Contents

About This Document

1. Overview

Java Compliance 1-1
What Oracle Tuxedo Features Are Supported 1-2
What Is In This Manual 1-3

2. Installing XMLink

Contents of the Installation Package 2-1
Hardware and Software Requirements 2-1
Installing XMLink on UNIX 2-3
Installing XMLink on Windows 2-4

3. Configuring XMLink

Configuration Notes 3-1
Managed and Non-Managed Environments 3-2
Configuring the Environment 3-5
Setting XMLink Properties 3-8
Sample Configuration 3-12

4. Using Java to Call Services

Getting an Oracle Tuxedo Connection 4-3
Managing Oracle Tuxedo Transactions 4-6
Calling Oracle Tuxedo Services 4-7
Working with Application Data 4-9
Example: Calling Services using Java 4-14

5. Using XML to Call Services

Inputting Data Using XML 5-2
Returning Data in XML 5-5

Using an EJB to Input XML 5-8

A. Troubleshooting

Configuration Issues A-1
Tuxedo Error Messages A-2
Using XMLink with Tuxedo 6.5 ..... A-5

B. New Features in XMLink

XMLink 3.0 ..... B-1
XMLink 2.6 ..... B-4
XMLink 2.1 ..... B-5
XMLink 2.0 ..... B-5

Index

Using 

Tuxedo Edition

Prolifics

XMLink Release 3.2
Document 1205

May 2016

Copyright

This software manual is documentation for XMLink™ 3.2. It is as accurate as possible at this time; however, both this manual and XMLink itself are subject to revision.

Prolifics and XMLink are trademarks of Prolifics, Inc.

Adobe and Adobe Reader are registered trademarks of Adobe Systems Incorporated.

Linux is a registered trademark of Linus Torvalds.

Tuxedo is a registered trademark of Oracle Corporation.

WebSphere is a registered trademarks of International Business Machines Corporation.

Java and all Java-based marks are trademarks or registered trademarks of Oracle Corporation in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other product names mentioned in this manual may be trademarks or registered trademarks of their respective owners, and are used for identification purposes only.

Send suggestions and comments regarding this document to:

Technical Publications Manager
Prolifics, Inc.
24025 Park Sorrento, Suite 405
Calabasas, CA 91302

http://prolifics.com
support@prolifics.com
(800) 458-3313

Contents

About This Document

What You Need to Know viii
Documentation Website viii
How to Print the Document ix
Documentation Conventions ix
Contact Us! xi

1. Overview

Java Compliance 1-1
What Oracle Tuxedo Features Are Supported 1-2
What Is In This Manual 1-3

2. Installing XMLink

Contents of the Installation Package 2-1
Hardware and Software Requirements 2-1
Installing XMLink on UNIX 2-3
    Implementing File Protection 2-3
    Determining File Location 2-3
    How to Install from CD-ROM 2-4
Installing XMLink on Windows 2-4
    How to Run the Setup Program 2-4

3. Configuring XMLink

Configuration Notes 3-1

Managed and Non-Managed Environments 3-2
Managed Environment Settings 3-3
Installing Resource Adapter Archives 3-4
Deploying in a Non-Managed Environment 3-4
Configuring the Environment 3-5
Setting XMLink Properties 3-8
Setting JVM Properties 3-8
Setting Properties on a Resource Adapter 3-8
Setting Properties on a Connection Factory 3-10
Sample Configuration 3-12

4. Using Java to Call Services

Getting an Oracle Tuxedo Connection 4-3
Example: Connecting with a Java Client 4-3
Understanding the ConnectionFactory Interface 4-3
Supplying Connection Parameters 4-4
Getting Information about XMLink 4-5
Calling getRecordFactory 4-6
Specifying Transaction Access 4-6
Managing Oracle Tuxedo Transactions 4-6
Calling Oracle Tuxedo Services 4-7
Understanding the Interaction Interface 4-7
Understanding the TuxInteractionSpec Interface 4-8
Working with Application Data 4-9
Understanding the RecordFactory Interface 4-10
Working with Record Objects 4-11
FMLRecord and FML32Record Objects 4-12
Sample: Getting FML Data 4-12
CArrayRecord and StringRecord Objects 4-13
Character Encoding Support 4-14
Example: Calling Services using Java 4-14

5. Using XML to Call Services

Inputting Data Using XML 5-2
Elements in the XML Input DTD 5-3

Agenda Element 5-3
Connection Element 5-3
Transaction and Servicecall Elements 5-3
Field Elements 5-4
Data Elements 5-4
Example: Using XML Input 5-4
Returning Data in XML 5-5
Elements in the XML Output DTD 5-6
Resultset Elements 5-6
Returndata Elements 5-6
Error Elements 5-7
Xactionmsg Elements 5-7
Examples: XML Return Data 5-7
Using an EJB to Input XML 5-8

A. Troubleshooting

Configuration Issues A-1
Tuxedo Error Messages A-2
Using the TuxedoException Class A-3
Using the TuxedoReturnCodeWarning Class A-4
Using XMLink with Tuxedo 6.5 ..... A-5

B. New Features in XMLink

XMLink 3.0 ..... B-1
Changes to Connection Factory Properties B-1
No Default Setting for XMLink.tconn B-2
Improved XML support B-2
XMLink 2.6 ..... B-4
Embedded FML B-4
Character Encoding Support B-4
XMLink 2.1 ..... B-5
XMLink 2.0 ..... B-5
JNDI Lookup B-5
Installation Issues B-6
Setting Properties for Connections and Connection Factories B-6

Deploying in a Non-managed Environment..... B-6

Index

About This Document

This document explains XMLink and describes how to use XMLink to call services in an Oracle Tuxedo application.

This document covers the following topics:

- Chapter 1, “Overview,” gives an overview of XMLink.
- Chapter 2, “Installing XMLink,” describes how to install XMLink on Windows and UNIX.
- Chapter 3, “Configuring XMLink,” describes how to configure XMLink.
- Chapter 4, “Using Java to Call Services,” describes how to write client code to call services in an Oracle Tuxedo application.
- Chapter 5, “Using XML to Call Services,” describes how to use XML to exchange data.
- In addition, there are appendices containing troubleshooting tips and an explanation of new features.

What You Need to Know

This document is intended for application developers interested in building Java applications that access services in an Oracle Tuxedo application. It assumes a familiarity with Oracle Tuxedo applications and Java programming.

Even though XMLink works with any J2C 1.5-compliant application server, this document primarily describes how to use XMLink with IBM WebSphere application server. For any information on other application servers, check the Release Notes area of the XMLink documentation site at http://docs.prolifics.com/docs/tconn/index.html.

- For more information in general about Java, refer to the Oracle Java site at http://www.oracle.com/technetwork/java/index.html.
- For more information about J2EE Architecture, refer to http://docs.oracle.com/javaee/1.2.1/devguide/html/DevGuideTOC.html.
- For more information about Oracle Tuxedo applications, refer to http://www.oracle.com/technetwork/middleware/tuxedo/overview/index.html.

Documentation Website

The XMLink documentation website includes the *Using XMLink* manual in HTML and PDF formats and the Java API documentation in Javadoc format. The website also enables you to search the HTML files for both the manual and the Java API.

This documentation is also distributed with the product either in the `docs` directory or on the product CD.

XMLink product documentation is available on the Prolifics corporate website at http://docs.prolifics.com/docs/tconn/index.html.

How to Print the Document

You can print a copy of this document from a web browser, one file at a time, by using the File→Print option on your web browser.

A PDF version of this document is available from the documentation website. You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format.

If you do not have the Adobe Acrobat Reader, you can get it for free from the Adobe website at https://get.adobe.com/reader/otherversions/.

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
Ctrl+Tab	Indicates that you must press two or more keys simultaneously. Initial capitalization indicates a physical key.
italics	Indicates emphasis or book titles.
boldface text	Indicates terms defined in the glossary.

Convention	Item

`monospace text`	Indicates code samples, commands and their options, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`chmod u+w *`
	`/usr/prolifics`
	`tconn.jar`

`monospace italic text`	Identifies variables in code representing the information you supply. *Example*: `String` *`expr`*

`MONOSPACE UPPERCASE TEXT`	Indicates environment variables, logical operators, SQL keywords, mnemonics, or Prolifics constants.
	Examples:
	`CLASSPATH`
	`OR`

`{ }`	Indicates a set of choices in a syntax line. One of the items should be selected. The braces themselves should never be typed.
`|`	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.
`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed. *Example*: `java com.prolifics.tconn.TConnTool [-list]`
`...`	Indicates one of the following in a command line: ■ That an argument can be repeated several times in a command line ■ That the statement omits additional optional arguments ■ That you can enter additional parameters, values, or other information The ellipsis itself should never be typed. *Example*: `formlib [-v]` *`library-name`* `[`*`file-list`*`]...`

Convention	Item
. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

Contact Us!

Your feedback on the documentation is important to us. Send us e-mail at support@prolifics.com if you have questions or comments. In your e-mail message, please indicate that you are using the documentation for XMLink 3.0.

If you have any questions about this version of XMLink, or if you have problems installing and running XMLink, contact Customer Support via:

- Email at support@prolifics.com
- Prolifics website at http://profapps.prolifics.com

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address and phone number
- Your company name and company address
- Your machine type
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

CHAPTER

1 Overview

XMLink provides Java programmers, specifically those developing EJBs (Enterprise Java Beans), access to services developed as part of Oracle Tuxedo applications. Typically, XMLink integrates existing services in an Oracle Tuxedo application with a new application using Java and EJBs.

Java Compliance

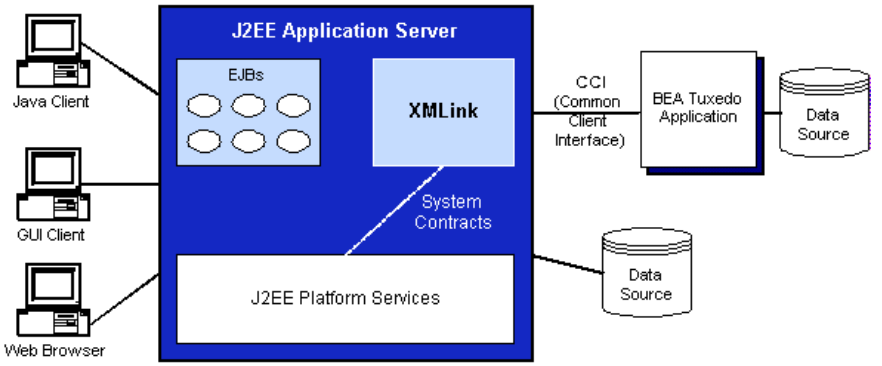
XMLink adheres to the J2EE (Java 2 Platform, Enterprise Edition) Connector Architecture (J2C) Specification, version 1.5. J2EE is designed to be used with multitiered, enterprise applications which separate the business logic and presentation aspects of applications from the system services provided by the J2EE platform. The addition of J2EE Connector to the J2EE platform allows integration of existing Enterprise Information Systems (EISs) to Java-based applications.

In the language of the J2C specification, XMLink is an outbound resource adapter and supports two deployment scenarios needed for J2EE compliance. J2EE resource adapters can either be run under the auspices of an application server (such as WebSphere) in which case they rely on services provided by the application server's framework, or they can be run in a non-managed environment, in which case they are local to a Java client and do not need an application server. This non-managed environment is conceptually similar to a two-tier application.

