SimbaProvider SDK 4.6

Developer's Guide for Java

October 6, 2016

Simba Technologies Inc.

Copyright ©2016 Simba Technologies Inc. All Rights Reserved.

Information in this document is subject to change without notice. Companies, names and data used in examples herein are fictitious unless otherwise noted. No part of this publication, or the software it describes, may be reproduced, transmitted, transcribed, stored in a retrieval system, decompiled, disassembled, reverse-engineered, or translated into any language in any form by any means for any purpose without the express written permission of Simba Technologies Inc.

Trademarks

Simba, the Simba logo, SimbaEngine, and Simba Technologies are registered trademarks of Simba Technologies Inc. in Canada, United States and/or other countries. All other trademarks and/or servicemarks are the property of their respective owners.

Simba Technologies Inc.

938 West 8th Avenue
Vancouver, BC Canada
V5Z 1E5

Tel. +1 (604) 633-0008
Fax. +1 (604) 633-0004

www.simba.com

Printed in Canada

Table of Contents

Developer's Guide for Java i
Introduction 1
Java Provider Prerequisites 1
Developing Your Provider in Java 2
  Copying and Modifying the XMLA Sample Provider as a Starting Point 2
Basic Provider Implementation Tasks 3
  Handling Errors 3
  Returning Schema Table Metadata 4
  Executing an MDX Query 6
Threading Safety 7
  J2EE Threading Issues 8
Appendix A – JISO Interfaces and Classes 9
Appendix B – Supported API Interfaces 11
  XMLA Supported Schema Rowsets and Properties 11
Appendix C – Supported API Interfaces 15
  Logging 15
  Packaging as a J2EE Application 15
  Session Management 17
  Connection Pooling 18
  Using Java Native Interfaces (JNI) 18
  JNI in a J2EE application 19

Table of Figures

Figure 1 - Threading situations to avoid 8

Introduction

Welcome to the SimbaProvider for OLAP SDK™ 4.6 Developers Guide. This guide describes the basic development tasks necessary to take the sample Java OLAP provider in the SDK and use it as a starting point in building your own provider.

Note: This guide assumes that you already have the SimbaProvider for OLAP SDK™ installed and configured for development as described in the SimbaProvider for OLAP SDK™ 4.6 Quick Start Guide. It also assumes that you have a good understanding of OLAP principles and terminology.

Development of a sample provider is usually done by taking the sample provider included in the SDK and modifying it to access your own data source. Version 4.6 of the SimbaProvider for OLAP SDK™ includes a sample data source implemented using .csv (text) files. This allows you to get up and running quickly with the sample since no additional drivers or software (e.g. servers) are necessary. Note that prior to SimbaProvider for OLAP SDK™ 4.5, the sample data required SQL Server, Microsoft Analysis Services, and other components to be installed. Existing users who have followed these steps in the past will therefore need to review the new sample code described in this document to understand any required changes.

Java Provider Prerequisites

A Java provider is composed of a "native" ISO implementation written in C++, and a Java "wrapper" around this C++ implementation. Methods calls are marshaled between the Java classes and the underlying C++ classes using JNI.

Before building your Java provider, you must modify and build the library for ISO implementation in C++ to work with your own data source. The Visual Studio solution for this project is *\Simba Technologies\SimbaProvider for OLAP SDK 4.6\SampleProvider\Win32\CustomerJavaXMLA VS2015.sln.* You can also test your C++ implemention by building the C++ sample provider as described in *the OLAP Developers Guide for C++.*

Once your ISO implementation is complete, you can use the sample Ant build script to build a WAR file containing an XMLA Web Service.

The sample Java provider referenced in this document includes an Eclipse workspace located in *\Simba Technologies\SimbaProvider for OLAP SDK 4.6\SampleProvider\Java\Eclipse Workspace* that you will start with and modify. The end result will be an XMLA web service that will be deployed to either JBoss or Tomcat.

The Eclipse workspace consists of an Ant script that builds and deploys your Java WAR to JBoss or Tomcat. With version 4.6 of the SimbaProvider for OLAP SDK™ the supplied Java JARs implement the XMLA Web Service and interface with your ISO implementation. No Java code development is required.

