Oracle® Tuxedo JCA Adapter

Programming Guide

12*c* Release 1 (12.1.1)

June 2012

ORACLE®

Oracle Tuxedo JCA Adapter Programming Guide, 12*c* Release 1 (12.1.1)

Copyright © 2010, 2012 Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

Oracle Tuxedo JCA Adapter Programming Guide

Oracle Tuxedo JCA Adapter Programming Guide

This chapter contains the following topics:

Overview

The Oracle Tuxedo JCA Adapter (Tuxedo JCA Adapter) supports both standard JEE Connector Architecture Common Client Interface (CCI) and Oracle Tuxedo Java Application-To-Monitor Interface (JATMI). Both interfaces allow client applications to access services located in remote Oracle Tuxedo application domains. The Tuxedo JCA Adapter supports transparent routing and load balancing internally. Requests are load-balanced and routed to different remote Oracle Tuxedo application domains that provide the same service.

Prerequisites

Developing an Tuxedo JCA Adapter application requires the following prerequisites:

- JDK 1.5, or later
- Java Application Server
  - Oracle WebLogic Server 10gR3 and later
  - IBM WebSphere 6.1 and later
  - JBoss Application Server 5.1.0 and later
  - WildFly 8.2.0.Final

  Note: For WildFly 8.2.0.Final, Oracle Tuxedo JCA Adapter Rolling Patch 006 or later should be installed. If you encounter any problems using Tuxedo JCA Adapter with WildFly 8.2.0.Final server, see Oracle Tuxedo JCA Adapter 12c Troubleshooting Guide for WildFly 8.2.0.Final.
- Text Editor, XML Editor, or an IDE
  - IBM WebSphere Application Server Toolkit

Common Development Tasks

Developing an application for the Tuxedo JCA Adapter requires the following steps:

1. Identify remote Oracle Tuxedo resources needed.
2. Configure the resource deployment descriptor.
3. Configure the Tuxedo JCA Adapter.
4. Create resource archive.
5. Deploy the Tuxedo JCA Adapter.

Note: These steps can be done independent of application development, but *must* be completed before running the client application. For more information, see the Oracle Tuxedo JCA Adapter Users Guide.

Using Connection Instance and Connection Factory

All client applications are required to lookup the Connection Factory for the Tuxedo JCA Adapter in the JNDI tree to retrieve a connection instance. The exact lookup string may differ depending on the configuration.

Different application servers may use different implementations to provide this information to the Oracle JCA Adapter. For example, Oracle WebLogic server uses the `weblogic-ra.xml` file (using the `<jndi-name>` XML tag) to provide this information as shown in Listing 1.

IBM WebSphere configures this information through the administration console using the "`JNDI name`" field of the "`J2C Connection Factory`" page.

Listing 1 Oracle WebLogic Server Connector

```
<weblogic-connector
  xmlns=http://www.bea.com/ns/weblogic/90>
  <jndi-name>eis/TuxedoConnector</jndi-name>
  ...
  <outbound-resource-adapter>
    <connection-definition-group>
<connection-factory-interface>javax.resource.cci.ConnectionFactory</connec
tion-factory-interface>
      <connection-instance>
         <jndi-name>eis/TuxedoConnectionFactory</jndi-name>
       </connection-instance>
     </connection-definition-group>
   <outbound-resource-adapter>
</weblogic-connector>
```

Listing 2 shows a Connection Factory lookup and Connection instance code example.

Listing 2 Connection Factory Lookup/Connection Instance Code Example

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

...

import javax.resource.cci.ConnectionFactory;
import javax.resource.cci.Connection;

...

Context                  ctx;
ConnectionFactory cf;
Connection              c;

...

ctx = new InitialContext();
cf   = ctx.lookup("eis/TuxedoConnectionFactory");
c    = cf.getConnection();