XMLink also implements an XML interface and the CCI (Common Client Interface) layer. The XML interface allows for data exchange in an XML format. The CCI API simplifies application access and development since developers can use the same set of API calls to connect to any underlying EIS.

The latest J2C specification, API documentation, and class files can be found at http://www.oracle.com/technetwork/java/javaee/index-138715.html.

Figure 1-1 XMLink in the J2EE Connector Architecture



What Oracle Tuxedo Features Are Supported

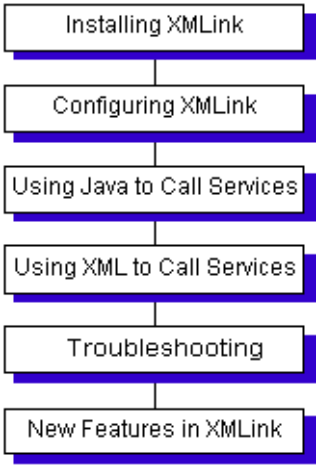
The Oracle Tuxedo features to which XMLink provides access are:

- Service calls
- Asynchronous service calls that do not expect a reply (such as `tpacall` calls with the flag `TPNOREPLY`)

XMLink also supports transaction demarcations using `BEGIN`, `COMMIT`, and `ROLLBACK` for Tuxedo 7.1 and later.

What Is In This Manual

The chapters in this manual describe the tasks needed to install and deploy XMLink.



CHAPTER

2 Installing XMLink

Contents of the Installation Package

- Java files compiled in Java archives
- DTDs used for exchanging data in XML
- Software libraries used to access your Oracle Tuxedo applications
- Online documentation in HTML and PDF formats
- Resource adapter archives

Hardware and Software Requirements

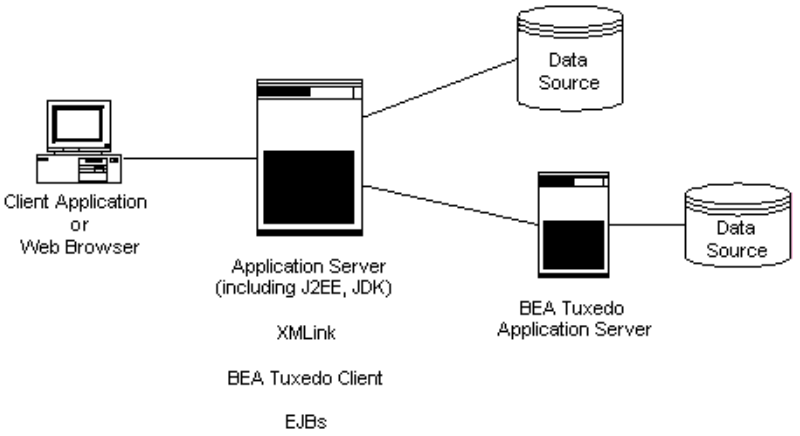
XMLink requires a maximum of 2.1 MB of disk space, of which 1.5 MB is for online documentation.

In addition to XMLink, you need:

- J2EE-compliant application server, version 1.4 (The application server must support the Connection Architecture 1.5 Specification.)

- J2SE, version 1.4 (Latest versions of IBM WebSphere include this package.)
- JRE (Java Runtime Environment) installed on clients (Latest versions of IBM WebSphere include this package.)
- Oracle Tuxedo Server and Client installed, version 6.5 or higher
- Connector architecture reference packages needed by your application server
- For the XML interface, which uses the XML Connector class, an XML parser is required, such as Xerces.

Figure 2-1 General Installation Setup



Installing XMLink on UNIX

Implementing File Protection

Once they are installed, the files distributed with XMLink should not be modified except under special circumstances. To prevent inadvertent changes to the files, we recommend that write-access to them be limited to a system administrator or a specially created `prolifics` login, and that general users be allowed only read-access.

Two suggested ways of implementing the above recommendations are:

- (UNIX only) Login as `root` to install the files. After installation is complete, set the permissions so that only `root` can modify the files but all others can read and/or execute them. See `chmod` in your system manual, or type `man chmod` for information on setting permissions.
- Create a dummy login ID (for example, `prolifics`), then log in as that user and perform the installation. This allows whomever has access to the `prolifics` login account to control ownership, permissions, and modifications. This approach accommodates systems for which access to the root account is tightly controlled.

Determining File Location

After deciding who is going to own the XMLink files (`root` or a dummy login ID), determine where they will be installed. Do not change this directory once it is set up because users are likely to embed the directory name in makefiles, shell scripts, and so forth. The default installation directory on UNIX is `/usr/prolifics`. On Windows the default directory is *`C:`*`\Program Files\Prolifics\XMLink`, where *`C`* is the letter of the drive where you are installing XMLink.

How to Install from CD-ROM

Installing XMLink on UNIX requires you to copy the distribution from the delivered media.

1. Log in as `root` or with the login you are using for the installation.
2. At the command line, type the following.

   ```
   mkdir /usr/prolifics
   ```

3. Go to the `/usr/prolifics` directory by typing the following.

   ```
   cd /usr/prolifics
   ```

4. Mount the CD-ROM device as `/cdrom`.
5. In `/usr/prolifics`, to uncompress and extract the contents of the Panther distribution, type the following.

   ```
   zcat < /cdrom/CompressedTarFilename | tar -xvf -
   ```

   For XMLink 3.0, the tar file name is `tconn30.tar.Z`.

When XMLink software is loaded, your regular prompt is displayed.

Installing XMLink on Windows

XMLink is supplied in compressed form on CD-ROM along with a Windows-based setup program.

How to Run the Setup Program

1. Insert the CD-ROM in the appropriate drive.

2. If the setup program does not start automatically, choose Start→Run. In the Run dialog box, type *d*:\`setup` (where *d* is the letter of the drive from which you are installing).

3. Choose to install XMLink. The setup program guides you through the steps to install the software.

CHAPTER

3 Configuring XMLink

This chapter primarily focuses on how to configure XMLink to run in IBM WebSphere, but since XMLink can be used with any J2C 1.5-compliant application server, the environment section can be referenced by all XMLink configurations. If there is additional information on configuring XMLink on other application servers, you can find it in the Release Notes section at http://docs.prolifics.com/docs/tconn/.

Configuration Notes

Before configuring your installation of XMLink, check to see if any of the following information applies to your system.

Libraries for Oracle Tuxedo interaction

There are native and workstation versions in both UNIX and Windows:

- For Oracle Tuxedo 6.5, `TConn6.rar` contains the libraries `libtconn6w.so` (`tconn6w.dll` for Windows) and `libtconn6n.so` (`tconn6n.dll` for Windows) for the workstation client and native client, respectively.

  For more information on using Oracle Tuxedo 6.5 with XMLink, see page A-5, "Using XMLink with Tuxedo 6.5."

- For Oracle Tuxedo 7.1 and later, `TConn.rar` contains the libraries `libtconnw.so` (`tconnw.dll` for Windows) and `libtconnn.so` (`tconnn.dll` for Windows) for the workstation client and native client, respectively.

Note: If you are using a managed environment, only install one of these `rar` files. Since they contain duplicate classes, the `CLASSPATH` setting needs to find the correct version.

Tuxedo Variables

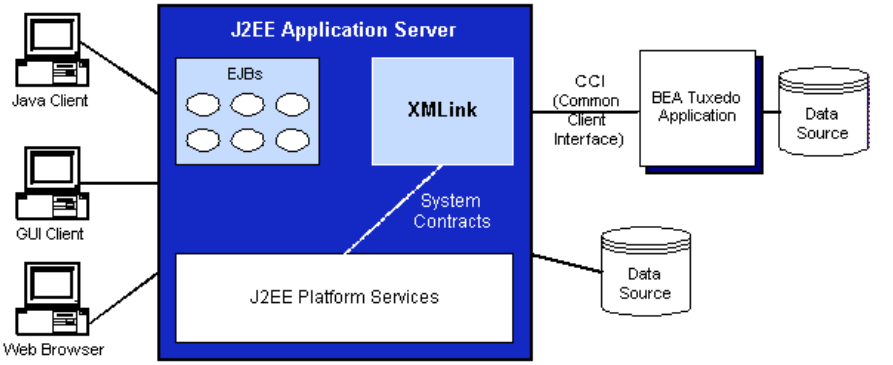
If a Tuxedo variable is set in the environment, it affects all connections and connection factories if not overridden by a corresponding connection factory property.

Managed and Non-Managed Environments

With J2C-compliant resource adapters like XMLink, you have the option of running them in a managed environment or non-managed environment.

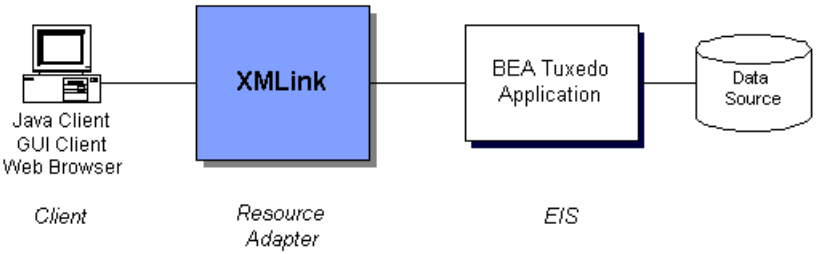
A managed environment would be in a J2EE-compliant application server, such as WebSphere, which supports component-based applications and associated technologies (like EJB, servlets, and JSPs). The application server also provides other services that can be used in conjunction with the resource adapter, such as security, transaction support, and connection pooling. Figure 3-1 (which is also found in Chapter 1) illustrates the use of XMLink in a managed environment.

Figure 3-1 Running XMLink in a Managed Environment



A non-managed environment is like a two-tier application where the application client uses the resource adapter to directly access the EIS, as illustrated in Figure 3-2.

Figure 3-2 Running XMLink in a Non-managed Environment



Managed Environment Settings

For WebSphere application server, there are options for installing your resource adapter archives so that your resource adapters are readily available to your applications. However, problems can occur if local environment settings override the WebSphere application server defaults.

In a managed environment, it is recommended that you use the application servers's settings for any installed resource adapters, such as XMLink. This means that the XMLink class files and libraries will not need to be added to `CLASSPATH` or `PATH` (for UNIX and AIX, `LD_LIBRARY_PATH` or `LIBPATH`)

Installing Resource Adapter Archives

To install resource adapter archives in a WebSphere managed environment, refer to your WebSphere documentation.

Deploying in a Non-Managed Environment

XMLink contains a utility, `TConnTool`, to facilitate deployment in non-managed environments. It supports a set of methods that can be used to build a customized deployment tool. It also includes a `main()` that supports a rudimentary command line interface.

The following describes the command line usage of TConnTool:

```
java com.prolifics.tconn.TConnTool [-deploy] [-remove] [-info]
JndiName

java com.prolifics.tconn.TConnTool [-list]
```

`-deploy`
: Deploys a Connection Factory with the JNDI name, *`JndiName`*, using the properties provided in the file, *`JndiName`*`.properties`.

`-remove`
: Removes a Connection Factory previously deployed by this tool, whose JNDI name is *`JndiName`*.