Developing Your Provider in Java

Copying and Modifying the XMLA Sample Provider as a Starting Point

This section describes how take the sample Java XMLA provider in the base kit, copy it, and modify it so that it can be used as a starting point in creating in your own XMLA provider. The Java XMLA provider generates a web service for use in JBoss or Tomcat. This section assumes that you are familiar with setting up a web service in JBoss or Tomcat and therefore does not contain the specific steps necessary to run the webservice.

1. If you have built the deployment WAR file using the Ant build script, then browse to *<Install Path>\SimbaProvider for OLAP SDK Eval 4.6\SampleProvider\Java* and remove the *build* directory.
2. Create a new directory on your hard drive. This directory will be the root of your provider source code tree.
3. Navigate to *<Install Path>\SimbaProvider for OLAP SDK Eval 4.6*
4. Copy and paste the *SampleProvider* and *Simba* folders to this new directory.
5. Choose an appropriate name for your new provider and rename the *SampleProvider* folder copied in the previous step. From now on, we will refer to this new folder as *<YourProviderFolder>*.
6. (Optional) Change the filename to use when outputting the .war file by opening *build.xml* file in *<YourProviderFolder>\Java* and adjusting the *dist.war* property value.

   Rename this value from *XmlaWebService.war* to your desired WAR filename, which from here on is referred to as *<YourXmlaAlias>.war*. Note that this will affect your XMLA service end point URL.
7. Update the SIMBAHOME environment variable to reflect the new location of Simba jars. This will be *<YourProviderFolder>*.
8. Open *<YourProviderFolder>\Java\Eclipse Workspace\TestFiles\DataSources.xml* and change the values for the *<DataSourceName>*, *<DataSourceDescription>*, *<URL>* and *<ProviderName>* elements. The value for the *<URL>* element needs to be updated to *http://localhost/<YourXmlaAlias>localhost/<YourXmlaAlias>* following values to reflect the information for your own provider:
9. Rebuild and redeploy your new solution using Ant to verify you have made all necessary changes. Run Ant from the command line or from Eclipse by loading the workspace located under *<YourProviderFolder>/Eclipse Workspace*.

The provider can now be accessed as a web service by navigating to *http://localhost:8080/<YourXmlaWebAlias>* using a web browser, but will not yet provide access to the underlying OLAP data stored in .csv files. The following sections will describe the

implementation of key elements of the provider, and provide suggestions on how to modify them for your own data source.

Basic Provider Implementation Tasks

No Java code implementation is required to build the Java XMLA provider to use your underlying C++ ISO implementation. This is the recommended and easiest way to re-use your implementation with the ODBO provider projects.

If you want to build a Java XMLA provider entirely within Java that does not use your C++ ISO implementation then you need to implement the interfaces described in "Appendix A – JISO Interfaces and Classes" on page 9. The Javadocs located at *\Simba\Java\javadoc* described the interfaces and methods in more detail. The following subsection provide more information on some of the interfaces.

Handling Errors

Use the information provided in this section, when your provider implementation needs to notify the user about "real world" errors which occur (for example, failure to connect).

The SimbaProvider for OLAP SDK includes a Java exception class called *OLAPException* (defined in *Simba.Olap.jar*) along with a number of derived exceptions, which must be thrown by your code when errors occur.

Since XMLA providers exist as web services, they return results and errors to remote client consumer applications through the following XML responses:

- **Discover/Execute errors**: if a *Discover* or *Execute* command sent to a provider causes an error, a provider must return a *SOAP fault* response containing details about the error including:

  Fault Code: set to soap:Server if the error occurred due to a problem on the server, or set to soap:Client if the Discover/Execute command sent by the consumer caused the problem.
  Fault String: a detailed description of the problem.
  Fault Actor: the URL that the consumer application used to access the provider.
  Detail: application-specific error information.

  Note: No data or partial result set is returned in this case.

