BEA WebLogic Server®

WebLogic Tuxedo Connector Programmer's Guide

Version 10.0
Document Revised: March 30, 2007

Contents

Introduction to WebLogic Tuxedo Connector Programming

Developing WebLogic Tuxedo Connector Client EJBs

Developing WebLogic Tuxedo Connector Service EJBs

Using WebLogic Tuxedo Connector for RMI/IIOP and CORBA Interoperability

WebLogic Tuxedo Connector JATMI Transactions

WebLogic Tuxedo Connector JATMI Conversations

Using FML with WebLogic Tuxedo Connector

WebLogic Tuxedo Connector JATMI VIEWs

How to Create a Custom AppKey Plug-in

Application Error Management

Introduction to WebLogic Tuxedo Connector Programming

Note: For information on how to develop WebLogic Server Enterprise JavaBeans (EJBs), see Programming WebLogic Enterprise JavaBeans.

The following sections provide information about the development environment you will be using to write code for applications that interoperate between WebLogic Server and Tuxedo:

- Guide to this Document
- Developing WebLogic Tuxedo Connector Applications
- WebLogic Tuxedo Connector JATMI Primitives
- WebLogic Tuxedo Connector TypedBuffers

Guide to this Document

This document introduces the BEA WebLogic Server WebLogic Tuxedo Connector application development environment. It describes how to develop EJBs that allow WebLogic Server to interoperate with Tuxedo objects.

The document is organized as follows:

- Chapter 1, "Introduction to WebLogic Tuxedo Connector Programming," provides information about the development environment you will be using to write code for applications that interoperate between WebLogic Server and Tuxedo.
- Chapter 2, "Developing WebLogic Tuxedo Connector Client EJBs," provides information on how to create client EJBs.

- Chapter 3, "Developing WebLogic Tuxedo Connector Service EJBs," provides information on how to create service EJBs.
- Chapter 4, "Using WebLogic Tuxedo Connector for RMI/IIOP and CORBA Interoperability," provides information on how to develop CORBA applications for the WebLogic Tuxedo Connector.
- Chapter 5, "WebLogic Tuxedo Connector JATMI Transactions," provides information on global transactions and how to define and manage them in your applications.
- Chapter 6, "WebLogic Tuxedo Connector JATMI Conversations," provides information on conversations and how to define and manage them in your applications.
- Chapter 7, "Using FML with WebLogic Tuxedo Connector," discusses the Field Manipulation Language (FML) and describes how the WebLogic Tuxedo Connector uses FML.
- Chapter 8, "WebLogic Tuxedo Connector JATMI VIEWs," provides information on View buffers and how to define and manage them in your applications.
- Chapter 9, "How to Create a Custom AppKey Plug-in," provides information on how to develop a Custom AppKey Plug-in.
- Chapter 10, "Application Error Management," provide mechanisms to manage and interpret error conditions.

Developing WebLogic Tuxedo Connector Applications

Note: For more information on the WebLogic Tuxedo Connector JATMI, view the Javadocs for WebLogic Classes. The WebLogic Tuxedo Connector classes are located in the `weblogic.wtc.jatmi` and `weblogic.wtc.gwt` packages.

In addition to the Java code that expresses the logic of your application, you will be using the Java Application -to-Transaction Monitor Interface (JATMI) to provide the interface between WebLogic Server and Tuxedo.

Developing WebLogic Tuxedo Connector Clients

Note: For more information, see "Developing WebLogic Tuxedo Connector Client EJBs" on page 2-1.

A client process takes user input and sends a service request to a server process that offers the requested service. WebLogic Tuxedo Connector JATMI client classes are used to create clients that access services found in Tuxedo. These client classes are available to any service that is made available through a the WebLogic Tuxedo Connector WTCServer MBean.

Developing WebLogic Tuxedo Connector Servers

Note: For more information, see "Developing WebLogic Tuxedo Connector Service EJBs" on page 3-1.

Servers are processes that provide one or more services. They continually check their message queue for service requests and dispatch them to the appropriate service subroutines. WebLogic Tuxedo Connector uses EJBs to implement services which Tuxedo clients invoke.

WebLogic Tuxedo Connector Interoperability with Tuxedo CORBA objects

Note: For more information, see "Using WebLogic Tuxedo Connector for RMI/IIOP and CORBA Interoperability" on page 4-1.

The WebLogic Tuxedo Connector provides bi-directional interoperability between WebLogic Server and Tuxedo CORBA objects. The WebLogic Tuxedo Connector:

- Enables Tuxedo CORBA objects to invoke upon EJBs deployed in WebLogic Server using the RMI/IIOP API (Inbound).
- Enables objects (such as EJBs or RMI objects) to invoke upon CORBA objects deployed in Tuxedo using the RMI/IIOP API (Outbound).
- Enables objects (such as EJBs or RMI objects) to invoke upon CORBA objects deployed in Tuxedo using a CORBA Java API (Outbound).

WebLogic Tuxedo Connector JATMI Primitives

The JATMI is a set of primitives used to begin and end transactions, allocate and free buffers, and provide the communication between clients and servers.

Table 1-1 JATMI Primitives

Name	Operation
`tpacall`	Use for asynchronous invocations of a Tuxedo service during request/response communication.`tpacall` has two forms: • deferred synchronous • asynchronous
`tpcall`	Use for synchronous invocation of a Tuxedo service during request/response communication.
`tpconnect`	Use to establish a connection to a Tuxedo conversational service.
`tpdiscon`	Use to abort a conversational connection and generate a TPEV_DISCONIMM event when executed by the process controlling the conversation.
`tpdequeue`	Use for receiving messages from a Tuxedo /Q during request/response communication.
`tpenqueue`	Use for placing a message on a Tuxedo /Q during request/response communication.
`tpgetrply`	Use for retrieving replies from a Tuxedo service during request/response communication.
`tprecv`	Use to receive data across an open connection from a Tuxedo application during conversational communication.
`tpsend`	Use to send data across a open connection to a Tuxedo application during conversational communication.
`tpterm`	Use to close a connection to a Tuxedo object.

WebLogic Tuxedo Connector TypedBuffers

WebLogic Tuxedo Connector provides an interface called `TypedBuffers` that corresponds to Tuxedo typed buffers. Messages are passed to servers in typed buffers. The WebLogic Tuxedo Connector provides the following buffer types:.

Table 1-2 TypedBuffers

Buffer Type	Description
TypedString	Buffer type used when the data is an array of characters that terminates with the null character. Tuxedo equivalent: STRING.
TypedCArray	Buffer type used when the data is an undefined array of characters (byte array), any of which can be null. Tuxedo equivalent: CARRAY.
TypedFML	Buffer type used when the data is self-defined. Each data field carries its own identifier, an occurrence number, and possibly a length indicator. Tuxedo equivalent: FML.
TypedFML32	Buffer type similar to TypeFML but allows for larger character fields, more fields, and larger overall buffers. Tuxedo equivalent: FML32.
TypedXML	Buffer type used when data is an XML based message. Tuxedo equivalent: XML for Tuxedo Release 7.1 and higher.
TypedView	Buffer type used when the application uses a Java structure to define the buffer structure using a view description file. Tuxedo equivalent: VIEW
TypedView32	Buffer type similar to View but allows for larger character fields, more fields, and larger overall buffers. Tuxedo equivalent: VIEW32.
TypedMBString	Buffer type used when the data is a wide array of characters to support multibyte characters. Tuxedo equivalent: MBSTRING.

Developing WebLogic Tuxedo Connector Client EJBs

Note: For more information on the WebLogic Tuxedo Connector JATMI, view the Javadocs for WebLogic Classes. The WebLogic Tuxedo Connector classes are located in the `weblogic.wtc.jatmi` and `weblogic.wtc.gwt` packages.

The following sections describe how to create client EJBs that take user input and send service requests to a server process or outbound object that offers a requested service.

- Joining and Leaving Applications
- Basic Client Operation
- Example Client EJB

WebLogic Tuxedo Connector JATMI client classes are used to create clients that access services found in Tuxedo.

Joining and Leaving Applications

Tuxedo and WebLogic Tuxedo Connector have different approaches to connect to services.

Joining an Application

The following section compares how Tuxedo and WebLogic Tuxedo Connector join an application:

- Tuxedo uses `tpinit()` to join an application.

- WebLogic Tuxedo Connector uses a WTCServer MBean to provide information required to create a path to the Tuxedo service. Security and client authentication is provided by configuring the Remote TDM and Imported Services MBean components of a WTCServer MBean. This pathway is created when the WebLogic Server is started and a WTCServer MBean is present in the `config.xml` file and assigned (targeted) to a server.
- WebLogic Tuxedo Connector uses `TuxedoConnectionFactory` to get a `TuxedoConnection` object and then uses `getTuxedoConnection()` to make a connection to the Tuxedo object. The following example shows how a WebLogic Server application joins a Tuxedo application using WebLogic Tuxedo Connector.

Listing 2-1 Example Client Code to Join a Tuxedo Application

```
.
.
.
try {
     ctx = new InitialContext();
     tcf =
          (TuxedoConnectionFactory)
          ctx.lookup("tuxedo.services.TuxedoConnection");
     } catch (NamingException ne) {

// Could not get the tuxedo object, throw TPENOENT
throw new TPException(TPException.TPENOENT,
     "Could not get TuxedoConnectionFactory : " + ne);
     }

myTux = tcf.getTuxedoConnection();
.
.
.
```

Leaving an Application

The following section compares how Tuxedo and WebLogic Tuxedo Connector leave an application:

- Tuxedo uses `tpterm()` to leave an application.
- WebLogic Tuxedo Connector uses the JATMI primitive `tpterm()` to close a connection to a Tuxedo object.
- WebLogic Tuxedo Connector closes the pathway to a Tuxedo service when a WTCServer MBean is assigned a new target server or the server is shutdown.

Basic Client Operation

A client process uses Java and JATMI primitives to provide the following basic application tasks:

- Get a Tuxedo Object
- Perform Message Buffering
- Send and Receive Messages
- Close a Connection to a Tuxedo Object

A client may send and receive any number of service requests before leaving the application.

Get a Tuxedo Object

Establish a connection to a remote domain by looking up "`tuxedo.services.TuxedoConnection`" in the JNDI tree to get `TuxedoConnectionFactory`, and use it to get a `TuxedoConnection` object.

Perform Message Buffering

Use the following `TypedBuffers` when sending and receiving messages between your application and Tuxedo:

Table 2-1 TypedBuffers

Buffer Type	Description
TypedString	Buffer type used when the data is an array of characters that terminates with the null character. Tuxedo equivalent: STRING.
TypedCArray	Buffer type used when the data is an undefined array of characters (byte array), any of which can be null. Tuxedo equivalent: CARRAY.

Table 2-1 TypedBuffers

Buffer Type	Description
TypedFML	Buffer type used when the data is self-defined. Each data field carries its own identifier, an occurrence number, and possibly a length indicator. Tuxedo equivalent: FML.
TypedFML32	Buffer type similar to TypeFML but allows for larger character fields, more fields, and larger overall buffers. Tuxedo equivalent: FML32.
TypedXML	Buffer type used when data is an XML based message. Tuxedo equivalent: XML for Tuxedo Release 7.1 and higher.
TypedView	Buffer type used when the application uses a Java structure to define the buffer structure using a view description file. Tuxedo equivalent: View **Note:** This buffer type is missing from the WLS Javadocs, see `TypedView` instead.
TypedView32	Buffer type similar to View but allows for larger character fields, more fields, and larger overall buffers. Tuxedo equivalent: View32. **Note:** This buffer type is missing from the WLS Javadocs, see `TypedView32` instead.
TypedMBString	Buffer type used when the data is a wide array of characters to support multibyte characters. Tuxedo equivalent: MBSTRING.

Send and Receive Messages

WebLogic Tuxedo Connector clients support three types of communications with Tuxedo service applications:

- Request/Response Communication
- Conversational Communication
- Enqueuing and Dequeuing Messages

Request/Response Communication

Note: WebLogic Tuxedo Connector does not provide a JATMI primitive to support setting the priority of a message request. All messages originating from a WebLogic Tuxedo Connector client have a message priority of 50.

Use the following JATMI primitives to request and receive response messages between your WebLogic Tuxedo Connector client application and Tuxedo:

Table 2-2 JATMI Primitives

Name	Operation
`tpacall`	Use for asynchronous invocations of a Tuxedo service.This JATMI primitive has two forms: • deferred synchronous • asynchronous
`tpcall`	Use for synchronous invocation of a Tuxedo service.
`tpgetrply`	Use for retrieving replies from deferred synchronous calls to a Tuxedo service.
`tpcancel`	Use to cancel an outstanding message reply for a call descriptor returned by `tpacall`. **Note:** You can not use `tpcancel` to cancel a call descriptor associated with a transaction.

Using Synchronous Service Calls

Use `tpcall` to send a request to a service and synchronously await for the reply. The service specified must be advertised by your Tuxedo application. Logically, `tpcall()` has the same functionality as calling `tpacall()` and immediately calling `tpgetreply()`.

Using Deferred Synchronous Service Calls

A deferred synchronous `tpacall` allows you to send a request to a Tuxedo service and not immediately wait for the reply. This allows you to send a request, perform other work, and then retrieve the reply.

A deferred `tpacall()` service call sends a request to a Tuxedo service and immediately returns from the call. The service specified must be advertised by your Tuxedo application. Upon successful completion of the call, `tpacall()` returns an object that serves as a descriptor. The calling thread is now available to perform other tasks. You can use the call descriptor to:

- Get the correct reply for the sent request using `tpgetreply()`
- Cancel an outstanding message reply using `tpcancel()`.