...
```

The `"ctx.lookup()"` call uses the string configured in `"jndi-name"`.

Using the DMConfigChecker Utility

The `DMConfigChecker` utility is used to encrypt configuration file passwords. It checks the Tuxedo JCA Adapter configuration file syntax and replaces all the unencrypted password elements with the encrypted password. If necessary, this utility can also generate a key file which is used to encrypt/decrypt the passwords.

For more information, see the Oracle Tuxedo JCA Adapter Command Reference.

Developing an Oracle Tuxedo JCA Adapter Client Application

For client applications, the Tuxedo JCA Adapter implements the necessary connection creation, session authentication, data privacy, data transformation, routing/load balancing, and transaction processes. This makes it easier, consistent, and transparent for an Tuxedo JCA Adapter client to access Oracle Tuxedo services.

Not all application servers run client programs in the same manner; they may have their own toolset and implementation methodology. In general, after you develop the client application, you must do the following steps to run client application server programs:

1. Build the client application.
2. Configure the client application.
3. Deploy the client application.
4. Run the client application.

Note: Refer to your target Application Server documentation for detailed information on how to build, configure, deploy, and run client applications.

This section contains the following topics:

CCI Client Programming

Client applications can access Oracle Tuxedo services using the JEE Connector Architecture Common Client Interface (CCI).

To develop a CCI-based Tuxedo JCA Adapter client application, you must do the following steps:

1. Create a new interaction and specification instance.
2. Set the imported Oracle Tuxedo service name and call.
3. Create and input a new Oracle Tuxedo Typed Buffer data record instance.
4. Execute service requests.
5. Release resources.
6. Retrieve data record output reply

The code examples in this section perform service calls to an Oracle Tuxedo service. The service name is "`TOUPPER`" and requires configuration. For more information, see Configuration File Examples. The "`TOUPPER`" service uses a STRING Typed Buffer for input and output.

To create a new interaction instance, the client application must place a call to the Connection. The interaction between client applications (`javax.resource.cci.Interaction`) and Oracle Tuxedo services must be customized using the interaction specification (`javax.resource.cci.InteractionSpec`) as shown in Listing 3.

Listing 3 Create a New Interaction Instance and Specification

```
Interaction                    ix;
TuxedoInteractionSpec ixspec;

...

ix        = c.reateInteraction()
ixspec = new TuxedoInteractionSpec();
```

You must import the following:

- the Oracle Tuxedo service name the client application wants to invoke (found in the Tuxedo JCA Adapter configuration file "`Import`" section)
- its input/output buffer type

The input/output buffer type must access the Oracle Tuxedo service code or query the Oracle Tuxedo Metadata Repository. For more information, see the Oracle Tuxedo Metadata Repository documentation.

Listing 4 shows a client application *synchronously* invoking the "TOUPPER" service.

Listing 4 CCI Client Application Invoking TOUPPER Service

```
ixspec.setFunctionName("TOUPPER");

ixspec.setInteractionVerb(InteractionSpec.SYNC_SEND_RECEIVE);
```

The input data sent must use an Oracle Tuxedo Typed Buffer.

Listing 5 shows the "TOUPPER" service using a STRING buffer type for input and output.

Listing 5 TOUPPER" Service Using a STRING Buffer Type

```
TuxedoStringRecord  inRec;
TuxedoStringRecord outRec;

...

inRec   = new TuxedoStringRecord();
outRec = new TuxedoStringRecord();
inRec.setRecordName("MyInputData");
outRec.setRecordName("MyOutputData");

inRec.setString(string_to_convert)
```

Listing 6 shows the actual "TOUPPER" service request, resource release, and reply data retrieval.

Listing 6 Service Request, Release Resources, and Output Data Retrieval

```
ix.execute(ixspec, inRec, outRec);

ix.close();
c.close();