  The following snippet shows an example of a SOAP fault:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
     <soap:Body>
          <soap:Fault>
               <faultcode>soap:Client</faultcode>
```

```
                <faultstring>The request contains an error. ---
                     &gt; Unknown set identifier: "[BadSet]"
                </faultstring
                <faultactor>http://localhost/CustomerXmla/XmlaWebS
                     ervice.asmx
                </faultactor>
                <detail />
           </soap:Fault>
     </soap:Body>
</soap:Envelope>
```

- **Result failures**: if the cells of a dataset or rowset returned by the provider contain an error, the error message will be embedded directly in the data result set a shown in the following snippet:

```
<Cell CellOrdinal="0">
     <Value>
           <Error>
           <Description> An integrity constraint violation or data
                conversion error was detected while accessing a
                field in the current row.
           </Description>
           </Error>
     </Value>
     <FmtValue> Infinite recursion </FmtValue>
</Cell>
```

The usage of the *OLAPException* class and those derived from it will automatically perform the necessary logic to propagate errors without having to deal with SOAP faults, and/or embedding XMLA errors; it is all handled for you.

Note: Good programming practice dictates that exceptions, and not assertions, be used to catch errors which occur due to “real world” events such as an invalid password, or inability to connect to a data store. The use of exceptions allows a provider implementation to perform clean up and other restorative procedures before the consumer application and end user are notified of the error. Assertions on the other hand must only be used for development purposes to ensure that code is robust.

Each of the exceptions derived from *OLAPException* is intended to report a specific error (e.g. *ConnectionFailedException* should be thrown when an error occurs in establishing a connection to the datasource). You may also want to derive your own exceptions for other errors that your provider needs to encounter. To do so, simply derive a class from *OLAPException*.

Returning Schema Table Metadata

The calling framework will request a multitude of schema information from a provider including the supported keywords and literals, for example.

When the calling framework requests schema rowsets, it invokes a number of methods in the provider to retrieve column information, rowset restrictions, and of course the rowset itself, populated with metadata.

In a Java provider, schema rowsets are retrieved from your datasource through the *IConnection.getSchemaReader()* method. This method takes in the GUID of the schema to return rowsets for, along with a collection of objects representing the restrictions and returns an *IRowReader* object which the calling framework can use to read each row. This method needs to performs three key operations:

- **Translation of Guids**: each schema GUID is translated into an appropriate schema rowset request from the datasource. The type of schema rowset being requested is identified using a GUID. Your datasource more than likely does not understand these GUIDs so you need to match each of them appropriately to you data source's constructs. The GUIDS for all the standard schema rowsets are defined statically in the *com.simba.olap.Schema* class.
- **Handling restrictions**: you need to filter the results based on the given restrictions. If your data source is capable of performing this sort of filtering, then the work involved here is simply a matter of formatting the restrictions in a form acceptable to your data source. If your data source does not perform any sort of filtering, then you will need to filter the results yourself. Restrictions are passed to this method in the form of an array of restriction values. Each array value is implicitly mapped to the restrictions of the given rowset in the order they appear. The Simba XMLA layers use the schema information that your implementation returns from the *IConnection.getSchemas()* method to determine which restrictions each schema rowset supports, and in what order they should appear.
- **Return an IRowReader**: a row reader must be returned so that the calling framework can get the metadata. This will be discussed in further detail below.

Returning Supported Schema Information