When you are ready to retrieve the reply, use `tpgetreply()` to dequeue the reply using the call descriptor returned by `tpacall()`. If the reply is not immediately available, the calling thread polls for the reply.

If `tpacall()` is in a transaction, you must receive the reply using `tpgetreply()` before the transaction can commit. You can not use `tpcancel` to cancel a call descriptor associated with a transaction. For example: If you make three `tpacall()` requests in a transaction, you must make three `tpgetreply()` calls and successfully dequeue a reply for each of the three requests for the transaction to commit.

Using Asynchronous Calls

The asynchronous `tpacall` allows you to send a request to a Tuxedo service and release the thread resource that performed the call to the thread pool. This allows a very large number of outstanding requests to be serviced with a much smaller number of threads.

An asynchronous `tpacall()` service call sends a request to a Tuxedo service. The service specified must be advertised by your Tuxedo application. Upon successful completion of the call, asynchronous `tpacall()` returns an object that serves as a descriptor. The calling thread is now available to perform other tasks. You can use the call descriptor to identify the correct message reply from `TpacallAsynchReply` for a sent message request or cancel an outstanding message reply using `tpcancel()`.

Note: You can not use the call descriptor to invoke `tpgetreply()`.

When the service reply is ready, the `callback` object is invoked on a different thread. If the original request succeeded, the `TpacallAsynchReply.sucess` method returns the reply from the service. If the original request failed, the `TpacallAsynchReply.failure` method returns a failure code.

You should implement the `callback` object using the following guidelines:

- The reply thread is obtained from the threadpool. The thread making the asynchronous `tpacall()` does not wait for the reply message.
- The user context of the reply thread will be restored to that of the original caller of asynchronous `tpacall()`.

- It is up to the callback object to restore any additional context and resume whatever processing was interrupted when the original asynchronous `tpacall()` was made.
- It is up to you to synchronize work within the multi threaded environment. For example: If an asynchronous `tpacall()` request is made and the reply is returned immediately, it is possible for the call back object to be modified by the reply thread before the calling thread has finished.
- The reply thread will not retain the transaction context of the calling thread.
- If asynchronous `tpacall()` is in a transaction, you must receive the reply using `TpacallAsynchReply` before the transaction can commit. You can not use `tpcancel` to cancel a call descriptor associated with a transaction.

Conversational Communication

Note: For more information on Conversational Communication, see "WebLogic Tuxedo Connector JATMI Conversations" on page 6-1.

Use the following conversational primitives when creating conversational clients that communicate with Tuxedo services:

Table 2-3 WebLogic Tuxedo Connector Conversational Client Primitives

Name	Operation
`tpconnect`	Use to establish a connection to a Tuxedo conversational service.
`tpdiscon`	Use to abort a connection and generate a TPEV_DISCONIMM event when executed by the process controlling the conversation.
`tprecv`	Use to receive data across an open connection from a Tuxedo application.
`tpsend`	Use to send data across an open connection to a Tuxedo application.

Enqueuing and Dequeuing Messages

Use the following JATMI primitives to enqueue and dequeue messages between your WebLogic Tuxedo Connector client application and Tuxedo /Q:

Table 2-4 JATMI Primitives

Name	Operation
`tpdequeue`	Use for receiving messages from a Tuxedo /Q.
`tpenqueue`	Use for placing a message on a Tuxedo /Q.

Close a Connection to a Tuxedo Object

Use `tpterm()` to close a connection to an object and prevent future operations on this object.

Example Client EJB

The following Java code provides an example of the `ToupperBean.java` client EJB which sends a string argument to a server and receives a reply string from the server.

Listing 2-2 Example Client Application

```
.
.
.
public String Toupper(String toConvert)
   throws TPException, TPReplyException
{
            Context ctx;
            TuxedoConnectionFactory tcf;
            TuxedoConnection myTux;
            TypedString myData;
       Reply myRtn;
            int status;

            log("toupper called, converting " + toConvert);

            try {
              ctx = new InitialContext();
                        tcf = (TuxedoConnectionFactory) ctx.lookup(
                     "tuxedo.services.TuxedoConnection");
```

```
            }
            catch (NamingException ne) {
                        // Could not get the tuxedo object, throw TPENOENT
              throw new TPException(TPException.TPENOENT, "Could not get
TuxedoConnectionFactory : " + ne);
       }

            myTux = tcf.getTuxedoConnection();

            myData = new TypedString(toConvert);

            log("About to call tpcall");
       try {
                        myRtn = myTux.tpcall("TOUPPER", myData, 0);
       }
            catch (TPReplyException tre) {
                        log("tpcall threw TPReplyExcption " + tre);
              throw tre;
            }
       catch (TPException te) {
              log("tpcall threw TPException " + te);
              throw te;
       }
       catch (Exception ee) {
              log("tpcall threw exception: " + ee);
              throw new TPException(TPException.TPESYSTEM, "Exception: " + ee);
       }
     log("tpcall successfull!");

       myData = (TypedString) myRtn.getReplyBuffer();

       myTux.tpterm();// Closing the association with Tuxedo

       return (myData.toString());
}
.
.
.
```

Developing WebLogic Tuxedo Connector Service EJBs

The following sections provide information on how to create WebLogic Tuxedo Connector service EJBs:

- Basic Service EJB Operation
- Example Service EJB

Basic Service EJB Operation

A service application uses Java and JATMI primitives to provide the following tasks:

- Access Service Information
- Buffer Messages
- Perform the Requested Service

Access Service Information

Use the `TPServiceInformation` class to access service information sent by the Tuxedo client to run the service.

Table 3-1 JATMI TPServiceInformation Primitives

Buffer Type	Description
`getServiceData()`	Use to return the service data sent from the Tuxedo Client.
`getServiceFlags()`	Use to return the service flags sent from the Tuxedo Client.
`getServiceName()`	Use to return the service name that was called.

Buffer Messages

Use the following `TypedBuffers` when sending and receiving messages between your application and Tuxedo:

Table 3-2 TypedBuffers

Buffer Type	Description
TypedString	Buffer type used when the data is an array of characters that terminates with the null character. Tuxedo equivalent: STRING.
TypedCArray	Buffer type used when the data is an undefined array of characters (byte array), any of which can be null. Tuxedo equivalent: CARRAY.
TypedFML	Buffer type used when the data is self-defined. Each data field carries its own identifier, an occurrence number, and possibly a length indicator. Tuxedo equivalent: FML.
TypedFML32	Buffer type similar to TypeFML but allows for larger character fields, more fields, and larger overall buffers. Tuxedo equivalent: FML32.
TypedXML	Buffer type used when data is an XML based message. Tuxedo equivalent: XML for Tuxedo Release 7.1 and higher.
TypedView	Buffer type used when the application uses a Java structure to define the buffer structure using a view description file. Tuxedo equivalent: VIEW

Table 3-2 TypedBuffers

Buffer Type	Description
TypedView32	Buffer type similar to View but allows for larger character fields, more fields, and larger overall buffers. Tuxedo equivalent: VIEW32.
TypedXOctet	Buffer type used when the data is an undefined array of characters (byte array) any of which can be null. X_OCTET is identical in semantics to CARRAY. Tuxedo equivalent: X_OCTET.
TypedXCommon	Buffer type identical in semantics to View. Tuxedo equivalent: VIEW.
TypedXCType	Buffer type identical in semantics to View. Tuxedo equivalent: VIEW.
TypedMBString	Buffer type used when the data is a wide array of characters to support multibyte characters. Tuxedo equivalent: MBSTRING.

Perform the Requested Service

Use Java code to express the logic required to provide your service.

Return Client Messages for Request/Response Communication

Use the `TuxedoReply` class `setReplyBuffer()` method to respond to client requests.

Use tpsend and tprecv for Conversational Communication

Note: For more information on Conversational Communication, see “WebLogic Tuxedo Connector JATMI Conversations” on page 6-1.

Use the following JATMI primitives when creating conversational servers that communicate with Tuxedo clients:

Table 3-3 WebLogic Tuxedo Connector Conversational Client Primitives

Name	Operation
tpconnect	Use to establish a connection to a Tuxedo conversational service.
tpdiscon	Use to abort a connection and generate a TPEV_DISCONIMM event when executed by the process controlling the conversation.
tprecv	Use to receive data across an open connection from a Tuxedo application.
tpsend	Use to send data across a open connection to a Tuxedo application.

Example Service EJB

The following provides an example of the `TolowerBean.java` service EJB which receives a string argument, converts the string to all lower case, and returns the converted string to the client.

Listing 3-1 Example Service EJB

```
.
.
.

public Reply service(TPServiceInformation mydata) throws TPException {
     TypedString data;
     String lowered;
     TypedString return_data;

     log("service tolower called");

     data = (TypedString) mydata.getServiceData();
     lowered = data.toString().toLowerCase();
     return_data = new TypedString(lowered);

     mydata.setReplyBuffer(return_data);
     return (mydata);
```

```
}
.
.
.
```

Using WebLogic Tuxedo Connector for RMI/IIOP and CORBA Interoperability

Note: You will need to perform some administration tasks to configure the WebLogic Tuxedo Connector for CORBA interoperability. For information on how to administer the WebLogic Tuxedo Connector for CORBA interoperability, see Administration of CORBA Applications.

For information on how to develop Tuxedo CORBA applications, see CORBA Programming at http://e-docs.bea.com/tuxedo/tux90/interm/corbaprog.htm.

The following sections provide information on how to modify your applications to use WebLogic Tuxedo Connector to support interoperability between WebLogic Server and Tuxedo CORBA objects:

- How to Develop WebLogic Tuxedo Connector Client Beans using the CORBA Java API
- How to Develop RMI/IIOP Applications for the WebLogic Tuxedo Connector
- How to Use FederationURL Formats
- How to Manage Transactions for Tuxedo CORBA Applications

How to Develop WebLogic Tuxedo Connector Client Beans using the CORBA Java API

The WebLogic Tuxedo Connector enables objects (such as EJBs or RMI objects) to invoke upon CORBA objects deployed in Tuxedo using the CORBA Java API (Outbound). WebLogic Tuxedo Connector implements a WTC ORB which uses WebLogic Server RMI-IIOP runtime and CORBA support. This enhancement provides the following features:

- Support of out and inout parameters
- Support for a call a CORBA service from WebLogic Server using transactions and security.
- Support for an ORB hosted in JNDI rather than an instance of the JDK ORB used in previous releases.
- A wrapper is provided to allow users with legacy applications to use the new ORB without modifying their existing applications. BEA recommends that users migrate to the new method of looking up the ORB in JNDI instead of doing:

```
ORB orb = ORB.init(args, Prop);
```

To use CORBA Java API, you must use the WTC ORB. Use one of the following methods to obtain an ORB in your Bean:

```
Properties Prop;
Prop = new Properties();
Prop.put("org.omg.CORBA.ORBClass","weblogic.wtc.corba.ORB");
ORB orb = ORB.init(new String[0], Prop);
```

or

```
ORB orb = (ORB)(new InitialContext().lookup("java:comp/ORB"));
```

or

```
ORB orb = ORB.init();
```

You can use either of the following methods to reference objects deployed in Tuxedo:

- Using CosNaming Service
- Using FactoryFinder

Using CosNaming Service

Note: For more information on object references, see “How to Use FederationURL Formats” on page 4-12.

1. The WebLogic Tuxedo Connector uses the CosNaming service to get a reference to an object in the remote Tuxedo CORBA domain. This is accomplished by using a `corbaloc:tgiop` or `corbaname:tgiop` object reference. The following statements use the CosNaming service to get a reference to a Tuxedo CORBA Object:

```
// Get the simple factory.
org.omg.CORBA.Object simple_fact_oref =
     orb.string_to_object("corbaname:tgiop:simpapp#simple_factory");
```

Where:

- simpapp is the domain id of the Tuxedo domain specified in the Tuxedo UBB.
- simple_factory is the name that the object reference was bound to in the Tuxedo CORBA CosNaming server.

Example TouupperCorbaBean.java Code

Note: For an example on how to develop client beans for outbound Tuxedo CORBA objects, see the SAMPLES_HOME\server\examples\src\examples\wtc\corba\simpappcns package in your WebLogic Server examples distribution.

The following ToupperCorbaBean.java code provides an example of how to call the WTC ORB and get an object reference using the COSNaming Service.

Listing 4-1 Example Service Application

```
.
.
.
public String Toupper(String toConvert)
throws RemoteException
{
       log("toupper called, converting " + toConvert);

     try {
        // Initialize the ORB.
        String args[] = null;
         Properties Prop;

        Prop = new Properties();
        Prop.put("org.omg.CORBA.ORBClass",
                "weblogic.wtc.corba.ORB");

        ORB orb = (ORB) new InitialContext().lookup("java:comp/ORB");

        // Get the simple factory.
        org.omg.CORBA.Object simple_fact_oref =
        orb.string_to_object("corbaname:tgiop:simpapp#simple_factory");

        //Narrow the simple factory.
        SimpleFactory simple_factory_ref =
        SimpleFactoryHelper.narrow(simple_fact_oref);

        // Find the simple object.
```

```
        Simple simple = simple_factory_ref.find_simple();

        // Convert the string to upper case.
        org.omg.CORBA.StringHolder buf =
          new org.omg.CORBA.StringHolder(toConvert);
        simple.to_upper(buf);
        return buf.value;
     }
     catch (Exception e) {
        throw new RemoteException("Can't call TUXEDO CORBA server: " +e);
     }
}
.
.
.
```

Using FactoryFinder

Note: For more information on object references, see "How to Use FederationURL Formats" on page 4-12.

