Platform root directories. Here is an example of a `build.properties` file:

```
# the root dir where Xcalia Core is installed
xic.home=c:/dev/xcalia/core

# the root dir where XIP is installed
xip.home=c:/dev/xcalia/platform
```

The sample needs to create a small data source on your hard drive. The path of the data source can be specified in `${sample.root.dir}/etc/jalist.properties`, by changing the value of the `dbFilesPaths` property (you must use a full file name with an absolute path). If you keep the default value a file `customer.jalist` will be added in the `c:/` directory.

Deploying the sample

With a command interpreter, go to `${sample.root.dir}`. Then, enter the following command:

```
ant
```

Alternately, you can create a new project in Eclipse, import your sample directory and run Ant from the Eclipse Ant view.

This Ant script has the following effects: a RAR (Resource ARchive) file and a WAR (Web ARchive) file are deployed in the Xcalia Intermediation Platform.

Running the sample

The first step consists in launching the Xcalia Intermediation Platform.

Go into the `bin/` directory of your Xcalia Intermediation Platform installation and run the following command:

```
start-xip
```

After a few seconds, the platform is started.

NB: if XIP does not start, please check the system variable `XCALIA_CORE_HOME` points to your Xcalia Core installation directory.

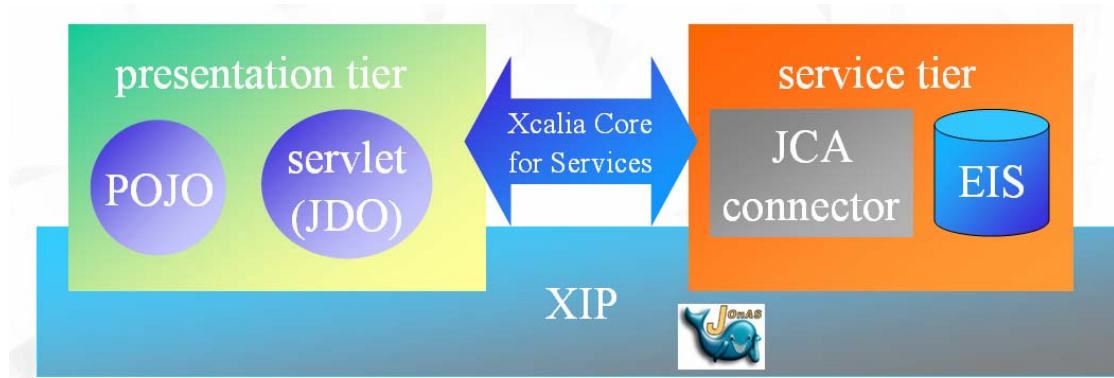
For the second step, you need to run a web browser and go at the following URL:

<http://localhost:9000/web-application/Customer>

The results of the sample execution should be displayed after a few seconds.

Understanding the sample

The following figure displays the architecture of the example:



Two components are deployed in the Xcalia Intermediation Platform:

- a component dedicated to the service tier, provided as a RAR file; it contains a JCA connector that accesses data from an EIS
- a component dedicated to the presentation tier, provided as a WAR file; it contains a servlet and POJO persistent classes

The communication between the two tiers is done with Xcalia Core for Services.

Here is the directory structure of the sample sources:

```
xcalia.samples.service.jca
|- ra -> the JCA connector
|   |_ eis -> the EIS embedded in the JCA connector
|_ webapp
|   |- invoker -> the configuration of the JCA invoker
|   |- pojo -> the persistence-capable classes
|   |- servlet -> the servlet of the web application
|_ util -> a tool for initializing the database
```

The service tier

The service tier is packaged in a RAR file. This file contains:

- a JAR file that contains the Java classes that implement the JCA Connector
- a META-INF directory that contains
 - o ra.xml: describes the JCA Connector (general information and the classes that implement some interfaces of the JCA specification)
 - o jonas-ra.xml: an application server specific (in this case, the JONAS application server from ObjectWeb packaged with Xcalia Intermediation Platform) deployment descriptor; it allows to specify the JNDI name of the Connector's ConnectionFactory class

The implementation of the service tier is divided in two parts:

- the EIS part, which manages customer data
- the connector part that wraps the EIS part and makes it connectible to other Java applications

The EIS part

The EIS part consists in a Java class that contains several methods to read, update, create and delete customer data. The open source ObjectWeb Jalisto data source is used to store the data.

NB: Jalisto is a lightweight (non relational) 100% Java, data source that stores data on the file system. It has been contributed to ObjectWeb by Xcalia.