A provider must be able to return schema information when the consumer requests a *DISCOVER_SCHEMA_ROWSET* rowset. To do this, the calling framework will invoke the *IConnection.getSchemas()* method to retrieve information regarding the schema rowsets supported by your datasource. This includes which columns the schema rowsets have and which restrictions the schema rowsets support. This information is encapsulated in the *com.simba.olap.Schema* class.

Returning Schema Metadata Rows

As mentioned above in "Returning Schema Table Metadata", when the calling framework requests the actual rows, it will invoke the connection's *getSchemaReader()* method passing in the GUID of the table to return rows for along with a collection of objects representing the restrictions.

The *IRowReader* interface is an abstraction of a collection of cells organized into rows and columns. The *IRowReader* has methods to iterate through all rows in the dataset, and iterate through each field (i.e. column) within the current row. Row reader objects are used for several purposes in Java provider including:

1. To implement schema rowsets that return database metadata to the consumer.

2. To implement axis rowsets that return metadata about the axes of a dataset (multidimensional query result) to the consumer.
3. To hold dataset cells that are returned to the consumer as part of a query result.

An implementation of *IRowReader* must be composed of a *ColumnInfoCollection* member that holds a read-only collection of *ColumnInfo* objects. The *ColumnInfo* object encapsulates a column in a table.

Executing an MDX Query

The *ICommand* interface is responsible for executing MDX statements and returning results (e.g. dataset) for those statements that request a result.

Note that *executeDataset()* can also be invoked by a statement that does not produce a result. For this case, your implementation should return null as long as the command executed successfully.

If clients explicitly indicate that they do not expect a result, then the *executeNonQuery()* method will be called. Usually a client would choose to do this for statements such as *CREATE MEMBER* in MDX that do not produce a result.

Note: For any of the execute methods, if there are errors in the command text, an *ErrorsInCommandException* should be thrown.

Exposing Command Results

An implementation of *IDataset* is returned by the *ICommand.executeDataset()* method. Further information about some of the *IDataset* methods is below.

IDataset.createAxisReader()

A provider must return a table containing information about all tuples on each axis of a dataset. This is performed in the *IDataset.createAxisReader()* method which returns an *IRowReader* where each row in the rowset represents a tuple. Each row is composed of a column for the tuple ordinal followed by groups of columns, where each group corresponds to a single member of the tuple. Each column in the group represents a property of the corresponding member for that group. Each group has columns for at least five properties: MEMBER_UNIQUE_NAME, MEMBER_CAPTION, LEVEL_UNIQUE_NAME, LEVEL_NUMBER, and DISPLAY_INFO. There will be an additional column in each group for each dimension property requested in the DIMENSION PROPERTIES clause for the axis in the MDX statement. For more information, see https://technet.microsoft.com/en-us/library/ms144780.aspx.

IDataset.getCellDataReader()

A dataset must be able to return information about a range of cells in its result set. Cell properties should be included when the consumer includes a *CELL PROPERTIES* statement in an MDX command. This is accomplished in the *IDataset.getCellDataReader()* method of *Dataset* which takes in the cell ordinals to return information for.

Note that the *CELL PROPERTIES* statement specified by the consumer can be appended with any number of specific properties to return information for. The minimum set of properties supported by the sample provider are *VALUE*, *FORMATTED_VALUE*, *CELL_ORDINAL* and *FORMAT_STRING*. For a complete list, see http://technet.microsoft.com/en-us/library/ms145573.aspx.

Returning Rowsets

When a Java provider is called upon to execute a command which returns a rowset, the *ICommand.executeReader()* method is invoked by the calling framework. Multidimensional results must be returned to the calling framework as a flattened rowset. You will need to create your own *IRowReader* implementation and perform the necessary flattening to convert the multidimensional result into a tabular form.

Threading Safety

If you build your XMLA provider entirely in Java your JISO implementation must be thread-safe because your provider may be accessed from multiple threads. The application server may create different threads to deal with different requests, and any of these threads may be running concurrently. In general, all singleton objects and static data must be protected from concurrent access.