`-info`
: Displays the property settings for a Connection Factory previously deployed by this tool, whose JNDI name is *`JndiName`*.

`-list`
: Lists the JNDI names of Connection Factories previously deployed by this tool.

For the `-deploy` option, it is expected that the properties file is located in the current directory. This file should contain entries of the form *`name=value`*, such as `WSNADDR=//mymachine:12345`. If the first character on a line is '#', it is a comment. It is expected that components of any path given in *`JndiName`* for `-deploy` are pre-existent. If a path is not given, the root of the context tree is used. For WebSphere 4.0, the `eis` subcontext is recommended.

Note that when the `-list` option is used, `TConnTool` may take several seconds to complete the search. `TConnTool` performs an exhaustive search of the JNDI name space on the local machine.

The following are public methods provided by `TConnTool`, which offer the same functionality as the command line interface:

```
public void deploy(Context ctx, HashMap p, String JNDIname)

public void remove(Context ctx, String JNDIname) throws
NamingException

public HashMap info(Context ctx, String JNDIname) throws
NamingException, Exception

public String[] list(Context ctx, String ctxname)
```

For `deploy()`, the second parameter should contain valid connection factory properties and their settings. Similarly, `info()` returns a HashMap which contains connection factory properties and their settings.

For `list()`, the second parameter is the path for the Context at which to begin the search for deployed connection factories. To begin at the root, pass in an empty string. `list()` returns a String array containing the names of any deployed connection factories that are found.

Configuring the Environment

The following environment variables need to be set in order to use XMLink. In WebSphere, you have the option of specifying some of the environment variables either as a property setting or in the environment.

Note: The configuration settings may differ if you are using Java on the command line with XMLink vs. deploying XMLink in an application server.

Table 3-1 Environment Variables for XMLink with IBM WebSphere 6

Variable	Description	How to Configure
`CLASSPATH`	Specify the location of the following Java files:	environment
■ `tconn.jar`	XMLink classes	needs to be set for applications using XMLink
■ `j2ee.jar`	From the J2EE distribution. Needed for compiling user code. Contains necessary classes for JNDI lookup and J2C reference implementation.	
■ `j2cimpl.jar`	For Connection Factory lookup.	
■ `webcontainer.jar`	This jar is required by WebSphere and contains the following: ■ A plug-in tool that allows debugging functionality ■ Apache classes ■ Event listeners and servlet actions (requests/responses/filters) ■ Debugger and compiler for JSPs ■ XML Configuration	
■ `xerces.jar` or equivalent	For the XML interface in XMLink, you need a XML parser, such as Xerces. You can set this as a `-d` option when using command-line Java. WebSphere installs `xerces.jar` in the `WAS_HOME/lib` directory	
`JAVA_HOME`	Specify the location of your Java JDK or JRE installation.	environment

Table 3-1 Environment Variables for XMLink with IBM WebSphere 6

Variable	Description	How to Configure
`LD_LIBRARY_PATH` or `LIBPATH`	On UNIX, specify the location of the following shared libraries for non-managed environments: ■ `libtconn.so` (or its equivalent) from the XMLink distribution. For the names and uses of libraries, refer to page 3-1, "Libraries for Oracle Tuxedo interaction."	environment
`PATH`	On Windows, specify the location of the following DLLs for non-managed environments: ■ `tconn.dll` (or its equivalent) from the XMLink distribution. For the names and uses of DLLs, refer to page 3-1, "Libraries for Oracle Tuxedo interaction."	environment

Set the following Tuxedo variables:

Table 3-2 Tuxedo Variables

Variable	Description	How to Configure
`TUXCONFIG`	Specify the full path name of the binary `TUXCONFIG` file for native clients.	■ environment ■ property of a connection factory
`TUXDIR`	Specify the location of the Oracle Tuxedo installation.	
`WSNADDR`	If your Oracle Tuxedo client is a workstation client (not native), specify the list of one or more network addresses of the workstation listeners the client wants to contact, matching the addresses specified in the application configuration file. The setting contains the host machine and port number, for example: `//myhost:3445`	■ environment ■ property of a connection factory

Setting XMLink Properties

In WebSphere, you set properties in the Administrative Console. Some of these properties correspond to settings in an Oracle Tuxedo configuration.

Setting JVM Properties

`XMLink.tconn`

For XMLink 3.0, this JVM property has been supplanted by the `ConnectionType` property on the resource adapter. However, this property can still be specified either on the JVM or on the command line; however, there is no longer a default setting.

Specify the version of the XMLink Tuxedo libraries. There are native and workstation versions for Tuxedo 6.5 and Tuxedo 7.1+ in both UNIX and Windows. Set the property to one of the following values:

- `tconn6n` (Tuxedo 6.5 native)
- `tconn6w` (Tuxedo 6.5 workstation)
- `tconnn` (Tuxedo 7.1+ native)
- `tconnw` (Tuxedo 7.1+ workstation)

To specify the library on the command line, the syntax is:

```
java -DXMLink.tconn=tconnw
```

Setting Properties on a Resource Adapter

Among the properties in this section, `FIELDTBLS`, `FIELDTBLS32`, `FLDTBLDIR`, `FLDTBLDIR32` and `ConnectionType` may only be set here, since the values for these properties must remain the same for all XMLink Connection Factories of the current JVM process. The other Resource Adapter properties may act as defaults for multiple connection factories, and may be overridden for individual connection factories.

`ClientName`
: Authentication information. See Tuxedo docs for `tpinit()`. Overridden by `ClientName` connection factory property.

`ConnectionRetries`
: The number of additional times XMLink will try to establish a connection to the Tuxedo server after a failed initial attempt. Also, the number of times XMLink will try to reestablish a lost connection to the Tuxedo server during an attempted service call. Overridden by `ConnectionRetries` connection factory property. Default is 0.

`ConnectionRetryInterval`
: The delay in milliseconds between attempts to establish or reestablish a connection to the Tuxedo server. Overridden by `ConnectionRetryInterval` connection factory property. Default is 0, or as brief a delay as possible.

`ConnectionType`
: Either `native` or `workstation` are allowed values. Selects which of two DLLs or shared libraries to use when the resource adapter is started, where one is provided for a Tuxedo native connection and another is provided for a Tuxedo workstation connection. Overridden by the setting for the `XMLink.tconn` System property for the JVM, which may be assigned to a specific shared library or DLL by name.

`Data`
: Authentication information. See Tuxedo docs for `tpinit()`. Overridden by `Data` connection factory property.

`FIELDTBLS`
: Overrides `FIELDTBLS` Tuxedo environment variable. Must be unchanged for all XMLink resource adapters running in the same JVM.

`FIELDTBLS32`
: Overrides `FIELDTBLS32` Tuxedo environment variable. Must be unchanged for all XMLink resource adapters running in the same JVM.

`FLDTBLDIR`
: Overrides `FLDTBLDIR` Tuxedo environment variable. Must be unchanged for all XMLink resource adapters running in the same JVM.

`FLDTBLDIR32`
: Overrides `FLDTBLDIR32` Tuxedo environment variable. Must be unchanged for all XMLink resource adapters running in the same JVM.

`GroupName`
: Authentication information. See Tuxedo docs for `tpinit()`. Overridden by `GroupName` connection factory property.

`InteractionRetries`
: The number of additional times XMLink will try a Tuxedo service call during Interaction.execute(), following an attempt that fails due to a connection error or Tuxedo server system error. Overridden by `InteractionRetries` connection factory property. Default is 0.

`InteractionRetryInterval`
: The delay in milliseconds between Tuxedo service call attempts during Interaction.execute(). Overridden by `InteractionRetryInterval` connection factory property. Default is 0, or as brief a delay as possible.

`Password`
: Authentication information. See Tuxedo docs for `tpinit()`. Overridden by `Password` connection factory property.

`TUXCONFIG`
: Overrides `TUXCONFIG` Tuxedo environment variable. Overridden by `TUXCONFIG` connection factory property.

`ULOGPFX`
: Overrides `ULOGPFX` Tuxedo environment variable. Overridden by `ULOGPFX` connection factory property, when logging takes place after a connection is established.

`UserName`
: Authentication information. See Tuxedo docs for `tpinit()`. Overridden by `UserName` connection factory property.

`WSENVFILE`
: Overrides `WSENVFILE` Tuxedo environment variable. Overridden by `WSENVFILE` connection factory property.

`WSNADDR`
: Overrides `WSNADDR` Tuxedo environment variable. Overridden by `WSNADDR` connection factory property.

Setting Properties on a Connection Factory

The following list of properties can be configured as needed for each connection factory that is created.

`ClientName`
: Authentication information. See Tuxedo docs for `tpinit()`. Overrides ClientName property of resource adapter.

`ConnectionRetries`
: The number of additional times XMLink will try to establish a connection to the Tuxedo server after a failed initial attempt. Also, the number of times XMLink will try to reestablish a lost connection to the Tuxedo server during an attempted service call. Overrides `ConnectionRetries` property of resource adapter. Default is 0.

`ConnectionRetryInterval`
: The delay in milliseconds between attempts to establish or reestablish a connection to the Tuxedo server. Overrides `ConnectionRetryInterval` property of resource adapter. Default is 0, or as brief a delay as possible.

`Data`
: Authentication information. See Tuxedo docs for `tpinit()`. Overrides `Data` property of resource adapter.

`GroupName`
: Authentication information. See Tuxedo docs for `tpinit()`. Overrides `GroupName` property of resource adapter.

`InteractionRetries`
: The number of additional times XMLink will try a Tuxedo service call during `Interaction.execute()`, following an attempt that fails due to a connection error or Tuxedo server system error. Overrides `InteractionRetries` property of resource adapter. Default is 0.

`InteractionRetryInterval`
: The delay in milliseconds between Tuxedo service call attempts during `Interaction.execute()`. Overrides `InteractionRetryInterval` property of resource adapter. Default is 0, or as brief a delay as possible.

`Password`
: Authentication information. See Tuxedo docs for `tpinit()`. Overrides `Password` property of resource adapter.

`TUXCONFIG`
: Overrides `TUXCONFIG` Tuxedo environment variable and `TUXCONFIG` property of resource adapter.

`ULOGPFX`
: Overrides `ULOGPFX` Tuxedo environment variable and `ULOGPFX` property of resource adapter, when logging takes place after a connection is established.

`UserName`
: Authentication information. See Tuxedo docs for `tpinit()`. Overrides `UserName` property of resource adapter.