WebLogic Tuxedo Connector provides support for FactoryFinder objects using the find_one_factory_by_id method. This is accomplished by using a `corbaloc:tgiop` or `corbaname:tgiop` object reference. Use the following method to obtain the FactoryFinder object using the ORB:

```
// String to Object.
org.omg.CORBA.Object fact_finder_oref =
     orb.string_to_object("corbaloc:tgiop:simpapp/FactoryFinder");

// Narrow the factory finder.
FactoryFinder fact_finder_ref =
     FactoryFinderHelper.narrow(fact_finder_oref);

// Use the factory finder to find the simple factory.
org.omg.CORBA.Object simple_fact_oref =
     fact_finder_ref.find_one_factory_by_id(SimpleFactoryHelper.id());
```

Where:

- `simpapp` is the domain id of the Tuxedo domain specified in the Tuxedo UBB.
- `FactoryFinder` is the name that the object reference was bound to in the Tuxedo CORBA server.

WLEC to WebLogic Tuxedo Connector Migration

WLEC is no longer available or supported in WebLogic Server. WLEC users should migrate their applications to WebLogic Tuxedo Connector. For more information, see WLEC to WebLogic Tuxedo Connector Migration Guide.

Example Code

The following code provides an example of how to call the WTC ORB and get an object reference using FactoryFinder.

Listing 4-2 Example FactoryFinder Code

```
.
.
.
public ConverterResult convert (String changeCase, String mixed)
throws ProcessingErrorException
{
     String result;
     try {
     // Initialize the ORB.
     String args[] = null;
     Properties Prop;
     Prop = new Properties();
     Prop.put("org.omg.CORBA.ORBClass","weblogic.wtc.corba.ORB");
     ORB orb = (ORB)new InitialContext().lookup("java:comp/ORB");

     org.omg.CORBA.Object fact_finder_oref =
         orb.string_to_object("corbaloc:tgiop:simpapp/FactoryFinder");

     // Narrow the factory finder.
     FactoryFinder fact_finder_ref =
        FactoryFinderHelper.narrow(fact_finder_oref);

     // find_one_factory_by_id
     org.omg.CORBA.Object simple_fact_oref =
        fact_finder_ref.find_one_factory_by_id(FactoryFinderHelper.id());
```

```
      // Narrow the simple factory.
      SimpleFactory simple_factory_ref =
         SimpleFactoryHelper.narrow(simple_fact_oref);

      // Find the simple object.
      Simple simple = simple_factory_ref.find_simple();

      if (changeCase.equals("UPPER")) {
      // Invoke the to_upper opeation on M3 Simple object
      org.omg.CORBA.StringHolder buf =
         new org.omg.CORBA.StringHolder(mixed);
      simple.to_upper(buf);
      result = buf.value;
      }
      else
      {
      result = simple.to_lower(mixed);
      }

      }
      catch (org.omg.CORBA.SystemException e) {e.printStackTrace();

      throw new ProcessingErrorException("Converter error: Corba system exc
eption: " + e);
      }
      catch (Exception e) {
      e.printStackTrace();
      throw new ProcessingErrorException("Converter error: " + e);
      }
return new ConverterResult(result);
}
.
.
.
```

How to Develop RMI/IIOP Applications for the WebLogic Tuxedo Connector

Note: For more information on how to develop RMI/IIOP applications, see Programming WebLogic RMI.

RMI over IIOP (Internet Inter-ORB Protocol) extends RMI so that Java programs can interact with Common Object Request Broker Architecture (CORBA) clients and execute CORBA objects. The WebLogic Tuxedo Connector:

- Enables Tuxedo CORBA objects to invoke upon EJBs deployed in WebLogic Server (Inbound).
- Enables objects (such as EJBs or RMI objects) to invoke upon CORBA objects deployed in Tuxedo (Outbound).

The following sections provide information on how to modify RMI/IIOP applications to use the WebLogic Tuxedo Connector to interoperate with Tuxedo CORBA applications:

- How to Modify Inbound RMI/IIOP Applications to use the WebLogic Tuxedo Connector
- How to Develop Outbound RMI/IIOP Applications to use the WebLogic Tuxedo Connector

How to Modify Inbound RMI/IIOP Applications to use the WebLogic Tuxedo Connector

A client must pass the correct name to which the WebLogic Server's name service has been bound to the COSNaming Service.

The following code provides an example for obtaining a naming context. "WLS" is the bind name specified in the `cnsbind` command detailed in the Administration of CORBA Applications.

Listing 4-3 Example Code to Obtain a Naming Context

```
.
.
.
// obtain a naming context
    TP::userlog("Narrowing to a naming context");
    CosNaming::NamingContext_var context =
```

```
        CosNaming::NamingContext::_narrow(o);
    CosNaming::Name name;
    name.length(1);
    name[0].id = CORBA::string_dup("WLS");
    name[0].kind = CORBA::string_dup("");
.
.
.
```

How to Develop Outbound RMI/IIOP Applications to use the WebLogic Tuxedo Connector

An EJB must use a FederationURL to obtain the initial context used to access a remote Tuxedo CORBA object. Use the following sections to modify outbound RMI/IIOP applications to use the WebLogic Tuxedo Connector:

- How to Modify the ejb-jar.xml File to Pass a FederationURL to EJBs
- How to Modify EJBs to Use FederationURL to Access an Object

How to Modify the ejb-jar.xml File to Pass a FederationURL to EJBs

The following code provides an example of how to configure an `ejb-jar.xml` file to pass a FederationURL format to the EJB at run-time.

Listing 4-4 Example ejb-jar.xml File Passing a FederationURL to an EJB

```
<?xml version="1.0"?>

<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
1.1//EN' 'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>

<ejb-jar>
    <small-icon>images/green-cube.gif</small-icon>
    <enterprise-beans>
    <session>
        <small-icon>images/orange-cube.gif</small-icon>
              <ejb-name>IIOPStatelessSession</ejb-name>
```

```
                <home>examples.iiop.ejb.stateless.TraderHome</home>
                <remote>examples.iiop.ejb.stateless.Trader</remote>
                <ejb-class>examples.iiop.ejb.stateless.TraderBean</ejb-class>
                <session-type>Stateless</session-type>
                <transaction-type>Container</transaction-type>
                       <env-entry>
                     <env-entry-name>foreignOrb</env-entry-name>
                     <env-entry-type>java.lang.String </env-entry-type>

              <env-entry-value>corbaloc:tgiop:simpapp</env-entry-value>
                </env-entry>
         <env-entry>
                     <env-entry-name>WEBL</env-entry-name>
                     <env-entry-type>java.lang.Double </env-entry-type>
                     <env-entry-value>10.0</env-entry-value>
                </env-entry>
                <env-entry>
                     <env-entry-name>INTL</env-entry-name>
                     <env-entry-type>java.lang.Double </env-entry-type>
                     <env-entry-value>15.0</env-entry-value>
                </env-entry>
                <env-entry>
                     <env-entry-name>tradeLimit</env-entry-name>
                     <env-entry-type>java.lang.Integer </env-entry-type>
                     <env-entry-value>500</env-entry-value>
                </env-entry>
    </session>
    </enterprise-beans>
    <assembly-descriptor>
         <container-transaction>
                     <method>
                          <ejb-name>IIOPStatelessSession</ejb-name>
                          <method-intf>Remote</method-intf>
                          <method-name>*</method-name>
                     </method>
                <trans-attribute>NotSupported</trans-attribute>
         </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

To pass the FederationURL to the EJB at run-time, add an `env-entry` for the EJB in the `ejb-jar.xml` file for your application. You must assign the following `env-entry` sub-elements:

- Assign env-entry-name

- Assign env-entry-type
- Assign env-entry-value

Assign env-entry-name

The `env-entry-name` element is used to specify the name of the variable used to pass the value in the `env-entry-value` element to the EJB. The example code shown in Listing 4-4 specifies the `env-entry-name` as `foreignOrb`.

Assign env-entry-type

The `env-entry-type` element is used to specify the data type (example String, Integer, Double) of the `env-entry-value` element that is passed to the EJB. The example code shown in Listing 4-4 specifies that the `foreignOrb` variable passes `String` data to the EJB.

Assign env-entry-value

The `env-entry-value` element is used to specify the data that is passed to the EJB. The example code shown in Listing 4-4 specifies that the `foreignOrb` variable passes the following FederationURL format to the EJB:

```
corbaloc:tgiop:simpapp
```

Where `simpapp` is the `DOMAINID` of the Tuxedo remote service specified in the Tuxedo UBB.

How to Modify EJBs to Use FederationURL to Access an Object

This section provides information on how to use the FederationURL to obtain the InitialContext used to access a remote Tuxedo CORBA object.

The following code provides an example of how to use FederationURL to get an InitialContext.

Listing 4-5 Example TraderBean.java Code to get InitialContext

```
.
.
.
public void createRemote() throws CreateException {
     log("createRemote() called");

     try {
          InitialContext ic = new InitialContext();
```

```
     // Lookup a EJB-like CORBA server in a remote CORBA domain
          Hashtable env = new Hashtable();
          env.put(Context.PROVIDER_URL, (String)
             ic.lookup("java:/comp/env/foreignOrb")
             + "/NameService");

          InitialContext cos = new InitialContext(env);
          TraderHome thome =
             (TraderHome)PortableRemoteObject.narrow(
             cos.lookup("TraderHome_iiop"),TraderHome.class);
             remoteTrader = thome.create();
}
     catch (NamingException ne) {
     throw new CreateException("Failed to find value "+ne);
}
     catch (RemoteException re) {
     throw new CreateException("Error creating remote ejb "+re);
}
}
.
.
.
```

Use the following steps to use FederationURL to obtain an InitialContext for a remote Tuxedo CORBA object:

1. Retrieve the FederationURL format defined in the `ejb-jar.xml` file.

   Example:

   `"ic.lookup("java:/comp/env/foreignOrb")`

   The example code shown in Listing 4-4 specifies that the `foreignOrb` variable passes the following FederationURL format to the EJB:

   `corbaloc:tgiop:simpapp`

2. Concatenate the FederationURL format with "`/NameService`" to form the FederationURL.

Example:

```
"ic.lookup("java:/comp/env/foreignOrb") + "/NameService"
```

The resulting FederationURL is:

```
corbaloc:tgiop:simpapp/NameService
```

3. Get the InitialContext.

   Example:

```
env.put(Context.PROVIDER_URL, (String)
     ic.lookup("java:/comp/env/foreignOrb") + "/NameService");
InitialContext cos = new InitialContext(env);
```

   The result is the InitialContext of the Tuxedo CORBA object.

How to Use FederationURL Formats

This section provides information on the syntax for the following FederationURL formats:

- The `CORBA URL` syntax is described in the CORBA specification. For more information, see the OMG Web Site at http://www.omg.org/.
- The `corbaloc:tgiop` form is specific to the BEA tgiop protocol.

Using corbaloc URL Format

This section provides the syntax for corbaloc URL format:

```
<corbaloc> = "corbaloc:tgiop":[<version>] <domain>["/"<key_string>]

<version> = <major> "." <minor> "@" | empty_string

<domain> = TUXEDO CORBA domain name

<major> = number

<minor> = number

<key_string> = <string> | empty_string
```

Examples of corbaloc:tgiop

This section provides examples on how to use `corbaloc:tgiop`

```
orb.string_to_object("corbaloc:tgiop:simpapp/NameService");
```

```
orb.string_to_object("corbaloc:tgiop:simpapp/FactoryFinder");
orb.string_to_object("corbaloc:tgiop:simpapp/InterfaceRepository");
orb.string_to_object("corbaloc:tgiop:simpapp/Tobj_SimpleEventsService");
orb.string_to_object("corbaloc:tgiop:simpapp/NotificationService");
orb.string_to_object("corbaloc:tgiop:1.1@simpapp/NotificationService);
```

Examples using -ORBInitRef

You can also use the `-ORBInitRef` option to `orb.init` and `resolve_initial_reference`.

Given the following `-ORBInitRef` definitions:

```
-ORBInitRef FactoryFinder=corbaloc:tgiop:simp/FactoryFinder
-ORBInitRef InterfaceRepository=corbaloc:tgiop:simp/InterfaceRepository
-ORBInitRef
Tobj_SimpleEventService=corbaloc:tgiop:simp/Tobj_SimpleEventsService
-ORBInitRef NotificationService=corbaloc:tgiop:simp/NotificationService
```

then:

```
orb.resolve_initial_references("NameService");
orb.resolve_initial_references("FactoryFinder");
orb.resolve_initial_references("InterfaceRepository");
orb.resolve_initial_references("Tobj_SimpleEventService");
orb.resolve_initial_references("NotificationService");
```

Examples Using -ORBDefaultInitRef

You can use the `-ORBDefaultInitRef` and `resolve_initial_reference`.

Given the following `-ORBDefaultInitRef` definition:

```
-ORBDefaultInitRef corbaloc:tgiop:simpapp
```

then:

```
orb.resolve_initial_references("NameService");
```

Using the corbaname URL Format

You can also use the `corbaname` format instead of the `corbaloc` format.

Examples Using -ORBInitRef

Given the following `-ORBInitRef` definition:

```
-ORBInitRef NameService=corbaloc:tgiop:simpapp/NameService
```

```
then:
```

```
orb.string_to_object("corbaname:rir:#simple_factory");

orb.string_to_object("corbaname:tgiop:simpapp#simple_factory");

orb.string_to_object("corbaname:tgiop:1.1@simpapp#simple_factory");

orb.string_to_object("corbaname:tgiop:simpapp#simple/simple_factory");
```

How to Manage Transactions for Tuxedo CORBA Applications

Note: For more information on managing transactions in Tuxedo CORBA applications, see Overview of Transactions in BEA Tuxedo CORBA Applications at http://e-docs.bea.com/tuxedo/tux90/trans/gstrx.htm.

The WebLogic Tuxedo Connector uses the Java Transaction API (JTA) to manage transactions with Tuxedo Corba Applications. For more detailed information, see:

- Programming WebLogic JTA
- Transaction Design and Management Options

WebLogic Tuxedo Connector JATMI Transactions

The following sections provide information on global transactions and how to define and manage them in your applications:

Global Transactions

A global transaction is a transaction that allows work involving more than one resource manager and spanning more than one physical site to be treated as one logical unit. A global transaction is always treated as a specific sequence of operations that is characterized by the following four properties:

- Atomicity: All portions either succeed or have no effect.
- Consistency: Operations are performed that correctly transform the resources from one consistent state to another.
- Isolation: Intermediate results are not accessible to other transactions, although other processes in the same transaction may access the data.

- Durability: All effects of a completed sequence cannot be altered by any kind of failure.

JTA Transaction API

Note: For more detailed information, see the JTA API at http://java.sun.com/products/jta/index.htm.

The WebLogic Tuxedo Connector uses the Java Transaction API (JTA) to manage transactions.

Types of JTA Interfaces

JTA offers three types of transaction interfaces:

- Transaction
- TransactionManager
- UserTransaction

Transaction

The `Transaction` interface allows operations to be performed against a transaction in the target Transaction object. A transaction object is created to correspond to each global transaction created. Use the `Transaction` interface to enlist resources, synchronize registration, and perform transaction completion and status query operations.

TransactionManager

The `TransactionManager` interface allows the application server to communicate to the Transaction Manager for transaction boundaries demarcation on behalf of the application. Use the `TransactionManager` interface to communicate to the transaction manager on behalf of container-managed EJB components.

UserTransaction

The `UserTransaction` interface is a subset of the `TransactionManager` interface. Use the `UserTransaction` interface when it is necessary to restrict access to Transaction object.

JTA Transaction Primitives

The following table maps the functionality of Tuxedo transaction primitives to equivalent JTA transaction primitives.

Table 5-1 Mapping Tuxedo Transaction Primitives to JTA Equivalents

Tuxedo	Tuxedo Functionality	JTA Equivalent
`tpabort`	`Use to end a transaction.`	`setRollbackOnly` or `rollback`
`tpcommit`	`Use to complete a transaction.`	`commit`
`tpgetlev`	`Use to determine if a service routine is in transaction mode.`	`getStatus`
`tpbegin`	`Use to begin a transaction.`	`setTransactionTimeout` `begin`

Defining a Transaction

Transactions can be defined in either client or server processes. A transaction has three parts: a starting point, the program statements that are in transaction mode, and a termination point.

To explicitly define a transaction, call the `begin()` method. The same process that makes the call, the initiator, must also be the one that terminates it by invoking a `commit()`, `setRollbackOnly()`, or `rollback()`. Any service subroutines that are called between the transaction delimiter become part of the current transaction.

Starting a Transaction

Note: Setting `setTransactionTimeout()` to unrealistically large values delays system detection and reporting of errors. Use time-out values to ensure response to service requests occur within a reasonable time and to terminate transactions that have encountered problem, such as a network failure. For productions environments, adjust the time-out value to accommodate expected delays due to system load and database contention.

A transaction is started by a call to `begin()`. To specify a time-out value, precede the `begin()` statement with a `setTransactionTimeout(int seconds)` statement.

To propagate the transaction to Tuxedo, you must do the following:

- Look up a `TuxedoConnectionFactory` object in the JNDI.
- Get a `TuxedoConnection` object using `getTuxedoConnection()`.

Using TPNOTRAN

Service routines that are called within the transaction delimiter are part of the current transaction. However, if `tpcall()` or `tpacall()` have the flags parameter set to `TPNOTRAN`, the operations performed by the called service do not become part of that transaction. As a result, services performed by the called process are not affected by the outcome of the current transaction.

Terminating a Transaction

A transaction is terminated by a call to `commit()`, `rollback()`, or `setRollbackOnly()`. When `commit()` returns successfully, all changes to the resource as a result of the current transaction become permanent. In order for a `commit()` to succeed, the following two conditions must be met:

- The calling process must be the same one that initiated the transaction with a `begin()`
- The calling process must have no transaction replies outstanding

If either condition is not true, the call fails and an exception is thrown.

`setRollbackOnly()` and `rollback()` are used to indicate an abnormal condition and to roll back any call descriptors to their original state.

- Use `setRollbackOnly()` if further processing or cleanup is needed before rolling back the transaction.
- `Use rollback()` if no further processing or cleanup is required before rolling back the transaction.

WebLogic Tuxedo Connector Transaction Rules

You must follow certain rules while in transaction mode to insure successful completion of a transaction. The basic rules of etiquette that must be observed while in a transaction mode follow:

- You must propagate the transaction to Tuxedo using a `TuxedoConnection` object *after* you initiate a transaction with a `begin()`.
- `tpterm()` closes a connection to an object and prevents future operations on this object.

- Processes that are participants in the same transaction must require replies for their requests.
- Requests requiring no reply can be made only if the flags parameter of `tpacall()` is set to `TPNOREPLY`.
- A service must retrieve all asynchronous transaction replies before calling `commit()`.
- The initiator must retrieve all asynchronous transaction replies before calling `begin()`.
- The asynchronous replies that must be retrieved include those that are expected from non-participants of the transaction, that is, replies expected for requests made with a `tpacall()` suppressing the transaction but not the reply.
- If a transaction has not timed out but is marked abort-only, further communication should be performed with the `TPNOTRAN` flag set so that the work done as a result of the communication has lasting effect after the transaction is rolled back.
- If a transaction has timed out:
  - the descriptor for the timed out call becomes stale and any further reference to it will return `TPEBADDESC`.
  - further calls to `tpgetrply()` or `tprecv()` for any outstanding descriptors will return the global state of transaction time-out by setting `tperrono` to `TPETIME`.
  - asynchronous calls can be make with the *flags* parameter of `tpacall()` set to `TPNOREPLY | TPNOBLOCK | TPNOTRAN`.
- Once a transaction has been marked abort-only for reasons other than time-out, a call to `tpgetrply()` will return whatever represents the local state of the call, that is, it can either return success or an error code that represents the local condition.
- Once a descriptor is used with `tpgetrply()` to retrieve a reply, it becomes invalid and any further reference to it will return `TPEBADDESC`.
- Once a descriptor is used with `tpsend()` or `tprecv()`to report an error condition, it becomes invalid and any further reference to it will return `TPEV_DISCONIMM`.
- Once a transaction is aborted, all outstanding transaction call descriptions (made without the `TPNOTRAN` flag) become stale, and any further reference to them will return `TPEBADDESC`.
- WebLogic Tuxedo Connector does not guarantee that all calls for a particular transaction Id are routed to a particular server instance when load balancing. Load balancing is performed on a per call basis.

Example Transaction Code

The following provides a code example for a transaction:

Listing 5-1 Example Transaction Code

```
public class TransactionSampleBean implements SessionBean {

.....

public int transaction_sample () {

     int ret = 0;
     try {
          javax.naming.Context myContext = new InitialContext();
          TransactionManager tm = (javax.transaction.TransactionManager)
          myContext.lookup("javax.transaction.TransactionManager");

// Begin Transaction
          tm.begin ();

          TuxedoConnectionFactory tuxConFactory = (TuxedoConnectionFactory)
          ctxt.lookup("tuxedo.services.TuxedoConnection");

// You could do a local JDBC/XA-database operation here
// which will be part of this transaction.
.....

// NOTE 1: Get the Tuxedo Connection only after
// you begin the transaction if you want the
// Tuxedo call to be part of the transaction!

// NOTE 2: If you get the Tuxedo Connection before
// the transaction was started, all calls made from
// that Tuxedo Connection are out of scope of the
// transaction.

          TuxedoConnection myTux = tuxConFactory.getTuxedoConnection();

// Do a tpcall. This tpcall is part of the transaction.
          TypedString depositData = new TypedString("somecharacters,5000.00");

          Reply depositReply = myTux.tpcall("DEPOSIT", depositData, 0);

// You could also do tpcalls which are not part of
// transaction (For example, Logging all attempted
```

```
// operations etc.) by setting the TPNOTRAN Flag!
          TypedString logData =
          new TypedString("DEPOSIT:somecharacters,5000.00");

          Reply logReply = myTux.tpcall("LOGTRAN", logData,
          ApplicationToMonitorInterface.TPNOTRAN);

// Done with the Tuxedo Connection. Do tpterm.
          myTux.tpterm ();

// Commit Transaction...
          tm.commit ();

// NOTE: The TuxedoConnection object which has been
// used in this transaction, can be used after the
// transaction only if TPNOTRAN flag is set.
}
          catch (NamingException ne) {
          System.out.println ("ERROR: Naming Exception looking up JNDI: " + ne);
          ret = -1;
}
          catch (RollbackException re) {
          System.out.println("ERROR: TRANSACTION ROLLED BACK: " + re);
          ret = 0;
}
          catch (TPException te) {
          System.out.println("ERROR: tpcall failed: TpException: " + te);
          ret = -1;
}
          catch (Exception e) {
          log ("ERROR: Exception: " + e);
          ret = -1;
}

          return ret;
}
```

WebLogic Tuxedo Connector JATMI Conversations

Note: For more information on conversational communications for BEA Tuxedo, see Writing Conversational Clients and Servers at http://e-docs.bea.com/tuxedo/tux90/pgc/pgconv.htm.

The following sections provide information on conversations and how to define and manage them in your applications:

- Overview of WebLogic Tuxedo Connector Conversational Communication
- WebLogic Tuxedo Connector Conversation Characteristics
- WebLogic Tuxedo Connector JATMI Conversation Primitives
- Creating WebLogic Tuxedo Connector Conversational Clients and Servers
- Sending and Receiving Messages
- Ending a Conversation
- Executing a Disorderly Disconnect
- Understanding Conversational Communication Events
- WebLogic Tuxedo Connector Conversation Guidelines

Overview of WebLogic Tuxedo Connector Conversational Communication

WebLogic Tuxedo Connector supports BEA Tuxedo conversations as a method to exchange messages between WebLogic Server and Tuxedo applications. In this form of communication, a virtual connection is maintained between the client and the server and each side maintains information about the state of the conversation. The process that opens a connection and starts a conversation is the originator of the conversation. The process with control of the connection is the initiator; the process without control is called the subordinate. The connection remains active until an event occurs to terminate it.

During conversational communication, a half-duplex connection is established between the initiator and the subordinate. Control of the connection is passed between the initiator and the subordinate. The process that has control can send messages (the initiator); the process that does not have control can only receive messages (the subordinate).

WebLogic Tuxedo Connector Conversation Characteristics

WebLogic Tuxedo Connector JATMI conversations have the following characteristics:

- Data is passed using `TypedBuffers`. The type and sub-type of the data must match one of the types and sub-types recognized by the service.
- The logical connection between the conversational client and the conversational server remains active until it is terminated.
- Any number of messages can be transmitted across a connection between a conversational client and the conversational server.
- A WebLogic Tuxedo Connector conversational client initiates a request for service using `tpconnect` rather than a `tpcall` or `tpacall`.
- WebLogic Tuxedo Connector conversational clients and servers use the JATMI primitives `tpsend` to send data and `tprecv` to receive data.
- A conversational client only sends service requests to a conversational server.
- Conversational servers are prohibited from making calls to `tpforward`.

WebLogic Tuxedo Connector JATMI Conversation Primitives

Use the following WebLogic Tuxedo Connector primitives when creating conversational clients and servers that communicate between WebLogic Server and Tuxedo:

Table 6-1 WebLogic Tuxedo Connector Conversational Client Primitives

Name	Operation
tpconnect	Use to establish a connection to a Tuxedo conversational service.
tpdiscon	Use to abort a connection and generate a `TPEV_DISCONIMM` event.
tprecv	Use to receive data across an open connection from a Tuxedo application.
tpsend	Use to send data across a open connection to a Tuxedo application.

Creating WebLogic Tuxedo Connector Conversational Clients and Servers

The following sections provide information on how to create conversational clients and servers.

Creating Conversational Clients

Follow the steps outlined in "Developing WebLogic Tuxedo Connector Client EJBs" on page 2-1 to create WebLogic Tuxedo Connector conversational clients. The following section provide information on how to use `tpconnect` to open a connection and start a conversation.

Establishing a Connection to a Tuxedo Conversational Service

A WebLogic Tuxedo Connector conversational client must establish a connection to the Tuxedo conversational service. Use the JATMI primitive `tpconnect` to open a connection and start a conversation. A successful call returns an object that can be used to send and receive data for a conversation.

The following table describes `tpconnect` parameters:

Table 6-2 WebLogic Tuxedo Connector JATMI tpconnect Parameters

Parameter	Description
svc	Character pointer to a conversational service name. If you do not specify a *svc,* the call will fail and `TPException` is set to `TPEV_DISCONIMM`.
data	Pointer to the data buffer. When establishing a connection, you can send data simultaneously by setting the *data* parameter to point to a buffer. The `type` and `subtype` of the buffer must be recognized by the service being called. You can set the value of *data* to NULL to specify that no data is to be sent.
flags	Use flags or combinations of flags as required by your application needs. Valid flag values are: `TPSENDONLY`: specifies that the control is being retained by the originator. The called service is subordinate and can only receive data. Do not use in combination with `TPRECVONLY`. `TPRECVONLY`: specifies that control is being passed to the called service.The originator becomes subordinate and can only receive data. Do not use in combination with `TPSENDONLY`. `TPNOTRAN`: specifies that when *svc* is invoked and the originator is transaction mode, *svc* is not part of the originator's transaction. A call remains subject to transaction timeouts. If *svc* fails, the originator's transaction is unaffected. `TPNOBLOCK`: specifies that a request is not sent if a blocking condition exists. If `TPNOBLOCK` is not specified, the originator blocks until the condition subsides, a transaction timeout occurs, or a blocking timeout occurs. `TPNOTIME`: specifies that the originator will block indefinitely and is immune to blocking timeouts. If the originator is in transaction mode, the call is subject to transaction timeouts.

Example TuxedoConversationBean.java Code

The following provides a code example to use `tpconnect` to start a conversation:

Listing 6-1 Example Conversation Code