The Simba XMLA layers guarantee that each JISO object instance is protected from simultaneous access. However, if your JISO objects contain pointers to each other, the situation becomes more complex.

As an example of a situation to be aware of, consider your *Command* object. Assume for the purposes of this example that each *Command* object contains a “back-pointer” to the *Connection* object that created it. The XMLA layer guarantees that an instance of your *Command* object is called from only one thread at a time. However, it makes no guarantee that another thread won’t be calling your *Connection* object while your *Command* object is being called. If your *Command* object makes use of your *Connection* object through the back-pointer, then it is possible that the *Connection* object may be accessed from more than one thread simultaneously. You must be vigilant in protecting your JISO objects from situations such as this. As a general rule, you should avoid back-pointers wherever possible, except for simple uses such as reference-counting.

Also be aware of potential deadlock risks. This situation is more rare because typically objects call “up” to their creators (as in the example above) rather than “down” to the objects they create. As an example of deadlock potential, consider a *Connection* object that must call down to one of its *Command* objects for some reason. If the Command object simultaneously calls “up” to the *Connection* and both objects are instance locked, deadlock will result. Fortunately, this example is somewhat contrived and would only arise as the result of poor application of object-oriented design practices.

Figure 1 illustrates these two situations to avoid.

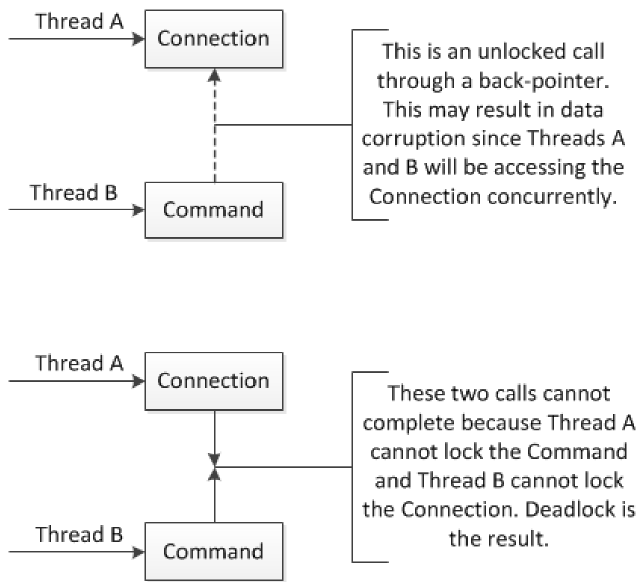

Figure 1 - Threading situations to avoid

J2EE Threading Issues

Multithreading must be considered carefully when developing a J2EE application. Using multiple threads in an Enterprise Java Bean is discouraged since the server has no way of managing your threads.

If however, you are using the JCA (J2EE Connector Architecture) resource adaptor in your implementation, then you can use multiple threads in a way that is compatible with J2EE. The *javax.resource.spi.work.WorkManager* provided by the application server allows you to schedule a task on an application server thread.

Appendix A – JISO Interfaces and Classes

The following table lists the JISO interfaces and various helper classes that you may use to help implement these interfaces.

Table 1 - JISO Interfaces and Classes

Interface/Class Name	Purpose
com.simba.olap.jiso.IConnection	Represents a connection to a data source. Also provides methods for retrieving information about the data source.
com.simba.olap.jiso.ICommand	*ICommand* provides a factory for creating *IDataset* and *IRowReader* objects by executing queries.
com.simba.olap.jiso.IDataset	This is an interface to an object that contains the result of an MDX Query.
com.simba.olap.jiso.IRowReader	This is an interface to an object that represents a forward-traversal-only table. It contains column information and is a factory for *IRows*.
com.simba.olap.jiso.IRow	An *IRow* is read from an *IRowReader*. An *IRow* contains all the cell data for a given row in the table.
com.simba.olap.jiso.IDisposable	This base interface for all the JISO interfaces provides a method for disposing of an object.
com.simba.olap.jiso.EmptyRowReader	This helper class implements the *IRowReader* interface for an empty table, one that contains no rows. It can be returned whenever an empty result is appropriate.
com.simba.olap.jiso.ConnectionStringParser	This helper class can be used to parse connection strings into a mapping of key-value pairs stored in a *ConnectionStringMap*.

com.simba.olap.jiso.ConnectionStringMap	This is a data structure that provides convenient lookup of connection string key-value pairs.
com.simba.olap.jiso.ConnectionStringBuilder	This helper class can be used to build a connection string, providing methods to easily append key-value pairs to a connection string.
com.simba.olap.jiso.CellOrdinalGenerator	This helper class can be used to generate the cell ordinals for your cell table.
com.simba.xmla.XmlaWebService	This is the base class web service, which you must extend to implement your web service. It provides methods for implementing the XMLA discover and execute methods, and also acts as a connection factory for your *IConnection* implementation.