String returned_data = outRec.getString();
```

To compile the Java code, the client application must import the following packages.

- `javax.resource.cci`
- `weblogic.wtc.jatmi`
- `com.oracle.tuxedo.adapter.cci`
- `com.oracle.tuxedo`

Listing 7 shows a CCI client application program example.

Listing 7 CCI Client Application Program Example

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.ejb.CreateException;

import javax.resource.cci.ConnectionFactory;
import javax.resource.cci.Connection;
import javax.resource.cci.Interaction;
import javax.resource.cci.Interactionspec;
import javax.resource.ResourceException;

import weblogic.wtc.jatmi.TPException;
import weblogic.wtc.jatmi.TPReplyException;

import com.oracle.tuxedo.adapter.TuxedoReplyException;
import com.oracle.tuxedo.adapter.cci.TuxedoStringRecord;
```

```
import com.oracle.tuxedo.adapter.cci.TuxedoInteractionSpec;

...

public String Toupper(String string_to_convert) throws TPException,
TuxedoReplyException
{
  Context                         ctx;
  ConnectionFactory      cf;
  Connection                    c;
  Interaction                     ix;
  TuxedoStringRecord    inRec;
  TuxedoStringRecord    outRec;
  TuxedoInteractionSpec ixspec;

  try {
    ctx = new InitialContext();
    cf   = ctx.lookup("eis/TuxedoConnectionFactory");
    c    = cf.getConnection();

    ix   = c.createInteraction();
    ixspec = new TuxedoInteractionSpec();
    ixspec.setFunctionName("TOUPPER");
    ixspec.setInteractionVerb(InteractionSpec.SYNC_SEND_RECEIVE);

    inRec   = new TuxedoStringRecord();
    outRec = new TuxdeoStringRecord();
    inRec.setRecordName("MyInputData");
    outRec.setRecordName("MyOutputData");

    outRec.setString(string_to_convert);
    ix.execute(ixspec, inRec, outRec);

    ix.close();
    c.close();
```

```
    String returned_data = outRec.getString();
    return returned_data;
  }
  catch (NamingException ne) {
    throw new TPException(TPException.TPESYSTEM,
                                  "Could not get TuxedoConnectionFactory");
  }
  catch (ResourceException re) {
    throw new TPException(TPException.TPESYSTEM,
                             "ResourceException occurred, reason: " + re);
  }
}
```

Transaction Client Programming

The Oracle JCA Adapter supports CCI-managed transaction client applications. The type of transaction depends largely on the transaction level (XA transactions or local transactions) configured in the Tuxedo JCA Adapter deployment descriptor. For more information, see the Oracle Tuxedo JCA Adapter Users Guide.

- CCI-Managed XA Client Programming
- CCI-Managed Local Transaction Programming

CCI-Managed XA Client Programming

To develop a VIEW buffer type-based CCI-managed XA client application, you must do the following steps:

1. Compile VIEW definition using viewj32 compiler.
2. Get user transaction, set transaction timeout, and start transaction.
3. Create new interaction and specification instance.
4. Set the imported Oracle Tuxedo service name.
5. Set the style of call.

6. Instantiate and initialize VIEW32 object.
7. Create new Oracle Tuxedo Typed Buffer data record instance.
8. Execute the service request.
9. Get the reply.
10. Commit the transaction.
11. Release the resources.
12. Retrieve the output data record reply.

The code examples in this section perform service calls to an Oracle Tuxedo service. The service name is `TOUPPER_V32` and requires configuration. For more information, see Configuration File Examples. The "`TOUPPER_32`" service requires a VIEW32 Typed Buffer for input and output.

Note: The equivalent of the VIEW32 Typed Buffer in the Tuxedo JCA Adapter is `TuxedoView32Record`. In the following examples, VIEW32 view is called "`View32`". The java code is generated using the `viewj32` compiler.

For more information, see Managing Typed Buffers in *Programming An Oracle Tuxedo ATMI Application Using C* and the Oracle Tuxedo JCA Adapter Command Reference Guide for `viewj` and `viewj32` information.

Listing 8 shows a VIEW32 definition file example.

Listing 8 VIEW32 Definition File Example

```
VIEW View32
short TEST_SHORT       -       1     -       -       0
string TEST_STRING     -       1     -       100     -
```

Compile using the following `viewj32` utility command:

```
java -classpath %CLASSPATH% weblogic.wtc.jatmi.viewj32 tuxedo.test.simapp
View32
```

In the above example, this command creates a "`View32.java`" Java file (package name "`tuxedo.test.simpapp`"),in the current working directory. If the VIEW32 file contains a nested view32 structure, a corresponding Java file is generated for each nested view.

Listing 9 shows how to create and start a user transaction. The transaction times out after 300 seconds.

Listing 9 Create and Start a User Transaction

```
UserTransaction utx;

...

utx = (UserTransaction)ctx.lookup("java:comp/UserTransaction");
utx.setTransactionTimeout(300);
utx.begin();
```

To create new interaction instance, the client application must place a call to the Connection. The interaction between client applications (`javax.resource.cci.Interaction`), and Oracle Tuxedo services must be customized using the interaction specification (`javax.resource.cci.InteractionSpec`) as shown in Listing 10.