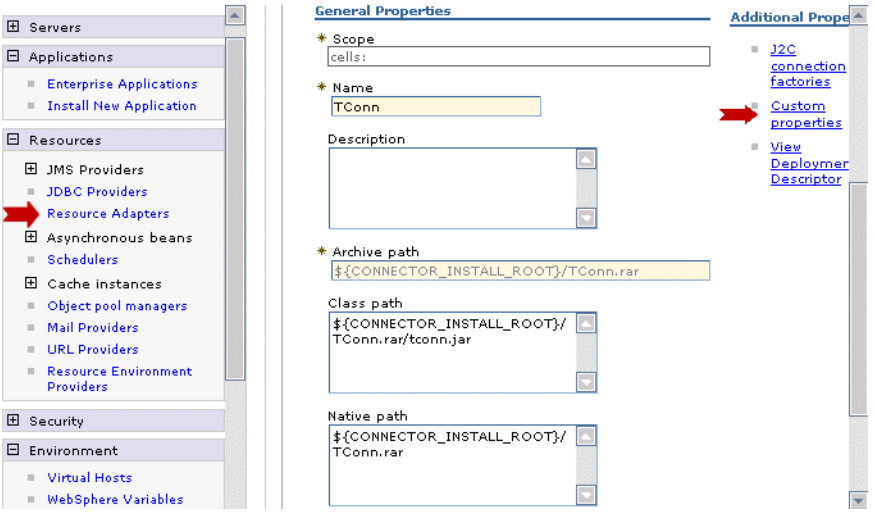
`WSENVFILE`
: Overrides `WSENVFILE` Tuxedo environment variable and overrides `WSENVFILE` property of resource adapter.

`WSNADDR`
: Overrides `WSNADDR` Tuxedo environment variable and overrides `WSNADDR` property of resource adapter.

Sample Configuration

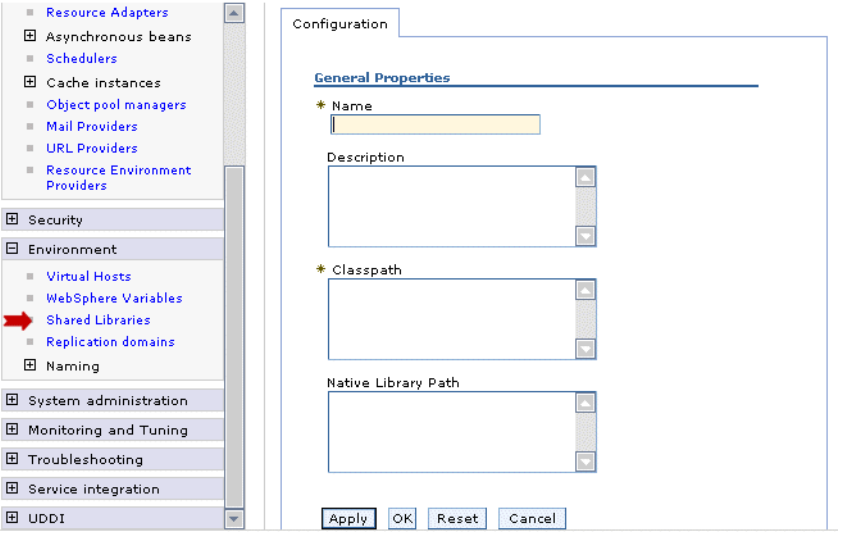
The following screens from the WebSphere Administrative Console illustrate a sample configuration. For more information about using the WebSphere Administrative Console, refer to the IBM WebSphere documentation.

First, the resource adapter was installed and its properties set.

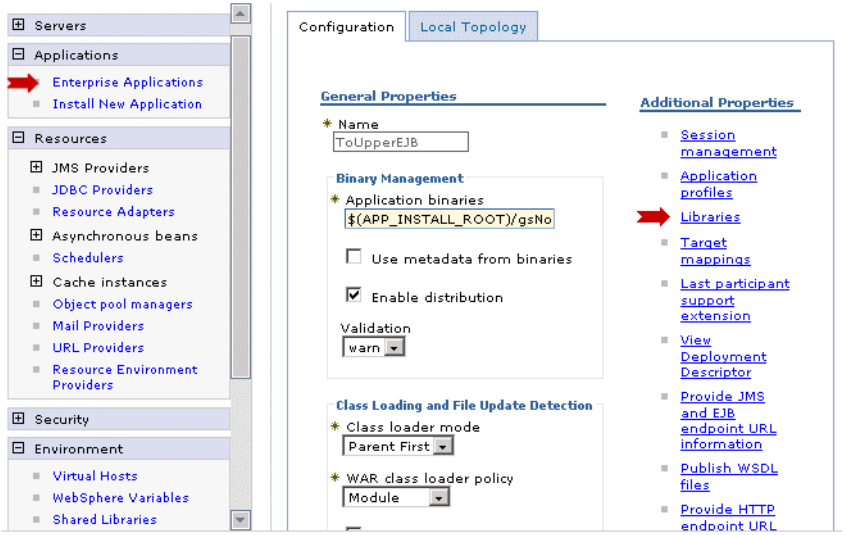


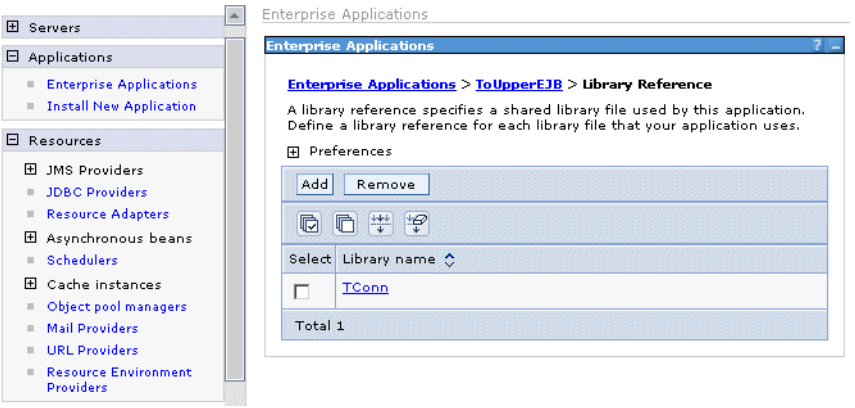
Note: When configuring an XMLink resource adapter in WebSphere 6, make sure that the native path is set to the same location as the archive path. The Class Path should contain the path to `tconn.jar`. For `TConn.rar`, this would typically be `${CONNECTOR_INSTALL_ROOT}/TConn.rar/tconn.jar`.

Next, shared library settings were configured that can later be associated with an application.



Then, the application and its libraries and other settings were configured.





CHAPTER

4 Using Java to Call Services

This chapter describes how to write Java clients to call services in an Oracle Tuxedo application. Alternatively, you can use XML to exchange data as described in Chapter 5, “Using XML to Call Services.”

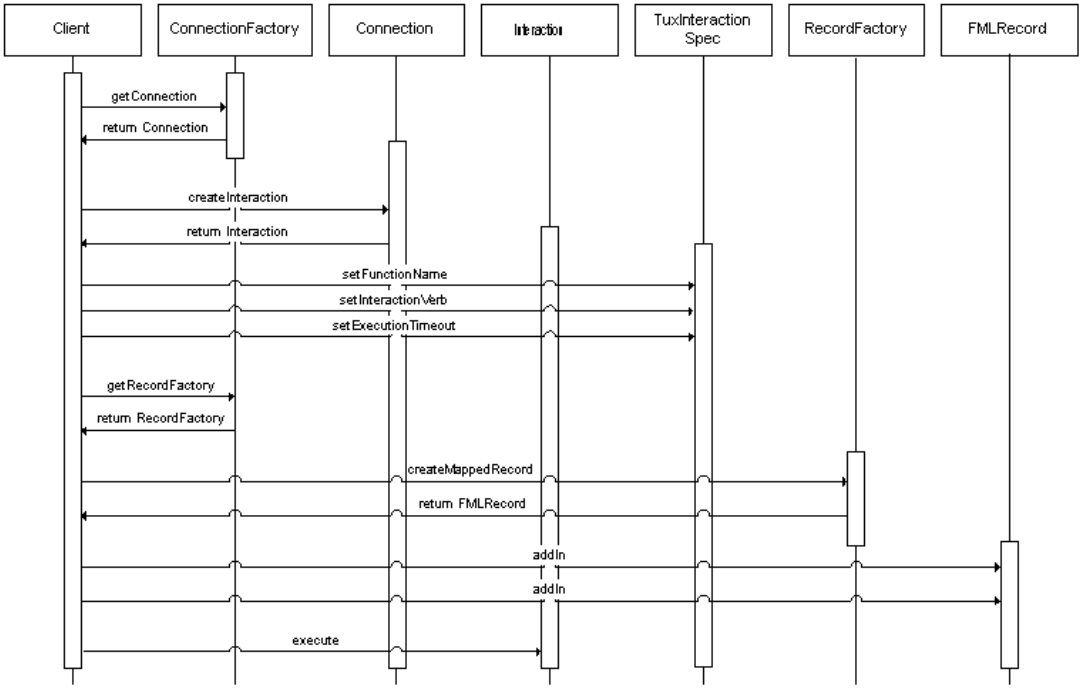
For Java documentation of the XMLink interfaces, refer to the Javadoc portion of the online documentation.

The major topics in this chapter are:

- Getting an Oracle Tuxedo Connection
- Managing Oracle Tuxedo Transactions
- Calling Oracle Tuxedo Services
- Working with Application Data

The following diagram illustrates the process described in this chapter for using the CCI interface to execute a Tuxedo service which uses an FML buffer. There are additional record types corresponding to the Oracle Tuxedo buffer types. For more information on record types, refer to page 4-11, "Working with Record Objects."

Figure 4-1 Overview of the steps to create an FML record in XMLink



Getting an Oracle Tuxedo Connection

To get a connection to your Oracle Tuxedo application, you need to create an instance of `ConnectionFactory`. This is the starting point for all interactions with XMLink. You then call the method `getConnection` on your `ConnectionFactory` instance, which returns an object of type `Connection`.

Connection factories can have any JNDI binding name you choose. The JDNI binding name is specified when you created the connection factory and can be the same as the connection factory name.

Example: Connecting with a Java Client

In XMLink 2.0, the following example connects to a Java client and performs JNDI lookup:

```
// get an initial JNDI naming context
javax.naming.Context initctx = new javax.naming.InitialContext();

// do JNDI lookup to get connection factory
// lookup doesn't return a ConnectionFactory object,
// so a cast is needed
javax.resource.ConnectionFactory cxf =
    (javax.resource.ConnectionFactory)
    initctx.lookup("ConnectionFactoryName");

// where "ConnectionFactoryName" is the JNDI binding path of a
// predeployed connection factory

// get a connection
Connection cx = cxf.getConnection();
```

Understanding the ConnectionFactory Interface

The following code shows the interface `ConnectionFactory`. XMLink's implementation class for `ConnectionFactory` is `ConnectionFactoryImpl`.

```
public interface ConnectionFactory
    extends java.io.Serializable, javax.resource.Referenceable
{
    public Connection getConnection ()
        throws ResourceException;
    public Connection getConnection(ConnectionSpec properties)
        throws ResourceException;

    public RecordFactory getRecordFactory()
        throws ResourceException;

    public ResourceAdaptorMetadata getMetaData()
        throws ResourceException;
}
```

Note that objects of class `ConnectionFactory` support the basic Java interfaces `Serializable` and `Referenceable`. Refer to the Java SDK documentation for descriptions of those interfaces.

Supplying Connection Parameters

Calling `getConnection` on an object of type `ConnectionFactory` returns an object of class `Connection`, which represents a connection to the Oracle Tuxedo application.