com.simba.xmla.IConnectionFactory	This is the basic interface which must be implemented by your web service so that the Simba XMLA layers can access your connection implementations.

Note: For more information on these classes and their methods, refer to the corresponding Javadoc documentation.

Appendix B – Supported API Interfaces

XMLA Supported Schema Rowsets and Properties

The following table lists the properties supported by the XMLA layer.

Table 2 - Property Support

Supported Properties	
	AxisFormat
	BeginRange
	Catalog
	Content
	DataSourceInfo
	EndRange
	Format
	LocaleIdentifier
	MDXSupport
	Password
	ProviderName
	ProviderVersion
	StateSupport
	UserName

Properties not Supported	Cube
	Timeout
	ClientProcessID
	DbpropMsmdActivityID
	DbpropMsmdCurrentActivityID
	DbpropMsmdFlattened2
	DbpropMsmdMDXCompatibility
	DbpropMsmdOptimizeResponse
	DbpropMsmdSubqueries
	Dialect
	MdxMissingMemberMode
	ReturnCellProperties
	SafetyOptions
	ShowHiddenCubes
	SspropInitAppName

Note: The Simba XMLA layers recognize and accept the unsupported properties. However, if a consumer application supplies them no action is taken.

The following table lists all available schema rowsets. Note that the first four rowsets are implemented by Simba. The remaining schema rowsets are part of your ISO implementation and, except for *DBSCHEMA_CATALOGS*, are also required for ODBO providers. You can implement

additional schema rowsets not listed here in your ISO implementation. Consumer applications will be able to retrieve the data in your additional schema rowsets via the *Discover* method.

Table 3 - Schema Rowset Support

Schema Rowset	Comments
DISCOVER_DATASOURCES	ProviderType restriction is not supported.
DISCOVER_PROPERTIES	
DISCOVER_ENUMERATORS	
DISCOVER_SCHEMA_ROWSETS	
DISCOVER_KEYWORDS	
DISCOVER_LITERALS	Additional informational column, LiteralID, at the end of each row in the rowset.
DBSCHEMA_CATALOGS	
MDSCHEMA_CUBES	
MDSCHEMA_DIMENSIONS	
MDSCHEMA_HIERARCHIES	
MDSCHEMA_LEVELS	
MDSCHEMA_MEASURES	
MDSCHEMA_MEMBERS	
MDSCHEMA_PROPERTIES	

MDSCHEMA_FUNCTIONS	
MDSCHEMA_FUNCTIONS	
MDSCHEMA_KPIS	
MDSCHEMA_SETS	
MDSCHEMA_MEASUREGROUPS	
MDSCHEMA_MEASUREGROUP_DIMENSIONS	

Appendix C – Supported API Interfaces

Logging

Java XMLA project uses Apache Commons logging. Apache Commons logging is just a wrapper around the actual logging toolkits. The actual logging can be performed by any of the following logging libraries:

- Apache Log4j
- JDK standard logging API
- Apache Simple Log

By default, Java XMLA uses *Log4j*. Assigning a different logging implementation to *org.apache.commons.logging.Log* variable, which by default is assigned *org.apache.commons.logging.impl.Log4Jlogger* class, can change the logging toolkit.

Also, using the *setLogFactory()* method of com.simba.axis.components.logger a different log factory can be set so that the entire application will be using your log factory to obtain and instance of the logger class.

The Jakarta Commons Logging (JCL) performs a discovery process upon initialization, which is well documented in Jakarta's documentation (http://jakarta.apache.org/commons/logging/commons-logging-1.0.4/docs/guide.html).

The system properties can be defined in a property file named *commonslogging. properties*, which must be placed somewhere on the application class path. The Java XMLA project already contains a *commons-logging.properties* file that assigns *org.apache.commons.logging.impl.Log4Jlogger* to *org.apache.commons.logging.Log*, which enforces usage of Log4j. Although without this file, Log4j would still be the immediate choice in the discovery process (please see discovery process above).

You can change the logger implementation by editing the *commonslogging.properties* file that accompanies the source code. However, note that there are currently only three logging toolkits that are compatible with Apache logging. Using any other logging toolkit requires that you implement the required interfaces for Jakarta commons logging compatibility.

Also note that changing the logger requires that you also provide the dependency jars for your logging toolkit of choice, and possibly the configuration files for your logging toolkit. The dependency jars must be placed under *<InstallDir>/Simba/Java/jars* directory, and the project Ant build script may need to be modified to package up your logging libraries inside the XMLA web service WAR file.

Packaging as a J2EE Application

If you are planning to use Enterprise Java Beans in your application, then you may want to look at packaging your J2EE application as an EAR file. Sun Microsystems has an overview on packaging a J2EE application, which you can find it at http://java.sun.com/j2ee/1.4/docs/tutorial/doc/ (look