Listing 10 Create a New Interaction Instance and Specification

```
Interaction                    ix;
TuxedoInteractionSpec ixspec;

...

ix        = c.reateInteraction()
ixspec = new TuxedoInteractionSpec();
```

You must import the following:

- the Oracle Tuxedo service name the client application wants to invoke (found in the Tuxedo JCA Adapter configuration file "Import" section)
- its input/output buffer type

The input/output buffer type must access the Oracle Tuxedo service code or query the Oracle Tuxedo Meta Data repository. Repository. For more information, see the Oracle Tuxedo Metadata Repository documentation.

Listing 11 shows a client application invoking the "TOUPPER_32" service using asynchronous interaction.

Listing 11 CCI Transaction Client Application Invoking TOUPPER Service

```
ixspec.setFunctionName("TOUPPER_V32");
ixspec.setInteractionVerb(InteractionSpec.SYNC_SEND);
```

The input data sent to Oracle Tuxedo must use an Oracle Tuxedo Typed Buffer.

Listing 12 shows the "TOUPPER_32" service using a VIEW32 Typed Buffer for input and output.

Listing 12 TOUPPER_32 Service Using a VIEW32 Buffer Type

```
View32                          myData;
TuxedoView32Record  inRec;

...

myData = new View32();
myData.setTEST_SHORT((short)4);
myData.setTEST_STRING(string_to_convert);

inRec   = new TuxedoView32Record(myData);
inRec.setRecordName("MyInputData");
```

Listing 13 shows the actual "TOUPPER_32" service request, resource release, and reply data retrieval.

Listing 13 Service Request, Release Resources, and Output Data Retrieval

```
TuxedoView32Record outRec;

...

ix.execute(ixspec, inRec);
ixspec.setInteractionVerb(InteractionSpec.SYNC_RECEIVE);
outRec = (TuxedoView32Record)ix.execute(ixspec, inRec);

utx.commit();
ix.close();
c.close();

View32 myDataBack = (View32)outRec.getView32();
String returned_data = myDataBack.getTEST_STRING();
```

To compile the Java code, the client application must import the following packages:

- javax.resource.cci
- weblogic.wtc.jatmi
- com.oracle.tuxedo.adapter.cci
- com.oracle.tuxedo

Listing 14 shows a transaction client application program example.

Listing 14 Transaction Client Application Program Example

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.ejb.CreateException;

import javax.resource.cci.ConnectionFactory;
import javax.resource.cci.Connection;
import javax.resource.cci.Interaction;
```

```
import javax.resource.cci.Interactionspec;
import javax.resource.ResourceException;

import weblogic.wtc.jatmi.TPException;
import weblogic.wtc.jatmi.TPReplyException;

import com.oracle.tuxedo.adapter.TuxedoReplyException;
import com.oracle.tuxedo.adapter.cci.TuxedoView32Record;
import com.oracle.tuxedo.adapter.cci.TuxedoInteractionSpec;

import tuxedo.test.simpapp.View32;

...

private void cleanup(UserTransaction utx, Interaction ix, Connection c)
{
  try {
     if (utx != null) utx.rollback();
     if (ix != null) ix.close();
     if (c != null) c.close();
  }
  catch (Exception e) {
    /* ignore */
  }
}

public String Toupper(String string_to_convert) throws TPException,
TuxedoReplyException
{
  Context                         ctx;
  ConnectionFactory      cf;
  Connection                    c;
  UserTransaction           utx;
  View32                          myData;
  View32                          myDataBack;
  Interaction                     ix;
  TuxedoView32Record    inRec;
  TuxedoView32Record    outRec;
```

```
InteractionSpec               ixspec;

try {
  ctx = new InitialContext();
  cf   = ctx.lookup("eis/TuxedoConnectionFactory");
  c    = cf.getConnection();
  utx = (UserTransaction)ctx.lookup("java:comp/UserTransaction");
  utx.setTransactionTimeout(300);
  utx.begin();

  ix   = c.createInteraction();
  ixspec = new TuxedoInteractionSpec();
  ixspec.setFunctionName("TOUPPER_V32");
  ixspec.setInteractionVerb(InteractionSpec.SYNC_SEND);

  myData = new View32();
  myData.setTEST_SHORT((short)4);
  myData.setTEST_STRING(string_to_convert);

  inRec   = new TuxedoView32Record(myData);
  inRec.setRecordName("MyInputData");

  ix.execute(ixspec, inRec);
  ixspec.setInteractionVerb(InteractionSpec.SYNC_RECEIVE);
  outRec = (TuxedoView32Record)ix.execute(ixspec, inRec);

  utx.commit();
  ix.close();
  c.close();

  myDataBack = (View32)outRec.getView32();
  String returned_data = myDataBack.getTEST_STRING();
  return returned_data;
}
catch (NamingException ne) {
  cleanup(utx, ix, c);
  throw new TPException(TPException.TPESYSTEM,
                                "Could not get TuxedoConnectionFactory");
```