Calling `getConnection` with the `ConnectionSpec` parameter allows a client to provide log-in information needed to log in to Oracle Tuxedo. In XMLink, the implementation class for `ConnectionSpec` is `TuxConnectionSpec`, which provides `set` and `get` methods for the following properties:

- `UserName`
- `ClientName`
- `Password`
- `GroupName`
- `Data`

The above properties correspond to members of Oracle Tuxedo's `TPINIT` data structure. Refer to Oracle Tuxedo documentation for `tpinit(3c)` for more information.

For container-managed sign-on, use `getConnection` without any parameters. The authentication information in this case is supplied by your application server.

The following code shows the interface `Connection`. XMLink's implementation class for `Connection` is `ConnectionImpl`.

```
public interface Connection
{
    public Interaction createInteraction()
        throws ResourceException;
    public LocalTransaction getLocalTransaction()
        throws ResourceException;
    public ConnectionMetaData getMetaData()
        throws ResourceException;
    public ResultSetInfo getResultSetInfo()
        throws ResourceException;
    public void close()
        throws ResourceException;
}
```

Calling `createInteraction` on a `Connection` instance creates an `Interaction` instance associated with the `Connection` instance. A single `Connection` instance can be associated with multiple `Interaction` instances. An `Interaction` instance is what is used to access Oracle Tuxedo services. For a description of the methods supported by `Interaction` instances, refer to page 4-7, "Calling Oracle Tuxedo Services."

Calling `getMetaData` on a `Connection` instance returns an object containing information about the Oracle Tuxedo connection represented by that `Connection` instance. Such an object supports the following methods:

```
public String getEISProductName()
public String getEISProductVersion()
public String getUserName()
```

Getting Information about XMLink

The method `getMetaData` provides information about the XMLink product. Calling `getMetaData` returns an object of class `ResourceAdapterMetadata`. Such objects support the following methods:

```
    public String getAdapterName()
    public String getAdapterShortDescription()
    public String getAdapterVendorName()
    public String getAdapterVersion()
    public String getSpecVersion()
    public String getInteractionSpecsSupported()
    public boolean supportsExecuteWithInputAndOutputRecord()
    public boolean supportsExecuteWithInputRecordOnly()
```

```
public boolean supportsLocalTransactionDemarcation()
```

The return values for these methods provide the name and version information for XMLink.

Calling getRecordFactory

The `getRecordFactory` method of the `ConnectionFactory` interface returns the `RecordFactory` interface. This object can be used to generate `Record` instances that, in turn, can be used to hold input data for Oracle Tuxedo services. For information about working with `Record` instances, refer to page 4-9, "Working with Application Data."

Specifying Transaction Access

Calling the `getLocalTransaction` method of the `ConnectionFactory` interface returns an object that allows access to Oracle Tuxedo transaction management functions. Each `Connection` instance can be associated with only a single `LocalTransaction` instance. For a description of the methods supported by the `LocalTransaction` interface, refer to page 4-6, "Managing Oracle Tuxedo Transactions."

Managing Oracle Tuxedo Transactions

Access to Oracle Tuxedo transaction demarcation functions is achieved through objects of type `LocalTransaction`. You get such an object by calling the `getLocalTransaction` method on a `Connection` instance. A `Connection` instance can only be associated with one `LocalTransaction` instance at a time.

A `LocalTransaction` instance supports the following methods:

```
public void begin()
public void commit()
public void rollback()
```

The method `begin` corresponds to the Oracle Tuxedo function `tpbegin`. The method `commit` corresponds to the Oracle Tuxedo function `tpcommit`. The method `rollback` corresponds to the Oracle Tuxedo function `tpabort`.

Warning: Transaction support is available in Oracle Tuxedo version 7.1 and higher.

Calling Oracle Tuxedo Services

Access to Oracle Tuxedo services is provided by objects of type `Interaction`. To create `Interaction` instances, call the `createInteraction` method on a `Connection` instance. Each `Interaction` instance retains an association with the `Connection` instance it was created by.

Understanding the Interaction Interface

The following code defines the `Interaction` interface:

```
public interface Interaction
{
    public void close() throws ResourceException;

    public boolean execute(InteractionSpec ispec,
        Record input,
        Record output) throws ResourceException;

    public Record execute(InteractionSpec ispec,
        Record input) throws ResourceException;
}
```

The `getConnection` method returns the `Connection` that the Interaction instance is associated with.

The `close` method terminates the `Interaction` instance. Calling this method also frees all Tuxedo record buffers used during the interaction. However, the Java record still has a copy of the data, so it may in fact be used later for a different interaction.

There are two versions of the `execute` method:

- One takes two `Record` instances as arguments: an input `Record` and an output `Record`. It feeds the data in the input `Record` to Oracle Tuxedo and modifies the output `Record` provided in the third argument to represent the return buffer from the Oracle Tuxedo service.
- The other version of the `execute` method provides only an input `Record` as an argument.

In either version of the `execute` method, the Oracle Tuxedo service being called is specified by the first parameter, which takes an object of type `InteractionSpec`.

For a code example, see page 4-14, "Example: Calling Services using Java."

Understanding the TuxInteractionSpec Interface

For XMLink, a class called `TuxInteractionSpec` implements `InteractionSpec` and supports the property set needed by XMLink. So in order to call an Oracle Tuxedo service, you must first create a `TuxInteractionSpec` object, and then set its properties to determine which Oracle Tuxedo service it specifies.

`TuxInteractionSpec` objects support the following properties:

- `FunctionName`—Specifies the name of the Oracle Tuxedo service.
- `InteractionVerb`—Specifies whether the service is asynchronous or synchronous using the following integer values:
  - `SYNC_SEND`—Specify for asynchronous Oracle Tuxedo services (to be called with `tpacall` and the `TPNOREPLY` flag set).
  - `SYNC_SEND_RECEIVE`—Specify for synchronous Oracle Tuxedo services (called with `tpcall`). This is the default setting.

    (The `InteractionSpec` interface also defines a third possible value for `InteractionVerb`: `SYNC_RECEIVE`. XMLink doesn't support `SYNC_RECEIVE`. An `execute` method invoked with an `InteractionSpec` parameter that has its `InteractionVerb` property set to `SYNC_RECEIVE` throws an exception.)
- `ExecutionTimeout`—Not yet implemented.

The following methods work with these properties:

```
public void setFunctionName(string name)
public string getFunctionName()
public void setInteractionVerb(int mode)
public int getInteractionVerb()
public void setExecutionTimeout(int milliseconds)
public int getExecutionTimeout()
```

After creating a new `InteractionSpec` instance:

- Use its `set` methods to identify it as representing a particular Oracle Tuxedo service.
- Provide the modified `InteractionSpec` instance as the first argument to a call to an `Interaction` instance's `execute` method.

For a code example, see page 4-14, “Example: Calling Services using Java.”

Working with Application Data

Oracle Tuxedo applications send and receive data using typed buffers, which allows the Oracle Tuxedo platform to transfer data between different operating systems. XMLink automatically converts data provided to it in the form of a Java `Record` instance to the appropriate Oracle Tuxedo buffer type for the service being called, and then converts the return data into a Java `Record` instance.

XMLink defines four record classes corresponding to the Oracle Tuxedo buffer types and one class for Prolifics’s Panther for JetNet buffer type.

Table 4-1

	XMLink record types
Oracle Tuxedo buffer types	
`FML`	`FMLRecord`
`FML32`	`FML32Record`
`STRING`	`StringRecord`

Table 4-1

	XMLink record types
CARRAY	CArrayRecord
Panther for JetNet buffer type	
JAMFLEX	JAMFLEXRecord

To call a Tuxedo service, you must create a Record object of the type corresponding to the buffer type the service takes as its argument. You get Record instances by calling the methods of a RecordFactory interface.

Understanding the RecordFactory Interface

Calling getRecordFactory on a ConnectionFactoryImpl instance returns an instance of RecordFactoryImpl. RecordFactoryImpl objects implement the RecordFactory interface.

The following code defines the RecordFactory interface:

```
public interface RecordFactory

  public MappedRecord createMappedRecord(String recordName)
    throws ResourceException;

  public IndexedRecord createIndexedRecord(String recordName)
    throws ResourceException;
```

XMLink's RecordFactoryImpl class also supports another method:

```
  public Record createRecord(String recordName)
    throws ResourceException;
```

To create an FMLRecord or FML32Record instance, use the createMappedRecord method of a RecordFactory instance. The createMappedRecord method takes a string argument used to identify what kind of record should be created. Use the strings "FML", "FML32" or "JAMFLEX" as the argument to the createMappedRecord method to create an FMLRecord, FML32Record or JAMFLEXRecord instance, respectively.

To create a `StringRecord` or `CArrayRecord` instance, use the `createRecord` method of a `RecordFactory` interface. As with the `createMappedRecord` method, the method's string argument is used to determine which type of record to return. Use the strings `"STRING"` and `"CARRAY"` as the argument to the `createRecord` method to create a `StringRecord` or `CArrayRecord` instance, respectively.

The `createMappedRecord` method returns a `MappedRecord` and the `createRecord` returns a `Record`. Since what you actually want are instances of `FMLRecord`, `FML32Record`, `StringRecord`, `CArrayRecord` or `JAMFLEXRecord`, you will have to cast the return values from `createMappedRecord` and `createRecord` to the appropriate types.

For example:

```
// create an FML32Record instance, using a previously
// acquired RecordFactory instance called rcf
FML32Record fml32r = (FML32Record)
    rcf.createMappedRecord("FML32");

// create a StringRecord instance
StringRecord strr = (StringRecord) rcf.createRecord("STRING");
```

XMLink client code will not typically need to use the `createIndexedRecord` method. `RecordFactoryImpl` objects do implement this method to create the `ArrayRecord` objects returned by the `getField` method of `FMLRecord` and `FML32Record` objects. For more information, see page 4-12, “FMLRecord and FML32Record Objects.”

Working with Record Objects

The following code defines the `Record` interface:

```
public interface Record extends Cloneable
{
    public String getRecordName();
    public void setRecordName(String Name);

    public void setRecordShortDescription(String description);
    pubic String getRecordShortDescription();

    public boolean equals(Object other);
    public int hashcode();
```

```
    Public Object clone() throws CloneNotSupportedException;
}
```

The `Record` objects supported by XMLink, objects of class `FMLRecord`, `FML32Record`, `CArrayRecord`, `StringRecord` and `JAMFLEXRecord` all support the methods listed above. In addition they also support methods specific to themselves, and client code that accesses them will primarily do so by means of those specific methods.

FMLRecord and FML32Record Objects

`FMLRecord` and `FML32Record` objects are instances of `MappedRecord`, and therefore implement the methods of the `Map` interface in addition to those of `Record`. Specifically, these classes extend the standard Java class `HashMap`.

However, underlying each instance of `FMLRecord` or `FML32Record` is a Tuxedo `FML` buffer or `FML32` buffer. `FML` buffers are made up of "fields". Client code should **NOT** use the methods of the `Map` interface to modify the contents of an `FMLRecord` or an `FML32Record`, because to do so will put the `Record` out of sync with the Oracle Tuxedo `FML` buffer that it represents. To update the contents of an `FMLRecord` or an `FML32Record`, use the following method, that is specific to these classes, and define interaction at the level of `FML` fields:

```
public void addIn(String name, String value)
```

The `addIn` method is used to populate the fields in an `FML` buffer. The `name` parameter is the name of the field to which a value is to be added. The value is added as a new occurrence to the field. To add multiple occurrences to a field, call `addIn` repeatedly with a single field name.