```
.
.
.
Context ctx;
Conversation myConv;
TuxedoConnection myTux;
TuxedoConnectionFactory tcf;
.
.
.
ctx = new InitialContext();
tcf = (TuxedoConnectionFactory) ctx.lookup
("tuxedo.services.TuxedoConnection");
myTux = tcf.getTuxedoConnection();
flags =ApplicationToMonitorInterface.TPSENDONLY;
myConv = myTux.tpconnect("CONNECT_SVC",null,flags);
.
.
.
```

Creating WebLogic Tuxedo Connector Conversational Servers

Follow the steps outlined in "Developing WebLogic Tuxedo Connector Service EJBs" on page 3-1 to create WebLogic Tuxedo Connector conversational servers.

Sending and Receiving Messages

Once a conversational connection is established between a WebLogic Server application and a Tuxedo application, the communication between the initiator (sends message) and subordinate (receives message) is accomplished using send and receive calls. The following sections describe how WebLogic Tuxedo Connector applications use the JATMI primitives `tpsend` and `tprecv`:

- Sending Messages
- Receiving Messages

Sending Messages

Use the JATMI primitive tpsend to send a message to a Tuxedo application.

The following table describes tpsend parameters:

Table 6-3 WebLogic Tuxedo Connector JATMI tpconnect Parameters

Parameter	Description
data	Pointer to the buffer containing the data sent with this conversation.
flags	The flag can be one of the following: TPRECVONLY: specifies that after the initiator's data is sent, the initiator gives up control of the connection. The initiator becomes subordinate and can only receive data. TPNOBLOCK: specifies that the request is not sent if a blocking condition exists. If TPNOBLOCK is not specified, the originator blocks until the condition subsides, a transaction timeout occurs, or a blocking timeout occurs. TPNOTIME: specifies that an initiator is willing to block indefinitely and is immune from blocking timeouts. The call is subject to transaction timeouts.

Receiving Messages

Use the JATMI primitive tprecv to receive messages from a Tuxedo application.

The following table describes tprecv parameters:

Table 6-4 WebLogic Tuxedo Connector JATMI tpconnect Parameters

Parameter	Description
flags	The flag can be one of the following: TPNOBLOCK: specifies that tprecv does not wait for a reply to arrive. If a reply is available, tprecv gets the reply and returns. If this flag is not specified and a reply is not available, tprecv waits for one of the following to occur: a reply, a transaction timeout, or a blocking timeout. TPNOTIME: specifies that tprecv waits indefinitely for a reply. With this flag, tprecv is immuned from blocking timeouts but is still subject to transaction timeouts. A flag value of 0 specifies that the initiator blocks until the condition subsides or a timeout occurs.

Ending a Conversation

A conversation between WebLogic Server and Tuxedo ends when the server process successfully completes its tasks. The following sections describe how a conversation ends:

- Tuxedo Application Originates Conversation
- WebLogic Tuxedo Connector Application Originates Conversation
- Ending Hierarchical Conversations

Tuxedo Application Originates Conversation

A WebLogic Server conversational server ends a conversation by a successful call to return. A TPEV_SVCSUCC event is sent to the Tuxedo client that originated connection to indicate that the service finished successfully. The connection is then disconnected in an orderly manner.

WebLogic Tuxedo Connector Application Originates Conversation

A Tuxedo conversational server ends a conversation by a successful call to tpreturn. A TPEV_SVCSUCC event is sent to the WebLogic Tuxedo Connector client that originated connection to indicate that the service finished successfully. The connection is then disconnected in an orderly manner.

Ending Hierarchical Conversations

The order in which an conversation ends is important to gracefully end hierarchal conversations.

Assume there are two active connections: A-B and B-C. If B is a WebLogic Tuxedo Connector application in control of both connections, a call to `return` has the following effect: the call fails and a `TPEV_SVCERR` event is posted on all open connections, and the connections are closed in a disorderly manner.

In order to terminate both connections in an orderly manner, the application must execute the following sequence:

1. B calls `tpsend` with `TPRECVONLY` to transfer control of the B-C connection to the Tuxedo application C.
2. C calls `departure` with `rval` set to `TPSUCCESS`, `TPFAIL`, or `TPEXIT`.
3. B calls `return` and posts an event (`TPEV_SVCSUCC` or `TPEV_SVCFAIL`) for A.

Conversational services can make request/response calls. Therefore, in the preceding example, the calls from B to C may be executed using `tpacall()`or `tpcall()`instead of `tpconnect`. Conversational services are not permitted to make calls to `tpforward`.

Executing a Disorderly Disconnect

WebLogic Server conversational clients or servers execute a disorderly disconnect is through a call to `tpdiscon`. This is the equivalent of “pulling the plug” on a connection.

A call to `tpdiscon`:

- Immediately tears down the connection and generates a `TPEV_DISCONIMM` at the other end of the connection. Any data that has not yet reached its destination may be lost. If the conversation is part of a transaction, the transaction must be rolled back.
- Can only be called by the initiator of the conversation.

Understanding Conversational Communication Events

WebLogic Tuxedo Connector JATMI uses five events to manage conversational communication. The following table lists the events, the functions for which they are returned, and a detailed description of each.

Table 6-5 WebLogic Tuxedo Connector Conversational Communication Events

Event	Received by	Description
TPEV_SENDONLY	Tuxedo `tprecv`	Control of the connection has passed; this Tuxedo process can now call `tpsend`
	JATMI `tprecv`	Control of the connection has passed; this JATMI process can now call `tpsend`
TPEV_DISCONIMM	Tuxedo `tprecv, tpsend, tpreturn`	The connection has been torn down and no further communication is possible. The JATMI `tpdiscon` posts this event in the originator of the connection. The originator sends it to all open connections when `tpreturn` is called. Connections are closed in a disorderly manner and if a transaction exists, it is aborted.
	JATMI `tprecv, tpsend, return`	The connection has been torn down and no further communication is possible. The Tuxedo `tpdiscon` posts this event in the originator of the connection. The originator sends it to all open connections when `return` is called. Connections are closed in a disorderly manner and if a transaction exists, it is aborted.
TPEV_SVCERR	Tuxedo `tpsend` or JATMI `tpsend`	Received by the originator of the connection indicating that the subordinate program issued a `tpreturn` (Tuxedo) or `return` (JATMI) and ended without control of the connection.
	Tuxedo `tprecv` or JATMI `tprecv`	Received by the originator of the connection indicating that the subordinate program issued a successful `tpreturn` (Tuxedo) or a successful `return` (JATMI) without control of the connection, but an error occurred before the call completed.

Table 6-5 WebLogic Tuxedo Connector Conversational Communication Events

Event	Received by	Description
TPEV_SVCSUCC	Tuxedo `tprecv`	Received by the originator of the connection, indicating that the subordinate service finished successfully; that is, `return` was successfully called.
	JATMI `tprecv`	Received by the originator of the connection, indicating that the subordinate service finished successfully; that is, `tpreturn` was called with TPSUCCESS.
TPEV_SVCFAIL	Tuxedo `tpsend` or JATMI `tpsend`	Received by the originator of the connection indicating that the subordinate program issued a `tpreturn` (Tuxedo) or `return` (JATMI) and ended without control of the connection. The service completed with status of TPFAIL or TPEXIT and the data is set to null.
	Tuxedo `tprecv` or JATMI `tprecv`	Received by the originator of the connection indicating that the subordinate program finished unsuccessfully. The service completed with status of TPFAIL or TPEXIT.

WebLogic Tuxedo Connector Conversation Guidelines

Use the following guidelines while in conversation mode to insure successful completion of a conversation:

- Use the JATMI conversational primitives as defined in the WebLogic Tuxedo Connector `Conversation` interface and `ApplicationToMonitorInterface` interface.
  - Always use a flag.
  - Only use flags defined in the WebLogic Tuxedo Connector JATMI.
- WebLogic Tuxedo Connector does not have a parameter that can be used to limit the number of simultaneous conversations to prevent overloading the WebLogic Server network.
- If Tuxedo exceeds the maximum number of possible conversations (defined by the `MAXCONV` parameter), `TPEV_DISCONIMM` is the expected WebLogic Tuxedo Connector exception value.
- A `tprecv` to an unauthorized Tuxedo service results in a `TPEV_DISCONIMM` exception value.

- If a WebLogic Tuxedo Connector client is connected to a Tuxedo conversational service which does `tpforward` to another conversational service, `TPEV_DISCONIMM` is the expected WebLogic Tuxedo Connector exception value.
- Conversations may be initiated within a transaction. Start the conversation as part of the program statements in transaction mode. For more information on transactions, see "WebLogic Tuxedo Connector JATMI Transactions" on page 5-1.
- If a WebLogic Tuxedo Connector remote domain experiences a `TPENOENT`, the remote domain will send back a disconnect event message and be caught on the WebLogic Tuxedo Connector application `tprecv` as a `TPEV_DISCONIMM` exception.

Using FML with WebLogic Tuxedo Connector

The following sections discuss the Field Manipulation Language (FML) and describe how the WebLogic Tuxedo Connector uses FML.

Overview of FML

Note: For more information about using FML, see Programming a BEA Tuxedo Application Using FML at http://e-docs.bea.com/tuxedo/tux90/fml/fml01.htm.

FML is a set of java language functions for defining and manipulating storage structures called fielded buffers. Each fielded buffer contains attribute-value pairs in fields. For each field:

- The attribute is the field's identifier.
- The associated value represents the field's data content.
- An occurrence number.

There are two types of FML:

- FML16 based on 16-bit values for field lengths and identifiers. It is limited to 8191 unique fields, individual field lengths of 64K bytes, and a total fielded buffer size of 64K bytes.
- FML32 based on 32-bit values for the field lengths and identifiers. It allows for about 30 million fields, and field and buffer lengths of about 2 billion bytes.

The WebLogic Tuxedo Connector FML API

Note: The WebLogic Tuxedo Connector implements a subset of FML functionality. For more information regarding FML32, refer to "FML32 Considerations."

The FML application program interface (API) is documented in the `weblogic.wtc.jatmi` package included in the Javadocs for WebLogic Server Classes.

FML Field Table Administration

Field tables are generated in a manner similar to Tuxedo field tables. The field tables are text files that provide the field name definitions, field types, and identification numbers that are common between the two systems. To interoperate with a Tuxedo system using FML, the following steps are required:

1. Copy the field tables from the Tuxedo system to WebLogic Tuxedo Connector environment.

   For example: Your Tuxedo distribution contains a bank application example called `bankapp`. It contains a file called `bankflds` that has the following structure:

   ```
   #Copyright (c) 1990 Unix System Laboratories, Inc.
   #All rights reserved
   #ident  "@(#) apps/bankapp/bankflds     $Revision: 1.3 $"
   # Fields for database bankdb

   # name                         number   type    flags   comments
   ACCOUNT_ID                     110      long    -       -
   ACCT_TYPE                      112      char    -       -
   ADDRESS                        109      string  -       -
   .
   .
   .
   ```

2. Converted the field table definition into Java source files. Use the `mkfldclass` utility supplied in the `weblogic.wtc.jatmi` package. This class is a utility function that reads a

FML32 Field Table and produces a Java file which implements the FldTbl interface. There are two instances of this utility:

- `mkfldclass`
- `mkfldclass32`

Use the correct instance of the command to convert the `bankflds` field table into FML32 java source. The following example uses mkfldclass.