```
  }
  catch (ResourceException re) {
    cleanup(utx, ix, c);
    throw new TPException(TPException.TPESYSTEM,
                                          "ResourceException occurred,
reason: " + re);
  }
  catch (javax.transaction.RollbackException rbe) {
    cleanup(utx, ix, c);
    throw new TPException(TPException.TPETRAN, "Exception: " + rbe);
  }
  catch (javax.transaction.NotSuppotedException hre) {
    cleanup(utx, ix, c);
    throw new TPException(TPException.TPETRAN, "Exception: " + nse);
  }
  catch (javax.transaction.HeuristicRollbackException hre) {
    cleanup(utx, ix, c);
    throw new TPException(TPException.TPETRAN, "Exception: " + hre);
  }
  catch (javax.transaction.HeuristicMixException hme) {
    cleanup(utx, ix, c);
    throw new TPException(TPException.TPETRAN, "Exception: " + hme);
  }
  catch (javax.transaction.SystemException se) {
    cleanup(utx, ix, c);
    throw new TPException(TPException.TPETRAN, "Exception: " + se);
  }
}
```

CCI-Managed Local Transaction Programming

The Tuxedo JCA Adapter supports local managed transaction client applications using CCI. The transaction requires an Tuxedo JCA Adapter specific extension in order to set per transaction timeouts.

To develop a *synchronous* CCI-based Tuxedo JCA Adapter local managed transaction client program using a VIEW32 Typed Buffer, you must do the following steps:

1. Create a new Local Transaction instance.
2. Create a new interaction and specification instance.
3. Set the imported Oracle Tuxedo service name.
4. Set the call style.
5. Start Local Transaction.
6. Create a new Oracle Tuxedo Typed Buffer data record instance.
7. Send the input data to the data record.
8. Execute the service request.
9. Commit Local Transaction.
10. Release the resources.
11. Retrieve output data record reply.

The code examples in this section perform service calls to an Oracle Tuxedo service. The service name is `TOUPPER` and requires configuration. For more information, see Configuration File Examples. The "`TOUPPER`" service requires a STRING Typed Buffer for input and output.

Listing 15 shows how to create a new Tuxedo JCA Adapter Local Transaction instance from the Oracle Tuxedo Connection (`com.oracle.tuxedo.adapter.cci.TuxedoJCALocalTransaction`).

Listing 15 Create a New Local Transaction Instance

```
TuxedoJCALocalTransaction ltx;

...

ltx = (TuxedoJCALocalTransaction)c.getLocalTransaction();
```

To create new interaction instance, the client application must place a call to the Connection. The interaction between client applications (`javax.resource.cci.Interaction`) and Oracle

Tuxedo services must be customized using the interaction specification (`javax.resource.cci.InteractionSpec`) as shown in Listing 16.

Listing 16 Create a New Interaction and Specification

```
Interaction                    ix;
TuxedoInteractionSpec ixspec;

...

ix        = c.reateInteraction()
ixspec = new TuxedoInteractionSpec();
```

Listing 17 shows how to create and start a managed local transaction. The transaction times out after 15 seconds.

Note: This is an Tuxedo JCA Adapter specific implementation and is not part of the standard CCI Local Transaction interface.

Listing 17 Create and Start a Local Transaction

```
ltx.begin(15);
```

You must import the following:

- the Oracle Tuxedo service name the client application wants to invoke (found in the Tuxedo JCA Adapter configuration file "Import" section).
- its input/output buffer type.

  The input/output buffer type must access the Oracle Tuxedo service code or query the Oracle Tuxedo Meta Data repository. Repository. For more information, see the Oracle Tuxedo Metadata Repository documentation.

Listing 18 shows the client application *synchronously* using the "`TOUPPER`" service.

Listing 18 Local Transaction Client Application Invoking TOUPPER Service