under 'Packaging Applications' header). You may also want to download the J2EE 1.4 specifications from http://java.sun.com/javaee/downloads/index.jsp.

An EAR file is essentially a JAR file with the following content:

- Web Component
- Enterprise Beans
- J2EE Application Client
- Resource adapter modules (JCA adapters)

Simba's Sample Provider deployment script creates a web component (a .war file) that can be directly deployed on an application server. In order to create a complete J2EE application EAR file, you need to create three more deployment descriptors to deploy your EJBs, J2EE application clients, and one descriptor for the entire EAR file. A minimal deployment only needs two additional descriptors, one for your EJBs and one for your EAR file.

Each of the items in the above list is essentially a JAR or a WAR file, and they are assembled together with their respective deployment descriptor into an EAR file. If you are using a JCA resource adaptor, then you will need to create a RAR file (which has a similar structure to a JAR file) with a resource adaptor descriptor in addition to the components above. Please consult the J2EE specification for more detail.

Here is a brief overview of what is included in an EAR file:

The EAR file deployment descriptors are:

- META-INF/application.xml containing:
    - Icons, a description and a name used by tools.
    - The modules contained within, their location within the EAR, the context root for each web application module, and alternative deployment descriptor references.
    - Security role names and descriptions

- META-INF/<vendor>-application.xml (optional vendor descriptor) containing:
    - Web application context-root overriding
    - A pass-by reference option for the application scope
    - Security mappings from role-name to user and group
    - A unique id that is managed during (re)deployment

EAR files can contain the following:

Enterprise JavaBean (EJB) component modules (.jar). The deployment descriptors are:

- META-INF/ejb-jar.xml
- META-INF/<vendor>-ejb-jar.xml- (required - vendor-specific)

Web application modules (.war). The deployment descriptors are:

- WEB-INF/web.xml
- WEB-INF/<vendor>-web.xml - (optional - vendor-specific)

Application client modules (.jar). The deployment descriptors are:

- application-client.xml
- <vendor>-application-client.xml - (vendor-specific)
- <vendor>-acc.xml (vendor specific)

Resource adapter modules (.rar). The deployment descriptors are:

- ra.xml
- <vendor>-ra.xml - (vendor-specific)

In the above list, <vendor> signifies the vendor of your application server. For example, if you are using JBoss, then <vendor> is JBoss.

Session Management

Sessions allow for statefulness in XMLA. This is useful for series of statements that should be performed together. An example of this is the creation of calculated members that are used in subsequent queries.

The client is responsible for sending proper SOAP headers for initiating, maintaining and ending sessions. The Simba Java XMLA session management sub-system is responsible for handling the lifetime of XMLA sessions. This subsystem will manage sessions according to the session headers received from the client, and will start a session, route a request to an existing session, or end a session depending on the request. For each session created, the sub-system will use the *IConnectionFactory* interface to create a new instance of your *IConnection* implementation. There will always be a one-to-one mapping between a session and a given connection.

In order to give the customer code enough control over the session behavior *IConnection* interface methods provides appropriate exceptions to abort the session when a session times out, or when a session becomes invalid. The following methods throw connection related exceptions:

- *IConnection.getSchemas*: should throw *ConnectionBrokenException* to terminate an invalid, broken or unusable connection. Also, throw *ConnectionTimedOutException* if a valid connection cannot be obtained or created in a timely manner.
- *IConnection.createCommand*: throw the same exceptions as *getSchemas* method above.

The *IConnection.getSchemas* method is called for *Discover* requests and, *IConnection.createCommand* is called for *Execute* method requests.

The subsystem will also manage the lifetime of a session according to a configurable timeout parameter. The time interval for this timer is configurable through the xmla.properties file, which can be found under the *<InstallDir>/Simba/Java/src/com/simba* directory. The timeout interval