```
java weblogic.wtc.jatmi.mkfldclass bankflds
```

The resulting file is called `bankflds.java` and has the following structure:

```
import java.io.*;
import java.lang.*;
import java.util.*;
import weblogic.wtc.jatmi.*;

public final class bankflds
        implements weblogic.wtc.jatmi.FldTbl
{
        /** number: 110  type: long */
        public final static int ACCOUNT_ID = 33554542;
        /** number: 112  type: char */
        public final static int ACCT_TYPE = 67108976;
        /** number: 109  type: string */
        public final static int ADDRESS = 167772269;
        /** number: 117  type: float */
.
.
.
```

3. Compile the resulting `bankflds.java` file using the following command:

   ```
   javac bankflds.java
   ```

   The result is a `bankflds.class` file. When loaded, the WebLogic Tuxedo Connector uses the class file to add, retrieve and delete field entries from an FML32 field.

4. Add the field table class file to your application CLASSPATH.

5. Update your `WTCServer` MBean.

   – Update the `WTCResources` MBean to reflect the fully qualified location of the field table class file.

   – Use the keywords required to describe the FML buffer type: fml16 or fml32.

– You can enter multiple field table classes in a comma separated list.

For example:

```
<wtc-resources>
     <name>BankappResources</name>
     <fld-tbl16-class>my.bankflds</fld-tbl16-class>
     <fld-tbl16-class>your.bankflds</fld-tbl16-class>
     <fld-tbl16-class>more.bankflds</fld-tbl16-class>
</wtc-resources>
```

6. Restart your WebLogic Server to load the field table class definitions.

Using the DynRdHdr Property for mkfldclass32 Class

WebLogic Tuxedo Connector provides a property that provides an alternate method to compile FML tables. You may need to use the `DynRdHdr` utility if:

- You are using very large FML tables and the `.java` method created by the mkfldclass32 class exceeds the internal Java Virtual Machine limit on the total complexity of a single class or interface.
- You are using very large FML tables and are unable to load the class created when compiling the `.java` method.

Use the following steps to use the `DynRdHdr` property when compiling your FML tables:

1. Convert the field table definition into Java source files.

   ```
   java -DDynRdHdr=Path_to_Your_FML_Table weblogic.wtc.jatmi.mkfldclass32
   userTable
   ```

   The arguments for this command are defined as follows:

Attribute	Description
`-DDynRdHdr`	WebLogic Tuxedo Connector property used to compile an FML table.
`Path_to_Your_FML_Table`	Path name of your FML table. This may be either a fully qualified path or a relative path that can be found as a resource file using the server's CLASSPATH.

Attribute	Description
`weblogic.wtc.jatmi.mkfldclass32`	This class is a utility function that reads an FML32 Field Table and produces a Java file which implements the FldTbl interface.
`userTable`	Name of the `.java` method created by the mkfldclass32 class.

2. Compile the *`userTable`* file using the following command:

   `javac` *`userTable`*`.java`

3. Add the *`userTable`*`.class` file to your application CLASSPATH.

4. Update the `WTCResources` MBean to reflect the fully qualified location of the *`userTable`*`.class` file.

5. Target your WTCServer. The *`userTable`*`.class` is loaded when the WTCServer service starts.

Once you have created the *`userTable`*`.class` file, you can modify the FML table and deploy the changes without having to manually create an updated *`userTable`*`.class`. When the WTC Service is started, WebLogic Tuxedo Connector will load the updated FML table using the location specified in the Resources tab of your WTC service configuration. If the *`Path_to_Your_FML_Table`* attribute changes, you will need to use the preceding procedure to update your *`userTable`*`.java` and *`userTable`*`.class` files.

Using TypedFML32 Constructors

Two new constructors for TypedFML32 are available to improve performance. The following topic provides explanation as to when to use these constructors.

The constructors are defined in the Javadocs for WebLogic Server Classes.

Gaining TypedFML32 Performance Improvements

To gain TypedFML32 performance improvements, you can choose to give size hints to TypedFML32 constructors. There are two parameters that are available to those constructor:

- A parameter that hints for maximum number of fields. This includes all the occurrences.
- A parameter for the total number of field IDs used in the buffer.

For instance, a field table used by the buffer contains 20 field IDs, and each field can occur 20 times. In this case, the first parameter should be 400 for the maximum number of fields. The second parameter should be 20 for the total number of field IDs.

```
TypeFML32 mybuffer = new TypeFML32(400, 20);
```

Note: This usually works well with any size of buffer; however, it does not work well with extremely small buffers.

If you have an extremely small buffer, use those constructor without hints. An example of an extremely small buffer is a buffer with less than 16 total occurrences. If the buffer is extremely large, for example contains more than 250000 total field occurrences, then the application should consider splitting it into several buffers smaller than 250000 total field occurrences.

tBridge XML/FML32 Translation

Note: The data type specified must be FLAT or NO. If any other data type is specified, the redirection fails.

The `TranslateFML` element of the WTCtBridgeRedirect MBean is used to indicate if FML32 translation is performed on the message payload. There are two types of FML32 translation: FLAT and NO.

FLAT

The message payload is translated using the WebLogic Tuxedo Connector internal FML32/XML translator. Fields are converted field-by-field values without knowledge of the message structure (hierarchy) and repeated grouping.

In order to convert an FML32 buffer to XML, the tBridge pulls each instance of each field in the FML32 buffer, converts it to a string, and places it within a tag consisting of the field name. All of these fields are placed within a tag consisting of the service name. For example, an FML32 buffer consisting of the following fields:

```
NAME          JOE
ADDRESS       CENTRAL CITY
PRODUCTNAME   BOLT
PRICE         1.95
PRODUCTNAME   SCREW
PRICE         2.50
```

The resulting XML buffer would be:

```
<FML32>
  <NAME>JOE</NAME>
  <ADDRESS>CENTRAL CITY</ADDRESS>
  <PRODUCTNAME>BOLT</PRODUCTNAME>
  <PRODUCTNAME>SCREW</PRODUCTNAME>
  <PRICE>1.95</PRICE>
  <PRICE>2.50</PRICE>
</FML32>
```

NO

No translation is used.

For JMS to Tuxedo, the tBridge maps a JMS TextMessage into a Tuxedo TypedBuffer (TypedString) and vice versa depending on the direction of the redirection. JMS BytesMessage are mapped into Tuxedo TypedBuffer (TypedCarray) and vice versa.

For Tuxedo to JMS, passing an FML/FML32 buffer behaves as if `translateFML` is set to `FLAT`. Therefore, in this case, setting `translateFML` to `NO` has no effect and if the Tuxedo buffer is of type FML/FML32, the translation takes place automatically.

FML32 Considerations

Remember to consider the following information when working with FML32:

- For XML input, the root element is required but ignored.
- For XML output, the root element is always <FML32>.
- The field table names must be loaded as described in "FML Field Table Administration" on page 7-2.
- The tBridge translator is capable of only "flat" or linear grouping. This means that information describing FML32 ordering is not maintained, therefore buffers that contain a series of repeating data could be presented in an unexpected fashion. For example, consider a FML32 buffer that contains a list of parts and their associated price. The expectation would be PART A, PRICE A, PART B, PRICE B, etc. however since there is no structural group information contained within the tBridge, the resulting XML could be PART A, PART B, etc., PRICE A, PRICE B, etc.

- When translating XML into FML32, the translator ignores `STRING` values. For example, `<STRING></STRING>` is skipped in the resulting FML32 buffer. All other types cause WTC to log an error resulting in translation failure.
- Embedded FML is not supported in this release.
- Embedded VIEW fields within FML32 buffers are supported in this release.
- TypedCArray is supported for FML/FML32 to XML conversion. Select from the following list of supported field types:
  - SHORT
  - LONG
  - CHAR
  - FLOAT
  - DOUBLE
  - STRING
  - CARRAY
  - INT (FML32)
  - DECIMAL (FML32)
- If you need to pass binary data, encode to a field type of your choice and decode the XML on the receiving side.
- If you need to use CARRAY fields in an XML input buffer, you must first encode the content using base64. You must decode the base64 data after it is received and before it is processed by an application.

Using the XmlFmlCnv Class for XML to and From FML/FML32 Translation

An alternative option to using the tBridge to automatically translate XML buffers to and from FML/FML32 is to use the `XmlFmlCnv` class which supports ordering, grouping and beautifying functionality. The following code listing is an example that uses the `XmlFmlCnv` class for conversion to and from XML buffer formats.

```
import weblogic.wtc.jatmi.TypedFML32;
```

```
import weblogic.wtc.jatmi.FldTbl;
import weblogic.wtc.gwt.XmlFmlCnv;

public class xml2fml
{
public static void main(String[] args) {
       String xmlDoc = "<XML><MyString>hello</MyString></XML>";
       TypedFML32 fmlBuffer = new TypedFML32(new MyFieldTable());
       XmlFmlCnv c = new XmlFmlCnv();
       fmlBuffer = c.XMLtoFML32(xmlDoc, fmlBuffer.getFieldTables());
       String result = c.FML32toXML(fmlBuffer);
       System.out.println(result);
}
}
```

See Class XmlFmlCnv.

Limitations of XmlFmlCnv Class

The `FLD_MBSTRING` field in FML32 is not supported by the `XmlFmlCnv.FML32toXML` method in this release.

MBSTRING Usage

A `TypedMBString` object can be used almost identically as a `TypedString` object in a WTC application code. The only difference is that `TypedMBString` has a codeset encoding name associated to the string data.

This section includes the following topics.

- Sending MBSTRING Data to a Tuxedo Domain
- Receiving MBSTRING Data from a Tuxedo Domain
- Using FML with WebLogic Tuxedo Connector

Sending MBSTRING Data to a Tuxedo Domain

When a Tuxedo message that contains an MBSTRING data is sent to another Tuxedo domain, `TypedMBString` uses the conversion function of `java.lang.String` class to convert between

Unicode and an external encoding. The `TypedMBString` has a codeset encoding name associated to the string data.

When a `TypedMBString` object is created by a WTC application code, the encoding name is set to null. The null value of the encoding name means that the default encoding name is used for Unicode string to byte array conversion while sending the MBSTRING data to a remote domain. By default, the Java's default encoding name for byte array string is used for the default encoding name.You can specify encoding or accept the default encoding. The following order defines the order of precedence for `TypedMBString`.

1. Specify the encoding name by `setMBEncoding()` method.
2. Specify the encoding name through the `setDefaultMBEncoding()` method of `weblogic.wtc.jatmi.MBEncoding` class.
3. Specify the encoding name through the `RemoteMBEncoding` attribute of the `WTCResourcesMBean`.
4. `MBENCODINGPROPERTY` system property value.
5. Accept the Java default encoding name.

Receiving MBSTRING Data from a Tuxedo Domain

When a Tuxedo message that contains an MBSTRING data is received from a remote domain, the following actions take place.

1. WTC determines the encoding of the MBSTRING data by the codeset `tcm` in the received message.
2. WTC creates a `TypedMBString` object.

   A `TypedMBString` object can be used almost identically as a `TyepdString` object in WTC application code. However, the TypedMBString has a codeset encoding name associated to the string data.
3. WTC passes the `TypedMBString` object to the WTC application code. The application code knows the encoding of the received MBSTRING data by the instance method `getMBEncoding()`.

Using FML with WebLogic Tuxedo Connector

FLD_MBSTRING is a field type added to `TypedFML32`. In this case, a `TypedMBString` object is passed to the `TypedFML32` method as the associated object type of FLD_MBSTRING. You can specify the encoding name used for the MBSTRING conversion for a FLD_MBSTRING field.

The following order defines the order of precedence for TypedFML32.

1. Specify the encoding name by `setMBEncoding()` method of the TypedMBString object for the field.
2. Specify the encoding name by `setMBEncoding()` method of the TypedFML32 object.
3. Specify the encoding name through the `setDefaultMBEncoding()` method of `weblogic.wtc.jatmi.MBEncoding` class.
4. Specify the encoding name through the `RemoteMBEncoding` attribute of the `WTCResourcesMBean`.
5. `MBENCODINGPROPERTY` system property value.
6. Accept the Java default encoding name.

Notes: The following methods must be updated when using FLD_MBSTRING: `Fldtype()`, `Fchg()`, `Fadd()`, `Fget()`, and `Fdel()`.

The on-demand encoding methods and auto-conversion methods needed in Tuxedo, such as `Fmbpack32()` and `Fmbunpack32()` are not needed by WebLogic Tuxedo Connector.

WebLogic Tuxedo Connector JATMI VIEWs

The following sections provide information about how to use WebLogic Tuxedo Connector VIEW buffers:

- Overview of WebLogic Tuxedo Connector VIEW Buffers
- How to Create a VIEW Description File
- How to Use the viewj Compiler
- How to Pass Information to and from a VIEW Buffer
- How to Use VIEW Buffers in JATMI Applications
- Using the XmlViewCnv Class for XML to and From View/View(32) Translation

Note: The TypedView and TypedView32 buffer types are missing from the WLS Javadocs, see `TypedView` and `TypedView32` instead.

Overview of WebLogic Tuxedo Connector VIEW Buffers

Note: For more information on Tuxedo VIEW buffers, see Using a VIEW Typed Buffer at http://e-docs.bea.com/tuxedo/tux90/pgc/pgbuf.htm#1262459.

WebLogic Tuxedo Connector allows you to create a Java VIEW buffer type analogous to a Tuxedo VIEW buffer type derived from an independent C structure. This allows WebLogic Server applications and Tuxedo applications to pass information using a common structure. WebLogic Tuxedo Connector VIEW buffers do not support FML VIEWs or FML VIEWs/Java conversions.

How to Create a VIEW Description File

Note: `fbname` and `null` fields are not relevant for independent Java and C structures and are ignored by the Java and C VIEW compiler. You must include a value (for example, a dash) as a placeholder in these fields.

Your WebLogic Server application and your Tuxedo application must share the same information structure as defined by the VIEW description. The following format is used for each structure in the VIEW description file:

```
$ /* VIEW structure */
VIEW viewname
type cname fbname count flag size null
```

where

- The file name is the same as the VIEW name.
- You can have only one VIEW description per file.
- The VIEW description file is the same file used for both the WebLogic Tuxedo Connector `viewj` compiler and the Tuxedo `viewc` compiler.
- *`viewname`* is the name of the information structure.
- You can include a comment line by prefixing it with the # or $ character.
- The following table describes the fields that must be specified in the VIEW description file for each structure.

Table 8-1 VIEW Description File Fields

Field	Description
type	Data type of the field. Can be set to `short`, `long`, `float`, `double`, `char`, `string`, `carray`, or `dec_t` (packed decimal).
cname	Name of the field as it appears in the information structure.
fbname	Ignored.
count	Number of times field occurs.

Table 8-1 VIEW Description File Fields

Field	Description
flag	Specifies any of the following optional flag settings: • N—zero-way mapping • C—generate additional field for associated count member (ACM) • L—hold number of bytes transferred for STRING and CARRAY
size	For `STRING` and `CARRAY` buffer types, specifies the maximum length of the value. This field is ignored for all other buffer types.
null	User-specified NULL value, or minus sign (`-`) to indicate the default value for a field. NULL values are used in `VIEW` typed buffers to indicate empty C structure members. The default NULL value for all numeric types is 0 (0.0 for `dec_t`). For character types, the default NULL value is `\0'. For `STRING` and `CARRAY` types, the default NULL value is " ". Constants used, by convention, as escape characters can also be used to specify a NULL value. The VIEW compiler recognizes the following escape constants: `\ddd` (where `d` is an octal digit), `\0`, `\n`, `\t`, `\v`, `\r`, `\f`, `\\`, `\'`, and `\"`. You may enclose `STRING`, `CARRAY`, and `char` NULL values in double or single quotes. The VIEW compiler does not accept unescaped quotes within a user-specified NULL value. You can also specify the keyword NONE in the NULL field of a VIEW member description, which means that there is no NULL value for the member. The maximum size of default values for string and character array members is 2660 characters.

Example VIEW Description File

The following provides an example VIEW description which uses VIEW buffers to send information to and receive information from a Tuxedo application. The file name for this VIEW is `infoenc`.

Listing 8-1 Example VIEW Description

```
VIEW infoenc
#type    cname    fbname  count flag size null
float    amount   AMOUNT  2     -    -    0.0
short    status   STATUS  2     -    -    0
int      term     TERM    2     -    -    0
char     mychar   MYCHAR  2     -    -    -
string   name     NAME    1     -    16   -
carray   carray1  CARRAY1 1     -    10   -
dec_t    decimal  DECIMAL 1     -    9    - #size ignored by
viewj/viewj32
END
```

How to Use the viewj Compiler

To compile a `VIEW` typed buffer, run the `viewj` command, specifying the package name and the name of the VIEW description file as arguments. The output file is written to the current directory.

To use the `viewj` compiler, enter the following command:

```
java weblogic.wtc.jatmi.viewj [options] [package] viewfile
```

To use the `viewj32` compiler, enter the following command:

```
java weblogic.wtc.jatmi.viewj32 [options] [package] viewfile
```

The arguments for this command are defined as follows:

<table>
<tr><th>Argument</th><th>Description</th></tr>
<tr><td><code>options</code></td><td>

<code>-associated_fields</code>:
Use to set <code>AssociatedFieldHandling</code> to true. This allows <code>set</code> and <code>get</code> accessor methods to use the values of the associated length and count fields if they are specified in the VIEW description file. If not specified, the default value for <code>AssociatedFieldHandling</code> is false.
<code>-bean_names</code>:
Use to create <code>set</code> and <code>get</code> accessor names that follow JavaBeans naming conventions. The first character of the field name is changed to upper case before the <code>set</code> or <code>get</code> prefix is added. The signature of indexed <code>set</code> accessors for array fields changes from the default signature of <code>void setAfield(T value, int index)</code> to <code>void setAfield(int index, T value).</code>
<code>-compat_names</code>:
Use to create <code>set</code> and <code>get</code> accessor names that are formed by taking the field name from the VIEW description file and adding a set or get prefix. Provides compatibility with releases prior to WebLogic Server 8.1 SP2. Default value is <code>-compat_names</code> if <code>-bean_names or -compat_names is not</code> specified.
<code>-modify_strings</code>:
Use to generate different Java code for encoding strings sent to Tuxedo and decoding strings received from Tuxedo. Encoding code adds a null character to the end of each string. Decoding code truncates each string at the first null character received.
<code>-xcommon</code>:
Use to generate output class as extending TypedXCommon instead of TypedView.
<code>-xtype</code>:
Use to generate output class as extending TypedXCType instead of TypedView.

Note: <code>-compat_names</code> and <code>-bean_names</code> are mutually exclusive options.
</td></tr>
</table>

Argument	Description
package	The package name to be included in the .java source file. Example: examples.wtc.atmi.simpview
viewfile	Name of the VIEW description file. Example: Infoenc

For example:

- A VIEW buffer is compiled as follows:

```
java weblogic.wtc.jatmi.viewj -compat_names examples.wtc.atmi.simpview inf
oenc
```

- A VIEW32 buffer is compiled as follows:

```
java weblogic.wtc.jatmi.viewj32 -compat_names -modify_strings examples.wtc
.atmi.simpview infoenc
```

How to Pass Information to and from a VIEW Buffer

The output of the `viewj` and `viewj32` command is a `.java` source file that contains `set` and `get` accessor methods for each field in the VIEW description file. Use these `set` and `get` accessor methods in your Java applications to pass information to and from a VIEW buffer.

The `AssociatedFieldHandling` flag is used to specify if the `set` and `get` methods use the values of the associated length and count fields if they are specified in the VIEW description file.

- `set` methods set the count for an array field and set the length for a string or carray field.
- Array `get` methods return an array that is at most the size of the associated count field.
- String and carray `get` methods return data that is at most the length of the associated length field.

Use one of the following to set or get the state of the `AssociatedFieldHandling` flag:

- Use the `-associated_fields` option for the `viewj` and `viewj32` compiler to set the `AssociatedFieldHandling` flag to true.
- Invoke the `void setAssociatedFieldHandling(boolean state)` method in your Java application to set the state of the `AssociatedFieldHandling` flag.

  - If false, the `set` and `get` methods ignore the length and count fields.
  - If true, the `set` and `get` methods use the values of the associated length and count fields if they are specified in the VIEW description file.
  - The default state is false.
- Invoke the `boolean getAssociatedFieldHandling()` method in your Java application to return the current state of `AssociatedFieldHandling`.

How to Use VIEW Buffers in JATMI Applications

Note: The TypedView and TypedView32 buffer types are missing from the WLS Javadocs, see `TypedView` and `TypedView32` instead.

Use the following steps when incorporating VIEW buffers in your JATMI applications:

1. Create a VIEW description file for your application as described in How to Create a VIEW Description File.
2. Compile the VIEW description file as described in How to Use the viewj Compiler.
3. Use the set and get accessor methods to pass information to and receive information from a VIEW buffer as described in How to Pass Information to and from a VIEW Buffer.

   See the `examples/wtc/atmi/simpview/ViewClient.java` file in your WebLogic Server distribution for an example of how a client uses accessors to pass information to and from a VIEW buffer.

   Note: For this release, WTC samples are available on the BEA dev2dev website in the Code Library.
4. Import the output of the VIEW compiler into your source code.
5. If necessary, compile the VIEW description file for your Tuxedo application and include the output in your C source file as described in Using a VIEW Typed Buffer at http://e-docs.bea.com/tuxedo/tux90/pgc/pgbuf.htm.
6. Configure a WTCServer MBean with a Resources Mbean that specifies the VIEW buffer type (VIEW or VIEW32) and the fully qualified class name of the compiled Java VIEW description file. The class of the compiled Java VIEW description file should be in your `CLASSPATH`.
7. Build and launch your Tuxedo application.
8. Build and launch your WebLogic Server Application.

How to Get VIEW32 Data In and Out of FML32 Buffers

A helper class is available to add and get VIEW32 data in and out of an FML32 buffer. The class name is `wtc.jatmi.FViewFld`. This class assists programmers in developing JATMI-based applications that use VIEW32 field type for FML32 buffers.

No change to configuration is required. You still configure the VIEW32 class path using the `ViewTbl32Classes` attribute in the `WTCResources` section of the WLS configuration file.

The following access methods are available in this helper class.

- `FViewFld(String vname, TypedView32 vdata);`
- `FviewFld(FviewFld to_b_clone);`
- `void setViewName(String vname)`
- `String getViewName();`
- `void setViewData(TypedView32 vdata)`
- `void TypedView32 getViewData();`

Listing 8-2 How to Add and Retrieve an Embedded TypedView32 buffer in a TypedFML32 Buffer

```
String toConvert = new String("hello world");
TypedFML32 MyData = new TypedFML32(new MyFieldTable());
Long d1 = new Long(1234);
Float d2 = new Float(12.32);
MyView data = new myView();
FviewFld vfld;
data.setamount((float)100.96);
data.setstatus((short)3);
vfld = new FviewFld("myView", data);

try {
  myData.Fchg(MyFieldTable.FLD0, 0, toConvert);
  myData.Fchg(MyFieldTable.FLD1, 0, 1234);
  myData.Fchg(MyFieldTable.FLD2, 0, d2);
  myData.Fchg(MyFieldTable.myview, 0, vfld);
} catch (Ferror fe) {
  log("An error occurred putting data into the FML32 buffer.  The error is
```

```
" + fe);
}

try {
  myRtn = myTux.tpcall("FMLVIEW", myData, 0);
} catch(TPReplyException tre) {
….
}
TypedFML32  myDataBack = (TypedFML32)myRtn.getReplyBuffer();
 Integer myNewLong;
 Float myNewFloat;
 myView View;
 String myNewString;

try {
  myNewString = (String)myDataBack.Fget(MyFieldTable.FLD0, 0);
  myNewLong = (Integer)myDataBack.Fget(MyFieldTable.FLD1, 0);
  myNewFloat = (Float)myDataBack.Fget(MyFieldTable.FLD2, 0);
  vfld = (FviewFld)myDataBack.Fget(MyFieldTable.myview, 0);
  view = (myView)vfld.getViewData();
} catch (Ferror fe) {
  ….
}
```

The following code listing is an example FML Description(MyFieldTable) related to the example in Listing 8-2.

```
*base 20000

#name     number  type    flags   comments
FLD0      10      string  -       -
FLD1      20      long    -       -
FLD2      30      float   -       -
myview    50      view32  -       defined in View description file
```

Using the XmlViewCnv Class for XML to and From View/View(32) Translation

Use the `XmlViewCnv` class to perform XML to View /View(32) or View/View(32) to XML translation. The following code listing is an example that uses the `XmlViewCnv` class for conversion to and from XML buffer formats.

```
import examples.wtc.atmi.simpview.infoenc; // View class import
weblogic.wtc.gwt.XmlViewCnv;
import weblogic.wtc.jatmi.TypedBuffer;

public class xml2view
{
   public static void main(String[] args) {
      String xmlDoc =
      "<VIEW32><infoenc><amount>1000.0</amount><infoenc></VIEW32>";

      infoenc convertMe = new infoenc();
      convertMe = (infoenc) XmlViewCnv.XMLToView(
         xmlDoc,
         convertMe.getClass(),
         convertMe.getSubtype());

      convertMe = (infoenc) echo.Echo(convertMe);

      result = XmlViewCnv.ViewToXML(
         (TypedBuffer) convertMe,
         convertMe.getClass(),
         true);


      System.out.println(result);
   }
}
```

How to Create a Custom AppKey Plug-in

The following sections provide information about how to create custom AppKey generator plug-ins:

- How to Create a Custom Plug-In
- Example Custom Plug-in

How to Create a Custom Plug-In

Note: You cannot customize Tuxedo AAA tokens.

1. Create your custom Java plug-in using the AppKey and UserRec interfaces. You can provide any required initialization parameters or a property file using the `param` parameter of the `init` method.
2. Compile your plug-in. Example:

   ```
   javac exampleAppKey.java
   ```
3. Update your CLASSPATH to include the path to your compiled plug-in. Example:

   ```
   export CLASSPATH=$CLASSPATH:/home/mywork
   ```
4. Start your server.
5. Configure your WTC Service to use the Custom Plug-in. For more information, see the Custom Plug-in.

Example Custom Plug-in

The `exampleAppKey.java` file is an example of a custom plug-in. It utilizes a `tpusrfile` file as the database to store the AppKey.

Listing 9-1 exampleAppKey.Java Custom Plug-In

```
import java.io.*;
import java.lang.*;
import java.util.*;
import java.security.Principal;
import weblogic.wtc.jatmi.AppKey;
import weblogic.wtc.jatmi.UserRec;
import weblogic.wtc.jatmi.DefaultUserRec;
import weblogic.wtc.jatmi.TPException;
import weblogic.security.acl.internal.AuthenticatedSubject;
import weblogic.security.WLSPrincipals;

/**
 * @author Copyright (c) 2002 by BEA Systems, Inc. All Rights Reserved.
 */

/**
 * @exclude
 * Sample AppKey plug-in using TPUSRFILE as the database for APPKEY.
 * It is installed through "Custom" option.
 * The syntax for option custom plug parameter input contains the full
 * pathname to the <tpusrfile>
```