Using the following `addIn` method allows you to create embedded FML for Oracle Tuxedo versions 7.1+ which contain support for this feature:

```
public void addIn(String name, FML32Record value)
```

Note that the field values are all input as strings. The underlying `FML` buffer will be populated with whatever data types are appropriate to the `FML` file definition, based on the field names. If any conversion is needed, it is performed by Tuxedo.

Sample: Getting FML Data

The following sample illustrates one method of getting data after calling `execute()`.

Warning: The `getField` method is for internal use only. Use this example as a basis for getting data from FML buffers.

```
//Call tuxedo service

rcout = (FMLRecord)iact.execute(tuxl,rc);

//Get the fields from the Record object.

Iterator it = rcout.entrySet().iterator();
  while ( it.hasNext()) {
     Map.Entry me = (Map.Entry)it.next();
     String fieldName = (String)me.getKey();
     IndexedRecord fieldValue = (IndexedRecord)me.getValue();
     Iterator it2 = fieldValue.iterator();
     System.out.println(fieldName);
     while (it2.hasNext())
       {
         System.out.println(" " + it2.next());
       }
   }
```

Note: In this example, if `rcout` were an `FML32Record`, it might contain embedded FML. In that case, `it2.next()` might return an `FML32Record`, rather than a string and different processing would be required.

CArrayRecord and StringRecord Objects

In addition to the basic methods of the `Record` interface, as listed above, `CArrayRecord` and `StringRecord` objects support the following methods:

```
public String getContents()
public void addIn (String name, String value)
```

The `getContents` method returns the contents of the `Record`.

The `addIn` method takes two String arguments, the value of the second argument will be appended to the `Record`'s contents. The first argument is ignored. A `Record` can be filled either by one single call to `addIn` or by a series of such calls.

For `CArrayRecord`, the output record used for `InteractionImpl.execute()` must contain initial data. For example:

```
CArrayRecord out = (CArrayRecord)rcf.createRecord("CARRAY");

out.addIn("", "|");
```

Character Encoding Support

Java strings are in Unicode. By default, XMLink passes to Tuxedo only the low order byte of each Unicode character. For characters in English and other European languages, this is sufficient since only the low order byte is significant.

Since other languages may need additional support, each of XMLink's `Record` classes offers `setEncoding()` and `getEncoding()` methods. The default encoding is `ISO-8859-1`, which is a single byte per character encoding. A multibyte encoding, such as `UTF8`, can be used for foreign characters. The Tuxedo server must be able to support and decode/encode according to the specified encoding setting.

Example: Calling Services using Java

Here is an sample outline of client code combining the connection code from earlier in the chapter with an example of interacting with a `Connection` instance to call a Tuxedo service:

```
// get an initial JNDI naming context
javax.naming.Context initctx = new javax.naming.InitialContext();

// do JNDI lookup to get connection factory
// lookup doesn't return a ConnectionFactory object,
// so a cast is needed
javax.resource.ConnectionFactory cxf =
    (javax.resource.ConnectionFactory)
    initctx.lookup("ConnectionFactoryName");

// where "ConnectionFactoryName" is the JNDI binding path of a
// predeployed connection factory

// get a connection
Connection cx = cxf.getConnection();

// cxf represents previously acquired ConnectionFactory instance
// first get a LocalTransaction instance
    LocalTransaction ltx = cx.getLocalTransaction();

// mark the beginning of a Tuxedo transaction
    ltx.begin();
```

```
// get an Interaction instance
    Interaction iact = cx.createInteraction();

// create a new InteractionSpec object
    TuxInteractionSpec tux1 = new TuxInteractionSpec;

// set the properties of the InteractionSpec instance
// note that InteractionVerb defaults to SYNC_SEND_RECEIVE,
// so the second line below isn't really necessary
    tux1.setFunctionName("TUXServiceName");
    tux1.setInteractionVerb(SYNC_SEND_RECEIVE);
    tux1.setExecutionTimeout(1000)

// get a RecordFactory instance
    RecordFactory rcf = cxf.getRecordFactory()

// create a Record Instance to hold input data
    FMLRecord rc = (FMLRecord) rcf.createMappedRecord("FML");

// Populate Record instance with data
    rc.addIn ("FirstField", "value1");
    rc.addIn ("FirstField", "value2");
    rc.addIn ("FirstField", "value3");
    rc.addIn ("SecondField", "anothervalue");
    // and so forth

// call the TUXEDO service
    MappedRecord ret = iact.execute(tux1, rc);

// call other services in the transaction by creating
// new InteractionSpec instances and input records
// and calling execute as needed

// mark the end of the TUXEDO transaction
    ltx.commit();

//close the InteractionInstance
    iact.close();
```

CHAPTER

5 Using XML to Call Services

To facilitate support for data exchange using XML, XMLink supports an alternate style of access to Oracle Tuxedo services. In this case, XML input is received by an instance of `XMLConnector`.

If you are writing server-side Java and want to use the XML invocation style, you can create an instance of `XMLConnector` and directly call its `process` method. XMLink also includes an EJB wrapper, called `TConnXML`, that acts as a facade to `XMLConnector`. For more information, see page 5-8, "Using an EJB to Input XML." You will typically use the `TConnXML` EJB to invoke Oracle Tuxedo services from an EJB client.

If you are using XMLink in a distributed server-side environment, using XMLink's `TConnXML` EJB allows you to easily access your Tuxedo services even when XMLink is installed on a remote application server.

Both the input `String` to `TConnXML.process` and the `InputStream` argument to `XMLConnector.process` must be in XML format and must conform to the `tconn.dtd` document type definition.

For Java documentation of the XMLink interfaces, refer to the Javadoc portion of the online documentation.

Inputting Data Using XML

`XMLConnector` objects support a method `process` that takes the following parameters:

- A `ConnectionFactory` object as its first parameter
- An `InputStream` as its second parameter
- An `OutputStream` as its third parameter

The contents of the `InputStream` parameter must be in XML format and must conform to the `tconn.dtd` document type definition.

The following is the document type definition for input to the `process` method:

```
<!-- DTD for TUXEDO service requests -->

<!-- Basic elements -->
<!ELEMENT agenda (connection?, (servicecall | transaction)*)>
<!ELEMENT transaction (servicecall*)>
<!ELEMENT connection EMPTY>
<!ELEMENT servicecall
    (FMLRecord | FML32Record | STRINGRecord | CARRAYRecord |
       JAMFLEXRecord)>
<!ELEMENT FMLRecord (field*)>
<!ELEMENT FML32Record (field*)>
<!ELEMENT STRINGRecord (data)>
<!ELEMENT CARRAYRecord (data)>
<!ELEMENT JAMFLEXRecord (field*)>
<!ELEMENT data (#PCDATA)>
<!ELEMENT field (#PCDATA)>

<!-- Attributes for connection element -->
<!ATTLIST connection username CDATA #REQUIRED>
<!ATTLIST connection clientname CDATA #REQUIRED>
<!ATTLIST connection password CDATA #REQUIRED>
<!ATTLIST connection groupname CDATA #REQUIRED>
<!ATTLIST connection data CDATA #REQUIRED>

<!-- Attributes for servicecall element -->
<!ATTLIST servicecall name CDATA #REQUIRED>
```

```
<!ATTLIST servicecall mode (SYNCH | ASYNCH) "SYNCH">
<!ATTLIST servicecall mode timeout CDATA "0">

<!-- Attributes for records -->
<!ATTLIST FMLRecord encoding CDATA #IMPLIED>
<!ATTLIST FML32Record encoding CDATA #IMPLIED>
<!ATTLIST STRINGRecord encoding CDATA #IMPLIED>
<!ATTLIST CARRAYRecord encoding CDATA #IMPLIED>
<!ATTLIST JAMFLEXRecord encoding CDATA #IMPLIED>

<!-- Attributes for field element -->
<!ATTLIST field name CDATA #REQUIRED>
```

Elements in the XML Input DTD

Agenda Element

Each XML document is delimited by an `agenda` element. The `agenda` is a list of the service requests to be made to the Oracle Tuxedo application. The `agenda` can optionally contain a single `connection` element. If there is a `connection` element, it must be the first element in the agenda.

Connection Element

A `connection` is an empty element with the attributes `username`, `clientname`, `password`, `groupname` and `data`. If a `connection` element is included in an `agenda`, the service requests in the `agenda` are made using a connection created according to the information provided as the `connection`'s attributes. If no `connection` element is provided, the service requests use a "default" connection type.

Transaction and Servicecall Elements

Other than the `connection` element, an `agenda` can contain any number of `transaction` elements and `servicecall` elements. `transaction` elements themselves can contain any number of `sevicecall` elements. So an `agenda` can contain service calls grouped into transactions, plus service calls that are independent of any transaction.

`transaction` elements have no attributes. They merely serve to group `servicecall` elements.

servicecall elements have the following attributes:

- name—Name of the service call being requested
- mode—One of the two values, SYNCH or ASYNCH, to indicate whether the service is synchronous or asynchronous
- timeout—Not yet implemented

Each servicecall element (whether inside a transaction element or not) must contain a single element. This can be either an FMLRecord, an FML32Record, a JAMFLEXRecord, a STRINGRecord, or a CARRAYRecord element. This element contains the input data for the service.

FMLRecord, FML32Record, and JAMFLEXRecord elements can contain any number of field elements. STRINGRecord and CARRAYRecord elements must each contain a single data element.

Field Elements

field records have a name attribute. This is the name of the field. More than one field element can have the same name. If a field element has the same name as another field element, then its contents will represent another occurrence of the field in the FML buffer.

Data Elements

A data element has no attributes, it simply delimits the data to be packaged into a STRING or CARRAY buffer.

Example: Using XML Input

The following is an example of a series of service calls invoked through the XML interface:

```
<agenda>
<connection username="managerxml"
            clientname="manager"
            password="password1"
            groupname=""
            data=""/>
```

```
<transaction>
  <servicecall name="FINDEMP">
    <FMLRecord>
      <field name=”employee_ssn”>111221111</field>
      <field name="last_name">Jones</field>
      <field name="first_name">Fred</field>
      <field name="dept_id">10</field>
    </FMLRecord>
  </servicecall>
</transaction>
</agenda>
```

When a request for Oracle Tuxedo services is received, the XML `InputStream` will be parsed, and the `servicecall` requests will be issued in the order in which they occur in the file. When a `transaction` element is encountered, a `tpbegin` is issued. When the `transaction` end tag is encountered, a `tpcommit` is issued. If any of the `servicecall` elements between the transaction demarcations returns an error, a `tpabort` will be issued.

Returning Data in XML

The `process` method fills the `OutputStream` with the return data from the service requests specified in the `InputStream`. The `OutputStream`'s contents will also be in XML format, and will correspond to the `tconnoutput.dtd` document type definition.