```
ixspec.setFunctionName("TOUPPER");
ixspec.setInteractionVerb(InteractionSpec.SYNC_SEND_RECEIVE);
```

The input data sent to Oracle Tuxedo must use an Oracle Tuxedo Typed Buffer.

Listing 19 shows the "TOUPPER" service using a STRING Typed Buffer for input and output.

Listing 19 TOUPPER Service Using a STRING Typed Buffer

```
TuxedoStringRecord  inRec;
TuxedoStringRecord  outRec;

...

inRec   = new TuxedoStringRecord();
outRec = new TuxedoStringRecord();
inRec.setRecordName("MyInputData");
outRec.setRecordName("MyOutputData");
inRec.setString(string_to_convert);
```

Listing 20 shows the actual "TOUPPER" service request, resource release, and reply data retrieval.

Listing 20 Service Request, Release Resources, and Output Data Retrieval

```
ix.execute(ixspec, inRec, outRec);

if (outRec.getTperrno() == 0) {
  ltx.commit();
}
else {
  ltx.rollback();
}
```

```
ltx = null;

ix.close();
c.close();

String returned_data = outRec.getString();
```

To successfully compile the Java code, the client application must import the following packages.

- `javax.resource.cci`
- `weblogic.wtc.jatmi`
- `com.oracle.tuxedo.adapter.cci`
- `com.oracle.tuxedo`

Listing 21 shows a local transaction client application program example.

Listing 21 Local Transaction Client Application Program Example

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.ejb.CreateException;

import javax.resource.cci.ConnectionFactory;
import javax.resource.cci.Connection;
import javax.resource.cci.Interaction;
import javax.resource.cci.Interactionspec;
import javax.resource.ResourceException;

import weblogic.wtc.jatmi.TPException;
import weblogic.wtc.jatmi.TPReplyException;

import com.oracle.tuxedo.adapter.TuxedoReplyException;
import com.oracle.tuxedo.adapter.cci.TuxedoView32Record;
import com.oracle.tuxedo.adapter.cci.TuxedoInteractionSpec;
import com.oracle.tuxedo.adapter.cci.TuxedoJCALocalTransaction;
```

```
...

public String Toupper(String string_to_convert) throws TPException,
TuxedoReplyException
{
  Context                         ctx;
  ConnectionFactory      cf;
  Connection                    c = null;
  TuxedoJCALocalTransaction ltx = null;
  Interaction                     ix = null;
  TuxedoStringRecord    inRec;
  TuxedoStringRecord    outRec;
  InteractionSpec             ixspec;

  try {
    ctx = new InitialContext();
    cf   = ctx.lookup("eis/TuxedoConnectionFactory");
    c    = cf.getConnection();
    ltx = (TuxedoJCALocalTransaction)c.getLocalTransaction();

    ix   = c.createInteraction();
    ixspec = new TuxedoInteractionSpec();
    ixspec.setFunctionName("TOUPPER");
    ixspec.setInteractionVerb(InteractionSpec.SYNC_SEND_RECEIVE);
    ltx.begin(15);

    inRec   = new TuxedoStringRecord();
    outRec = new TuxedoStringRecord();
    inRec.setRecordName("MyInputData");
    outRec.setRecordName("MyOutputData");
    inRec.setString(string_to_convert);

    ix.execute(ixspec, inRec, outRec);

    if (outRec.getTperrno() == 0) {
      ltx.commit();
    }
```

```
    else {
      ltx.rollback();
    }
    ix.close();
    c.close();

    String returned_data = outRec.getString();
    return returned_data;
  }
  catch (NamingException ne) {
    throw new TPException(TPException.TPESYSTEM,
                                    "Could not get
TuxedoConnectionFactory");
  }
  catch (ResourceException re) {
    if (ltx != null) {
      try {
        ltx.rollback();
      }
      catch (ResourceException xre) {
        /* ignore it */
      }
    }
      try {
       if (ix != null) ix.close();
       if (c != null) c.close();
    }
    catch (Exception e) {
       /* ignore it */
    }
    throw new TPException(TPException.TPESYSTEM,
                                          "ResourceException occurred,
reason: " + re);
  }
}
```

JATMI Client Programming

Client applications can access an Oracle Tuxedo service using the Java Application To Monitor Interface (JATMI). JATMI is a straight Java implementation of the Oracle Tuxedo ATMI interface.