The EIS contains two classes, in the package `xcalia.samples.service.jca.ra.eis`:

- the class `Customer` that describes the data persisted by Jalisto
- the class `CustomerService`, that contains method to manage customer data

The class `CustomerService` contains four methods:

- `String getCustomerName(Integer id)`: returns the name of the identified customer
- `void changecustomerName(Integer id, String name)`: changes the name of the identified customer
- `void createCustomer(Integer id, String name)`: creates a new customer
- `void deleteCustomer(Integer id)`: deletes the identified customer

These methods are accessible through the connector part which redirects the calls to the EIS part. Thus these methods are service methods that need to be mapped to persistent data through Xcalia Core for Services.

The connector part

The connector implements some interfaces of the JCA specification. All its classes are contained in the `xcalia.samples.service.jca.ra` package. The most noticeable part is the `InteractionImpl` class and its `execute()` method. This method redirects and marshals method calls to the underlying EIS part, and unmarshals the result.

The presentation tier

The presentation tier is packaged in a WAR file that contains the following arborescence:

```
WEB-INF
| - lib
|   | - jdo_2_0_0.jar
|   | - Xcalia Core jars
| - classes
|   | - the servlet class + all the needed classes
|   | - a valid license file
```

```

|   | - .jdo file
|   | - lido-package.xml
|   | - .xsm file
|   | - .xse file
|   | - lido.properties
| - web.xml
|_ jonas-web.xml

```

The `web.xml` file describes the web application: which servlet is used and from which URL it is callable.

The `jonas-web.xml` file describes the host of the web application. It is specific to JONaS.

This tier is implemented with a servlet that executes a customer data access scenario.

The customer data are transparently retrieved from the service tier using the JDO API.

The service tier thus acts as a data source, hiding the actual underlying Jalisti database. The mapping to this tier is done with Xcalia Core for Services.

The servlet needs the following elements:

- the persistent classes, implemented as POJO classes, and their JDO metadata file
- the mapping to the service
 - o the description of the service behaviour
 - o the configuration of the service invoker

Definition of persistent classes

A `Customer` class, with fields `identifier` and `name` has been defined. It is defined like your standard JDO persistent class, with the proper JDO metadata file.

Description of service behaviour

The service behaviour is described in a `.xsm` file.

The first part describes the service entity model, in other terms the supposed structure of the data that are managed by the service. The service entity model of this sample is:

```

<entitymodel>
  <entityclass name="CustomerEntity">
    <field name="identifier" type="Integer"/>
    <field name="name" type="String"/>
  </entityclass>
</entitymodel>

```

The persistent classes are mapped to this service entity model in the `lido-package.xml` file.

The service method model describes the behaviour of the service methods in terms of data from the service entity model management. Here is an example of method specification:

```

<servicemethod>
  <prototype>
    <name>getCustomerName</name>
    <return>String</return>
  </prototype>
</servicemethod>

```

```

        <parameters>
            <parameter name="id" type="Integer"/>
        </parameters>
    </prototype>
    <behavior>
        Map identifier = new Map;
        identifier->CustomerEntity.getField("identifier") = id;

        CustomerEntity customer = CustomerEntity.get(identifier);
        return customer.name;
    </behavior>
</servicemethod>

```

Configuration of the JCA invoker

The JCA Invoker is configured in a .xse file. The following must be provided:

- the JNDI name of the connector's `ConnectionFactory` implementation class
- the name of a class that implements `xcalia.lido.api.service.jca.JCAManager`. Such a class is required, because it describes the interactions with the JCA Connector

Here is the used .xse file:

```

<enablers>
    <jcainvoker service="CustomerService">
        <managerclass=
            "xcalia.samples.service.jca.webapp.invoker.MyJCAManager"/>
        <connectionFactory>
            <jndi-name>local_cxf</jndi-name>
        </connectionFactory>
    </jcainvoker>
</enablers>

```

In the JCA Manager, four methods must be implemented:

- `createInteractionSpec()`: describes how an `InteractionSpec` is configured from a service method
- `createInputRecord()`: creates an input `Record` from the arguments of the invocation (how to marshal the arguments)
- `createOutputRecord()`: creates an output `Record` from the return type of the service method
- `getInvocationResult()`: extracts a result object from an output `Record` (how to unmarshal the result)

Conclusion

In this article, we showed how to integrate a datasource whose specificities are hidden behind a service-oriented component, in a J2EE environment. Thus, a web application accesses data from a JCA component in a transparent manner. This was made possible with the help of Xcalia Core for Services.