The following is the document type definition for the output from a `process` call.

```
<!-- DTD for output from TUXEDO service requests -->

<!-- Basic elements -->
<!ELEMENT resultset ((returndata | error | xactionmsg)*)>
<!ELEMENT returndata
       (FMLRecord | FML32Record | STRINGRecord | CARRAYRecord |
        JAMFLEXRecord, UserReturnCode?)>
<!ELEMENT FMLRecord (field*)>
<!ELEMENT FML32Record (field*)>
<!ELEMENT STRINGRecord (data)>
<!ELEMENT CARRAYRecord (data)>
<!ELEMENT JAMFLEXRecord (field*)>
<!ELEMENT UserReturnCode (#PCDATA)>
```

```
<!ELEMENT error (#PCDATA)>
<!ELEMENT xactionmsg (EMPTY)>
<!ELEMENT field (#PCDATA)>
<!ELEMENT data (#PCDATA)>

<!-- Attributes for returndata element -->
<!ATTLIST returndata servicename CDATA #REQUIRED>

<!-- Attributes for error element -->
<!ATTLIST error servicename CDATA #REQUIRED>

<!-- Attributes for xactionmsg element -->
<!ATTLIST xactionmsg action (BEGIN | ABORT | COMMIT) #REQUIRED>

<!-- Attributes for records -->
<!ATTLIST FMLRecord encoding CDATA #IMPLIED>
<!ATTLIST FML32Record encoding CDATA #IMPLIED>
<!ATTLIST STRINGRecord encoding CDATA #IMPLIED>
<!ATTLIST CARRAYRecord encoding CDATA #IMPLIED>
<!ATTLIST JAMFLEXRecord encoding CDATA #IMPLIED>

<!-- Attributes for field element -->
<!ATTLIST field name CDATA #REQUIRED>
```

Elements in the XML Output DTD

Resultset Elements

Each such document will contain a single `resultset` element that can contain any number of `returndata` elements, `error` elements, and `xactionmsg` elements.

Returndata Elements

Each `returndata` element represents the data returned from an Oracle Tuxedo service call. A `returndata` element has a single attribute, `servicename`, that contains the name of the service that the `returndata` element represents. Each `returndata` element will contain a single element, either an `FMLRecord` element, an `FML32Record` element, a `STRINGRecord` element, or a `CARRAYRecord` element.

FMLRecord and FML32Record Elements

`FMLRecord` and `FML32Record` elements can each contain any number of `field` elements. Each `field` element has a `name` attribute, and contains the field's value as its data. If more than one of the `field` elements in a given `FMLRecord` or `FML32Record` element share the same `name` value, they represent different occurrences in a single field.

STRINGRecord and CARRAYRecord Elements

`STRINGRecord` and `CARRAYRecord` elements each contain a single `data` element, which has no attributes, and merely contains the value of the buffer as its contents.

Error Elements

Each `error` element corresponds to an Oracle Tuxedo service request that returned an error. An `error` message has a `name` attribute the value of which is the name of the Oracle Tuxedo service that returned the error. The `error` element contains the text of the error returned as its contents.

Xactionmsg Elements

Each `xactionmsg` element is empty and has a single attribute, called `action`. The `action` attribute can have the following values: `BEGIN`, `ABORT`, `COMMIT`. The `xactionmsg` elements are placed in the `returndata` elements to mark the boundaries of the transactions that had been specified in the service request.

Examples: XML Return Data

The following are examples of XML text representing the return values of service calls:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE resultset SYSTEM "tconnoutput.dtd">
<resultset>
  <returndata servicename="GetEmpData">
    <FML32Record>
      <field name=Fred>Fred occurrence one</field>
      <field name=Fred>Fred occurrence two</field>
      <field name=Fred>Fred occurrence three</field>
```

```
      <field name=Barney>This is Barney</field>
    </FML32Record>
  </returndata>
</resultset>
```

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE resultset SYSTEM "tconnoutput.dtd">
<resultset>
  <returndata servicename="TOUPPER">
    <STRINGRecord>
      <data>TO BE OR NOT TO BE</data>
    </STRINGRecord>
  </returndata>
</resultset>
```

Using an EJB to Input XML

`TConnXML`, the EJB wrapper to `XMLConnector`, supports the following interface:

```
public interface TConnXML extends javax.ejb.EJBObject
{
    String process(String xml)

    String process(String xml, String cfName)
}
```

Use the second variant of `process` to access a ConnectionFactory that has been deployed.

The first variant of `process` takes a single argument of type `String` and is like using XMLink in a non-managed environment. This is the input to the service requests.

The second variant of `process`, which is the recommended version, also requires a second `String` parameter. The value of the second parameter is used when doing a JNDI lookup to get a `ConnectionFactory` that has been deployed.

The first variant should only be used in cases where JNDI deployment of a ConnectionFactory is not desired. In this case, the ConnectionFactory is instantiated with default properties. `TUXCONFIG`, `WSNADDR` (or `WSENVFILE` must be set in the environment to specify the correct Tuxedo domain. Authentication in this case would be specified in XMLink.

The input `String` to `process` must be in XML format and must conform to the `tconn.dtd` document type definition. To see the DTD, refer to page 5-2, "Inputting Data Using XML."

APPENDIX

A Troubleshooting

Configuration Issues

Older Versions of JAR Files

If you are upgrading from an older version of WebSphere, check that the older versions of `connector.jar`, `jca.jar`, or `connector-1_0-pfd2-classes.jar` are not in the `CLASSPATH`.

Tuxedo 7.1 for Windows

For Tuxedo 7.1 for Windows, XMLink needs the Tuxedo 7.1 Rolling Patch from Oracle to work within WebSphere. The Rolling Patch must include the fix for "CR034485 stack overflow when loading a DLL on NT (S-05377)."

Specifying the Native Path and Archive Path

When configuring an XMLink resource adapter in WebSphere 6, make sure that the native path is set to the same location as the archive path. The Class Path should contain the path to `tconn.jar`. For `TConn.rar`, this would typically be `${CONNECTOR_INSTALL_ROOT}/TConn.rar/tconn.jar`.

Starting WebSphere as a Windows Service

Windows users who install WebSphere so that it is started as a Windows service (the default for WAS 6) should be aware this affects the use of operating system environment variables that can be used with the Oracle Tuxedo client within the WebSphere Application Server. If the operating system environment is changed in the control panel application, it does not effect the environment used by the process that starts Windows services. That process continues to pass the old environment to services it starts, until Windows is rebooted.

Determining Your XMLink Resource Adapter Version

As before, XMLink 3.0 provides two resource adapters `TConn.rar` and `TConn6.rar`. It is intended that `TConn.rar` should support all version of Oracle Tuxedo later than 6. However, on some platforms a version incompatibility may exist. In such cases additional resource adapter archives may be provided with XMLink for specific versions of Oracle Tuxedo. In those cases, it may be necessary to use the System property **`XMLink.tconn`**, rather than the new `ConnectionType` connection factory property setting.

XMLink Resource Adapter Files in a Managed Environment

As of WebSphere 4.0, using a resource adapter in a managed environment installs its files in a central location. For XMLink, this is:

```
$WAS_HOME\InstalledConnectors\TConn.rar\...
```

System settings for `CLASSPATH`, `PATH` or `LD_LIBRARY_PATH` (`LIBPATH`) should not reference alternate versions of these files.

A problem can occur if you previously ran XMLink in a non-managed environment which would specify these variables.

For more information on managed environments, refer to page 3-2, "Managed and Non-Managed Environments."

Tuxedo Error Messages

Note: XMLink throws a Tuxedo exception and sets the Tuxedo error code to `TPEBLOCK` when a transaction is attempted with Tuxedo 6 versions. Transaction support is only available on Tuxedo 6.x when a Tuxedo 7.1 workstation client is used in conjunction with a Tuxedo 6.4+ server. In this case, the patch for the Tuxedo 7.1 client is required.

Using the TuxedoException Class

CCI interface methods are often declared to throw a ResourceException. The Connector Architecture specification defines several extensions of ResourceException, which XMLink may use where appropriate. XMLink also defines the TuxedoException class, which extends ResourceException, for use mainly with Tuxedo generated errors.

The TuxedoException class defines five public methods that can be used to access information about the error. They are:

```
public int getError()
public int getErrorDetail()
public String getStrError()
public String getStrErrorDetail()
public String toString()
```

- `getError()` returns the Tuxedo error code associated with `tperrno`.
- `getErrorDetail()` returns the code returned by `tperrordetail()`.
- `getStrError()` returns the string returned by `tpstrerror()`.
- `getStrErrorDetail()` returns the string returned by `tpstrerrordetail()`.
- `toString()` returns a `String` containing an XMLink generated error message, plus all of the information provided by the other methods. Its format is:

```
message: tpstrerror (tpstrerrordetail) : tperrno (tperrordetail)
```

The `TuxedoException` class defines several `public static final` variables whose names correspond to the names of Tuxedo error codes associated with `tperrno` and `tperrordetail()`. These may be used to test for specific errors. For example:

```
try {
    outputRecord = interaction.execute(myTuxInteractionSpec,
inputRecord);
} catch (TuxedoException te) {
    if (te.getError() == TuxedoException.TPESVCFAIL)
    {
        // handle the service failure
    }
    // handle additional error codes that may be returned
}
```

The application may write information to the Tuxedo client log file using the static method `Tuxedo.userlog()`, which takes a single `String` parameter. XMLink provides several native methods which correspond roughly to Tuxedo ATMI methods. These methods should be used with caution. They are intended primarily for internal use by XMLink. `Tuxedo.tperrno()`, for example, may not return the same error code contained in the `TuxedoException`. That is because XMLink saves the error code right after the error occurs. This code may be changed by other ATMI calls made by XMLink before a CCI method returns to the application. Other methods of the Tuxedo class may have names which match Tuxedo ATMI methods, but parameters which do not.

In the managed environment, CCI methods may throw a `ResourceException`, which contains a chain of linked exceptions. Often the root cause is at the end of the chain, and may be a `TuxedoException`. The `getLinkedException()` method of `ResourceException` may be used in a loop until it returns null. The `ReourceException` at the end of the chain may then be tested to see if it is an instance of `TuxedoException`. If it is, it may be handled as previously described.

In addition to Tuxedo generated errors, XMLink occasionally generates a `TuxedoException` for other errors related to use of its native interface to access Tuxedo ATMI functions. In such cases, only the message portion of the exception, as returned by to`String()`, will be relevant. The `getStrError()` method of the `TuxedoException` may return the String `TPENONE`, which does not represent a real Tuxedo error code, and the `getStrErrorDetail()` method may return the String `"none"`, which does not come from Tuxedo. These strings are used when the `TuxedoException` is not caused by a Tuxedo generated error.

Using the TuxedoReturnCodeWarning Class

XMLink provides the `TuxedoReturnCodeWarning` class to hold the user return code returned by a Tuxedo service. `TuxedoReturnCodeWarning` extends the `ResourceWarning` class provided by the connector architecture, which further extends `ResourceException`. As such, several `TuxedoReturnCodeWarning` objects may be chained, just like `ResourceException` objects, but they are not thrown as exceptions. The application code uses standard CCI methods to access any `TuxedoReturnCodeWarning` objects associated with an `Interaction`.

The user return code is the second argument to `tpreturn()`. Typically, it is used to provide additional information to the Tuxedo client about a service that failed. In this case, a Tuxedo client error code of `TPESVCFAIL` is associated with a Tuxedo service

that returns `TPFAIL` as its return value. The return value is specified by the service in the first argument to `tpreturn()`. Even though this is a typical use, an application may use the user return code for some other purpose, since the client can access it even if the Tuxedo service returns `TPSUCCESS` as its return value.