```
 *
 * @author BEA Systems, Inc.
 */
public class exampleAppKey implements AppKey {
  private String  anon_user = null;
  private String  tpusrfile = null;
  private File    myfile;
  private HashMap userMap;
  private long    l_time;
  private int     dfltAppkey;
  private boolean allowAnon;
  private final static int USRIDX = 0;
  private final static int PWDIDX = 1;
  private final static int UIDIDX = 2;
  private final static int GIDIDX = 3;
  private final static int CLTIDX = 4;

  private final static byte[] tpsysadm_string = {
    (byte)'t', (byte)'p', (byte)'s', (byte)'y', (byte)'s',
    (byte)'a', (byte)'d', (byte)'m' };
  private final static byte[] tpsysop_string = {
    (byte)'t', (byte)'p', (byte)'s', (byte)'y', (byte)'s', (byte)'o',
    (byte)'p' };

  public void init(String param, boolean anonAllowed, int dfltAppKey)
    throws TPException {
```

```
if (param == null) {
  System.out.println("Error: tpusrAppKey.init@param == null");
  throw new TPException(TPException.TPESYSTEM,
                        "Invalid input parameter");
}

// get the tpusrfile name
parseParam(param);

myfile  = new File(tpusrfile);
if (myfile.exists() != true) {
  System.out.println("Error: exampleAppKey.init@file \"" + param
                     + "\" does not exist");
  throw new TPException(TPException.TPESYSTEM,
                        "Failed to find TPUSR file");
}
if (myfile.isFile() != true) {
 System.out.println("Error: exampleAppKey.init@the specified name \"" +
                     param + "\" is not a file");
  throw new TPException(TPException.TPESYSTEM,
                        "Invalid TPUSR file");
}
if (myfile.canRead() != true) {
  System.out.println("Error: exampleAppKey.init@file \"" + param +
                     "\" is not readable");
  throw new TPException(TPException.TPESYSTEM,
                        "Bad TPUSR file permission");
```

```
    }

    userMap = new HashMap();

    // create the cache
    if (createCache(tpusrfile) == -1) {
      System.out.println("Error: exampleAppkey.init@fail to create user
cache");
      throw new TPException(TPException.TPESYSTEM,
                            "fail to create user cache");
    }

    l_time     = myfile.lastModified();
    anon_user  = weblogic.security.WLSPrincipals.getAnonymousUsername();
    allowAnon  = anonAllowed;
    dfltAppkey = dfltAppKey;

    System.out.println("exampleAppKey installed!");

    return;
  }

  public void uninit() throws TPException {
    if (userMap != null) {
      userMap.clear();
    }
    return;
  }
```

```
  public UserRec getTuxedoUserRecord(AuthenticatedSubject subj) {
    Object[] obj = subj.getPrincipals().toArray();
    if (obj == null || obj.length == 0) {
      // a subject without principals is an anonymous user
      if (allowAnon) {
        return new DefaultUserRec(anon_user, dfltAppkey);
      }
                System.out.println("Error: 
exampleAppKey.getTuxedoUserRecord@return " +
                         "anonymous user not allowed");
      return null;
    }

    // looping through all Principal names if necessary to get first user
    // name defined in tpuser file
    Principal user;
    String    username;
    int       key;
    UserRec   rec;

    for (int i = 0; i < obj.length; i++) {
      user     = (Principal)obj[i];
      username = user.getName();
      if (username.equals(anon_user)) {
        return new DefaultUserRec(anon_user, dfltAppkey);
      }
      if ((rec = (UserRec)userMap.get(username)) != null) {
```

```
        return rec;
      }
    }
    System.out.println("WARN: exampleAppKey.getTuxedoUserRecord@return " +
                       "null UserRec");
    return null;
  }

  private int createCache(String fname) {
    FileInputStream fin;
    byte[]          line;

    try {
      fin = new FileInputStream(fname);

      while ((line = readOneLine(fin)) != null) {
        DefaultUserRec newUser = parseOneLine(line);
        if (newUser != null) {
          userMap.put(newUser.getRemoteUserName(), newUser);
        }
      }
      fin.close();
    }
    catch (FileNotFoundException fnfe) {
      System.out.println("Error: exampleAppKey.createCache@reason: " +
fnfe);
      return -1;
    }
```

```
  catch (SecurityException se) {
   System.out.println("Error: exampleAppKey.createCache@reason: " + se);
    return -1;
  }
  catch (IOException ioe) {
   System.out.println("Error: exampleAppKey.createCache@reason: " + ioe);
    return -1;
  }
  catch (Exception e) {
    System.out.println("Error: exampleAppKey.createCache@reason: " + e);
    return -1;
  }
  return 0;
}

private byte[] readOneLine(FileInputStream fh) {
  int    len  = 80;
  byte[] line = new byte[len];
  int    inp  = -1;
  int    idx  = 0;

  try {
    while ((inp = fh.read()) != -1) {
      if (idx == 0 && (inp == '\n' || inp == '\0')) {
        continue;
      }
      if (inp == '\n') {
```

```
          break;
        }
        if (idx == (len - 1)) {
          byte[] tmp = new byte[len + 80];
          System.arraycopy(line, 0, tmp, 0, len);
          line = tmp;
          len += 80;
        }
        line[idx] = (byte)inp;
        idx++;
      }
    }
    catch (Exception e) {
      System.out.println("Error: exampleAppKey.readOneLine@reason: " + e);
      return null;
    }

    if (inp == -1 && idx == 0) {
      return null;
    }

    byte[] tmp = new byte[idx];
    System.arraycopy(line, 0, tmp, 0, idx);

    return tmp;
  }
```

```
private DefaultUserRec parseOneLine(byte[] line) {
  String      name;
  int         key   = 0;
  DefaultUserRec usr;
  int         firstCharacter;
  int         i;
  int         sidx;
  int         fldlen;
  int         fn;
  byte[]      buid  = null;
  byte[]      bgid  = null;
  byte[]      clt   = null;
  byte[]      uname = null;

  firstCharacter = (int)line[0];
  if (firstCharacter == '#' || firstCharacter == '\n' ||
      firstCharacter == '!' || firstCharacter == '\0' ||
      firstCharacter == '\r') {
    return null;
  }
  fldlen = 0;
  sidx   = 0;
  for (i = 0, fn = 0; i < line.length && fn <= CLTIDX; i++) {
    if (line[i] == (byte)':') {
      switch (fn) {
      case USRIDX:
        uname = new byte[fldlen];
```

```
  System.arraycopy(line, sidx, uname, 0, fldlen);
  break;
case UIDIDX:
  buid = new byte[fldlen];
  System.arraycopy(line, sidx, buid, 0, fldlen);
  break;
case GIDIDX:
  bgid = new byte[fldlen];
  System.arraycopy(line, sidx, bgid, 0, fldlen);
  break;
case CLTIDX:
                 if (line[sidx  ] == (byte)'T' &&
                          line[sidx+1] == (byte)'P' &&
                          line[sidx+2] == (byte)'C' &&
                          line[sidx+3] == (byte)'L' &&
                          line[sidx+4] == (byte)'T' &&
                          line[sidx+5] == (byte)'N' &&
                          line[sidx+6] == (byte)'M' &&
                          line[sidx+7] == (byte)',') {
                        sidx   += 8;
                        fldlen -= 8;
                 }
  if (fldlen > 0) {
                         clt = new byte[fldlen];
                     System.arraycopy(line, sidx, clt, 0, fldlen);
  }
  break;
```

```
    default:
      break;
    }   // end of switch
    fn++;
    fldlen = 0;
    sidx   = i + 1;
  }     // end of if
  else {
    fldlen++;
  }
}

// try to tolerate incomplete line
if (fn <= CLTIDX && fldlen > 0) {
             switch (fn) {
              case USRIDX:
                     uname = new byte[fldlen];
                     System.arraycopy(line, sidx, uname, 0, fldlen);
                     break;
              case UIDIDX:
                     buid = new byte[fldlen];
                     System.arraycopy(line, sidx, buid, 0, fldlen);
                     break;
              case GIDIDX:
                     bgid = new byte[fldlen];
                     System.arraycopy(line, sidx, bgid, 0, fldlen);
                     break;
```

```
             case CLTIDX:
                    if (line[sidx  ] == (byte)'T' &&
                             line[sidx+1] == (byte)'P' &&
                             line[sidx+2] == (byte)'C' &&
                             line[sidx+3] == (byte)'L' &&
                             line[sidx+4] == (byte)'T' &&
                             line[sidx+5] == (byte)'N' &&
                             line[sidx+6] == (byte)'M' &&
                             line[sidx+7] == (byte)',') {
                           sidx   += 8;
                           fldlen -= 8;
                    }
                    clt = new byte[fldlen];
                    System.arraycopy(line, sidx, clt, 0, fldlen);
                    break;
            }
}

if (uname == null || buid == null || bgid == null) {
  return null;
}

name = new String(uname);
if (clt != null) {
  if (Arrays.equals(tpsysadm_string, clt) == true) {
    key = TPSYSADM_KEY;
  }
```

```
      else if (Arrays.equals(tpsysop_string, clt) == true) {
        key = TPSYSOP_KEY;
      }
    }

    if (key == 0) {
      Integer u_val;
      Integer g_val;
      int uid = 0;
      int gid = 0;

      try {
        u_val = new Integer(new String(buid));
        g_val = new Integer(new String(bgid));
        uid   = u_val.intValue();
        gid   = g_val.intValue();
        uid &= UIDMASK;
        gid &= GIDMASK;
        key = uid | (gid << GIDSHIFT);
      }
      catch (NumberFormatException nfe) {
        System.out.println("Error: exampleAppKey.readOneLine@reason: " +
nfe);
        return null;
      }
    }

    return new DefaultUserRec(name, key);
```

```
  }

  private void parseParam(String param) {
    String str;

    // trim the input
    tpusrfile = param.trim();

    return;
  }
}
```

Application Error Management

The following sections provide mechanisms to manage and interpret error conditions in your applications:

- Testing for Application Errors
- WebLogic Tuxedo Connector Time-Out Conditions
- Guidelines for Tracking Application Events

Testing for Application Errors

Note: To view an example that demonstrates how to test for error conditions, see "Example Transaction Code" on page 5-6.

Your application logic should test for error conditions after the calls that have return values and take suitable steps based on those conditions. In the event that a function returned a value, you may invoke a functions that tests for specific values and performs the appropriate application logic for each condition.

Exception Classes

The WebLogic Tuxedo Connector throws the following exception classes:

- `Ferror`: Exception thrown for errors occurring while manipulating FML.
- `TPException`: Exception thrown that represents a TPException failure.

- `TPReplyException`: Exception thrown that represents a TPException failure when user data is associated with the exception thrown.

Fatal Transaction Errors

In managing transactions, it is important to understand which errors prove fatal to transactions. When these errors are encountered, transactions should be explicitly aborted on the application level by having the initiator of the transaction call `commit()`. Transactions fail for the following reasons:

- The initiator or participant of the transaction caused it to be marked for rollback.
- The transaction timed out.
- A `commit()` was called by a participant rather than by the originator of a transaction.

WebLogic Tuxedo Connector Time-Out Conditions

There are two types of time-out which can occur when using the WebLogic Tuxedo Connector:

- Blocking time-out.
- Transaction time-out.

Blocking vs. Transaction Time-out

Blocking time-out is exceeding the amount of time a call can wait for a blocking condition to clear up. Transaction time-out occurs when a transaction takes longer than the amount of timed defined for it in `setTransactionTimeout()`. By default, if a process is not in transaction mode, blocking time-outs are performed. When the *flags* parameter of a a communication call is set to `TPNOTIME`, it applies to blocking time-outs only. If a process is in transaction mode, blocking time-out and the `TPNOTIME` flag are not relevant. The process is sensitive to transaction time-out only as it has been defined for it when the transaction was started. The implications of the two different types of time-out follow:

- If a process is not in transaction mode and a blocking time-out occurs on an asynchronous call, the communication call that blocked will fail, but the call descriptor is still valid and may be used on a re-issue call. Further communication in general is unaffected.
- In the case of transaction time-out, the call descriptor to an asynchronous transaction reply (done without the `TPNOTRAN` flag) becomes stale and may no longer be referenced. The

only further communication allowed is the one case described earlier of no reply, no blocking, and no transaction.

Effect on commit()

The state of a transaction if time-out occurs after the call to `commit()` is undetermined. If the transaction timed out and the system knows that it was aborted, `setRollbackOnly()` or `rollback()` returns with an error.

If the state of the transaction is in doubt, you must query the resource to determine if any of the changes that were part of that transaction have been applied to it in order to discover whether the transaction committed or aborted.

Effect of TPNOTRAN

Note: A transaction can time-out while waiting for a reply that is due from a service that is not part of that transaction.

When a process is in transaction and makes a communications call with *flags* set to `TPNOTRAN`, it prohibits the called service from becoming a participant of that transaction. The success or failure of the service does not influence the outcome of that transaction.

Guidelines for Tracking Application Events

You can track the execution of your applications by using `System.out.println()` to write messages to the WebLogic Server trace log. Create a `log()` method that takes a variable of type `String` and use the variable name as the argument to the call, or include the message as a literal within quotation marks as the argument to the call. In the following example, a series of messages are used to track the progress of a `tpcall()`.

Listing 10-1 Example Event Logging

```
.
.
.
log(“About to call tpcall”);
        try {
                myRtn = myTux.tpcall("TOUPPER", myData, 0);
```

```
        }
        catch (TPReplyException tre) {
                log("tpcall threw TPReplyExcption " + tre);
                throw tre;
        }
        catch (TPException te) {
                log("tpcall threw TPException " + te);
                throw te;
        }
        catch (Exception ee) {
                log("tpcall threw exception: " + ee);
                throw new TPException(TPException.TPESYSTEM, "Exception: " +
ee);
        }
        log("tpcall successfull!");
.
.
.
private static void
log(String s)
{       System.out.println(s);}
.
.
.
```