configuration parameter is called *XMLASessionTimeout*. If the value of this parameter is set to 0, then the timer is disabled.

Connection Pooling

The *Connection Pooling* subsystem manages the pooling of connections so that previously established connections may be reused. This can enhance the performance of your Java XMLA provider by saving on the overhead of establishing a connection, which for some datasources can be costly.

The connection pooling sub-system will group connections according to their connection strings. If a connection request is made with the same connection string as one that had been previously established, and the previously established connection is available in the pool, then the sub-system will reuse that connection. If no previously established connections are available, the sub-system will create a new one.

Connections should be returned to the pool when they are no longer needed (i.e. their associated session has ended). Returning a connection to a pool will reset any stateful data that was associated with that connection.

You can take advantage of this subsystem by implementing the *IPooledConnection* and *IPooledConnectionFactory* interfaces in the *com.simba.olap.connectionpooling* package. You interface to the system using the *ConnectionPoolManager* class, and configure it by setting the various property values on the *ConnectionPoolConfig* class.

Please refer to the associated javadocs for more information.

Using Java Native Interfaces (JNI)

Simba Java XMLA sample provider utilizes JNI to incorporate Simba's sample C++ implementation. Using JNI may pose certain restriction that requires attention when designing and implementing projects that use JNI. The following issues concern customers who are wishing to use JNI to either use a C++ ISO implementation or develop their own JNI for interfacing to the C++ MDX engine and ISM implementation. All of the following issues have been taken into consideration in Simba's sample implementation.

1. The JNI interface pointer is only valid in the current thread. A native method, therefore, must not pass the interface pointer from one thread to another. A VM implementing the JNI may allocate and store thread-local data in the area pointed to by the JNI interface pointer. Care must be taken not to return a peer pointer, from any method, which can potentially be used across thread boundaries in any of the Java classes.
2. A single library may be used to store all the native methods needed by any number of classes, as long as these classes are to be loaded with the same class loader. The VM internally maintains a list of loaded native libraries for each class loader.

Vendors should choose native library names that minimize the chance of name clashes. An immediate repercussion from this feature of JNI is that a web application that uses JNI may not be reloadable (without restarting the container). It basically means that the application server may

have to be restarted in order to reload that web application. The reason is that VM has already loaded the native library in another class loader and it will refuse to load it again. Therefore, the JNI library and its native interfaces must be loaded by a class loader that is only loaded once during the lifetime of the application server.

This can be achieved by placing the JNI library and the Java classes that contain the native interfaces under a directory (within your application server directory tree) that is only loaded once, or the application server must be somehow instructed to load these classes and the JNI library only once.

Simba's JXMLA provider sample cannot be reloaded without restarting the servlet container (hence, Tomcat). The reason is that JNI native library is packaged along with the rest of the code in a WAR file and it is loaded by the same class loader that loads up the rest of web application classes. This behavior can be changed if the JNI interfaces and the native dynamic link library were placed somewhere within the application server directory tree where they are only loaded once. However, that requires changes to the Ant build and deployment scripts.

JNI in a J2EE application

You can use a JCA managed connection in your J2EE application to handle your native classes. However, if you store a reference to your managed JCA connections in your EJB session object, then you won't be able to serialize (or passivate) that session object because it includes JNI references and as it was said above there are restrictions around using JNI references in multiple threads, or multiple JVMs.