Whenever a non-zero user return code is returned by a Tuxedo service, XMLink generates a `TuxedoReturnCodeWarning`, and adds it to the head of a linked list of them associated with the `Interaction`. (Note that if a user return code of zero has some special meaning for the application, application client code may call `Tuxedo.tpurcode()` directly.) The `getWarnings()` method of the `Interaction` class may be used to retrieve the `ResourceWarning` at the head of the linked list, typically a `TuxedoReturnCodeWarning`. Often there will be just one `ResourceWarning` if the `execute()` method of the Interaction was called just once, or if application code clears the list before successive calls to `execute()`. The `ResourceWarning` list can be cleared by calling the `clearWarnings()` method of the `Interaction`. The list is also cleared when the `close()` method of the Interaction is called. When the list is cleared, the `getWarning()` method returns null.

`TuxedoReturnCodeWarning` implements `getReturnCode()`, which returns a long, and `getInteractionSpec()`, which returns the `InteractionSpec` instance associated with the service call that generated the return code. `toString()` can be used to generate a `String` of the form "`<function-name> returned <code>`." Note that a `TuxedoReturnCodeWarning` will be generated for a non-zero user return code, regardless of whether `TPSUCCESS` or `TPFAIL` is returned by the service. Thus `getWarnings()` can be used directly after `execute()` is called, as well as in an exception handler block.

Using XMLink with Tuxedo 6.5

When using XMLink with Tuxedo 6.5, you need to be aware of the following issues:

- XMLink does not support transactions when used with Tuxedo 6.5. XMLink throws a `TPEBLOCK` exception if an attempt is made to begin a transaction using XMLink with Tuxedo 6.5.

Tuxedo 6.5 allows only one client/server association (connection) per client process. If one client begins a transaction on the connection, and another client makes a service call, that service call would become part of the transaction begun by the first client. Since this cannot be permitted, an exception is thrown.

- XMLink used as a Tuxedo 7.1 /WS client can be used with a Tuxedo 6.5 server. This can work around the lack of transaction support when used with Tuxedo 6.5. The Tuxedo 7.1 Rolling Patch is required for this to work.
- XMLink achieves support for its multithreaded client for Tuxedo 6.5 by serializing calls to Tuxedo API functions. This is necessary since Tuxedo 6.5 libraries are not thread-safe. `tpacall()`, rather than `tpcall()`, is used to reduce most of the overhead introduced by the serialization.
- XMLink does *not* support Tuxedo 6.5 CTS, a special "Client Threads Supplement" version of Tuxedo in which partial support for multithreaded native clients is provided.

APPENDIX

B New Features in XMLink

This appendix discusses the changes in XMLink.

XMLink 3.0

- Changes in Connection Factory properties
- Setting `XMLink.tconn`
- Improved XML support
- Support for `X_OCTET` buffers

Changes to Connection Factory Properties

J2C 1.5 supports the notion of resource adapter properties, in addition to connection factory properties. For XMLink 3.0, several of XMLink 2.60's connection factory properties (actually properties of `ManagedConnectionFactoryImpl`) were replaced with resource adapter properties. These are properties of the new `ResourceAdapterImpl` class.

For a current list of Connection Factory properties, refer to page 3-10, "Setting Properties on a Connection Factory."

No Default Setting for XMLink.tconn

The default setting for `XMLink.tconn` has been removed in XMLink 3.0, in favor of specifying the `ConnectionType` connection factory property. However, support is still available for setting this property either in the JVM or on the command line.

If `XMLink.tconn` is used, the native library is loaded when `Tuxedo.class` is loaded. Otherwise, using `ConnectionType`, the resource adapter is loaded only when it is *started* by the J2EE Application Server. For WebSphere 6, the administrative console does not let the user control starting and stopping the resource adapter, but some J2EE Application Servers support that.

Also, in the non-managed environment, `ResourceAdapter.start` is not called, so `XMLink.tconn` must be used.

Improved XML support

XMLink 3.0 contains improved XML support:

- `getXML()` method is available on all Record classes.

  This method returns a `String` containing the record contents in XML. The `String` is suitable for use with XMLink's `XMLConnector` class.

- `RecordFactoryImpl` contains a new method:

  ```
  public Record createRecordFromXML(String xml)
  ```

  The `String xml` must be in the form supported by `XMLConnector`. Any type of XMLink Record may be created this way and initialized with data.

- XMLConnector contains new methods:
  - `setOutputEncoding()`
  - `getOutputEncoding()`
  - `public Result processResult(InputStream in)`
    `throws ResourceException, SAXException, IOException`

`processResult` processes the output from `XMLConnector.process()`, in order to produce an instance of the new class, `XMLConnector.Result`.

- The new class `XMLConnector.Result` has the public properties `serviceName`, `tuxBuffer`, `error`, `userReturnCode`, `xaction` and `next`. There are also `set` and `get` methods for these properties. The `get` methods are as follows:
  - `public String getServiceName()`

    `getServiceName()` returns the value of a servicename attribute for either a returndata or error tag.
  - `public TuxBuffer getTuxBuffer()`

    `getTuxBuffer()` returns the data within an `FMLRecord`, `FML32Record`, `STRINGRecord`, `CARRAYRecord`, or `JAMFLEXRecord` tag as an instance of `TuxBuffer`.
  - `public Throwable getError()`

    `getError()` returns the contents of an error tag as a throwable. The throwable may actually be an instance of `TuxedoException`.
  - `public long getUserReturnCode()`

    `getUserReturnCode()` returns the data within a `UserReturnCode` tag as a `long`. It is assumed that a return of 0 has no particular significance, and is not indicative of the presence of a UserReturnCode tag in the input stream.
  - `public int getXaction()`

    `getXaction` returns an `int`. If this `int` is 0, no `xaction` tag was read from the input. Otherwise, it will have one of the following values:

    `XMLConnector.Result.BEGIN`

    `XMLConnector.Result.COMMIT`

    `XMLConnector.Result.ABORT`
  - `public Result getNext()`

    `getNext()` returns the next `Result` instance in a linked list. The head of the list returned by the first returndata, `xaction`, or error tag encountered. Additional such tags cause new `Result` instances to be appended to the list.

XMLink 2.6

The features for XMLink 2.6 include:

- JAMFLEX buffers
- Embedded FML
- Character encoding support

Embedded FML

Using the new `addIn` method on `FML32Record` allows you to create embedded FML for Oracle Tuxedo versions 7.1 and higher which contain support for this feature:

```
public void addIn(String name, FML32Record value)
```

Character Encoding Support

Java strings are in Unicode. By default, XMLink passes to Tuxedo only the low order byte of each Unicode character. For characters in English and other European languages, this is sufficient since only the low order byte is significant.

Since other languages may need additional support, each of XMLink's `Record` classes offers `setEncoding()` and `getEncoding()` methods. The default encoding is `ISO-8859-1`, which is a single byte per character encoding. A multibyte encoding, such as `UTF8`, can be used for foreign characters. The Tuxedo server must be able to support and decode/encode according to the specified encoding setting.

XMLink 2.1

For XMLink 2.1, a new Resource Adapter archive was added for use specifically with Tuxedo 6.5, `Tconn6.rar`.

XMLink 2.0

JNDI Lookup

For XMLink version 1.1, the documentation contained the following code to demonstrate a connection to a Java client:

```
ManagedConnectionFactoryImpl mcf =
    new ManagedConnectionFactoryImpl();

javax.resource.ConnectionFactory cxf =
  (javax.resource.ConnectionFactory)mcf.createConnectionFactory();

// get a connection
javax.resource.Connection cx = cxf.getConnection();
```

In XMLink 2.0, it is recommended to change the code as follows to incorporate the changes made for JNDI lookups:

```
// get an initial JNDI naming context
javax.naming.Context initctx = new InitialContext();

// do JNDI lookup to get connection factory
// note that lookup doesn't return a ConnectionFactory,
// so a cast is needed
javax.resource.ConnectionFactory cxf =
    (javax.resource.ConnectionFactory)initctx.lookup("TConn");
```

```
// get a connection
Connection cx = cxf.getConnection();
```

Installation Issues

XMLink's directory structure has changed in version 2.0, and a Resource Adapter Archive (`Tconn.rar`) has been added for use with WebSphere.

Also `tconnxmlout.jar` is available, a WebSphere deployable JAR file which can be imported into an Enterprise Application for XML connectivity to the resource adapter from an EJB.

Setting Properties for Connections and Connection Factories

More properties are available in XMLink 2.0 for connections and connection factories which correspond to settings in an Oracle Tuxedo configuration. If you are using WebSphere, you can set those properties in the WebSphere Administrative Console. For an explanation of the each property, refer to Oracle Tuxedo documentation.

Deploying in a Non-managed Environment

To facilitate deployment in a non-managed environment, a new class was added, `TConnTool`. The class supports a set of methods that can be used to build a customized deployment tool. It also includes a `main()` that supports a rudimentary command line interface. For more information, see page 3-4, "Deploying in a Non-managed Environment."

Index

C

CARRAYRecord
creating in Java 4-13
in XML input 5-2, 5-4
in XML output 5-5, 5-7
Character encoding 4-14
CLASSPATH
specifying 3-6
updating A-1
ConnectionFactory interface 4-3
Connections
to Tuxedo applications
specifying parameters 4-4
using Java 4-3
using XML 5-2

D

Deploying
in non-managed environment B-6
DTDs
tconn.dtd 5-2
tconnoutput.dtd 5-5

E

EJBs
calling Tuxedo services 5-8
Errors
in XML output 5-6, 5-7

F

Fields
in XML input 5-4
FML32Record
creating objects in Java 4-12
in XML input 5-2, 5-4
in XML output 5-5, 5-7
FMLRecord
creating objects in Java 4-12
in XML input 5-2, 5-4
in XML output 5-5, 5-7

I

Installing
XMLink 2-1
Interaction interface 4-7

J

J2EE Connector Architecture
specification 1-1
JAMFLEXRecord
in XML input 5-2, 5-4
in XML output 5-5

JAVA_HOME
specifying 3-6
JNDI lookup B-5

L

LD_LIBRARY_PATH
specifying 3-7

M

Managed environment 3-2

N

Non-managed environment 3-3

O

Oracle Tuxedo features
supported in XMLink 1-2

P

PATH
specifying 3-7

R

RecordFactory interface 4-10
Resource adapter 1-1
Resource adapter archives
installing 3-4

S

Service calls
in XML input 5-2, 5-4
using XML 5-1
writing in Java 4-1
Software requirements 2-1
STRINGRecord
creating in Java 4-13
in XML input 5-2, 5-4
in XML output 5-5, 5-7

T

TConnTool class 3-4, B-6
Transactions
calling methods 4-6
in Tuxedo applications
defining commit mode 4-6
in XML input 5-2, 5-3
Troubleshooting
CLASSPATH settings A-1
TUXCONFIG
specifying 3-7
TUXDIR
specifying 3-7
TuxInteractionSpec interface 4-8

U

UserReturnCode
in XML output 5-5

W

Workstation client
configuring 3-7
WSNADDR
specifying 3-7

X

XML
using to call services 5-1
XMLConnector 5-1