To develop a JATMI-based Tuxedo JCA Adapter client application, you must do the following steps:

1. Create a new interaction instance.
2. Create and input a new JATMI Typed Buffer data record instance.
3. Call Oracle Tuxedo service.
4. Retrieve output data record reply.
5. Release resources.

The code examples in this section perform service calls to an Oracle Tuxedo service. The service name is `TOUPPER` and requires configuration. For more information, see Configuration File Examples. The Oracle Tuxedo `TOUPPER` service requires a STRING Typed Buffer for input and output.

To create new interaction instance, the client application must place a call to the Connection. When you use the JATMI interaction extension (`com.oracle.tuxedo.adapter.cci.TuxedoInteractionSpec`), an interaction specification is not required to customize the interaction between client applications and Oracle Tuxedo services.The JATMI service invocation interface already includes these interaction specifications as shown in Listing 22.

Listing 22 New JATMI Interaction Instance

```
Interaction                    ix;

...

ix        = c.reateInteraction()
```

The input data must be transported using an Oracle Tuxedo Typed Buffer. Listing 23 shows the "TOUPPER" service using a STRING Typed Buffer for input and output.

Listing 23 TOUPPER" Service Using a STRING Buffer Type

```
TypedString inData;

...

inData  = new TypedString(string_to_convert);
```

Listing 24 shows the actual "TOUPPER" service request and data retrieval reply.

Listing 24 JATMI Client Application TOUPPER Service Request and Output Data Retrieval

```
Reply           myRtn;
TypedString outData;

myRtn = ix.tpcall("TOUPPER", inData, 0);
outData= (TypedString)myRtn.getReplyBuffer();
String returned_data = outData.toString()
```

Listing 25 shows how the resources are released.

Listing 25 JATMI Client Application Resource Release

```
ix.tpterm();
c.close();
```

To compile the Java code, the client application must import the following packages.

- javax.resource.cci

- weblogic.wtc.jatmi
- com.oracle.tuxedo.adapter.cci
- com.oracle.tuxedo

Listing 26 shows a JATMI client application program example.

Listing 26 JATMI Client Application Program Example

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.ejb.CreateException;

import javax.resource.cci.ConnectionFactory;
import javax.resource.cci.Connection;
import javax.resource.cci.Interaction;
import javax.resource.cci.Interactionspec;
import javax.resource.ResourceException;

import weblogic.wtc.jatmi.TPException;
import weblogic.wtc.jatmi.TPReplyException;
import weblogic.wtc.jatmi.Reply;
import weblogic.wtc.jatmi.TypedString;

import com.oracle.tuxedo.adapter.TuxedoReplyException;
import com.oracle.tuxedo.adapter.cci.TuxedoInteraction;

...

public String Toupper(String string_to_convert) throws TPException,
TPReplyException
{
Context                         ctx;
ConnectionFactory      cf;
Connection                    c;
Interaction                     ix;
TypedString                 inData;
```

```
TypedString                 outData;
Reply                           myRtn;

try {
    ctx = new InitialContext();
    cf   = ctx.lookup("eis/TuxedoConnectionFactory");
    c    = cf.getConnection();

    ix   = c.createInteraction();
    inData = new TypedString(string_to_convert);
    myRtn = ix.tpcall("TOUPPER", inData, 0);
    outData= (TypedString)myRtn.getReplyBuffer();

    String returned_data = outData.toString();

    ix.tpterm();
    c.close()
    return returned_data;
  }
  catch (NamingException ne) {
    throw new TPException(TPException.TPESYSTEM,
                                    "Could not get
TuxedoConnectionFactory");
  }
  catch (ResourceException re) {
    throw new TPException(TPException.TPESYSTEM,
                                            "ResourceException occurred,
reason: " + re);
  }
}
```

Inbound EJB Service Programming

You can use the Tuxedo JCA Adapter to access EJB-based Oracle Tuxedo client services. In order for the Tuxedo JCA Adapter to invoke an EJB, the EJB must use the

`weblogic.wtc.jatmi.TuxedoService` interface. This interface defines a single method called service as shown in Listing 27.

Listing 27 EJB Service Single Method

```
public Reply service(TPServiceInformation svcinfo)
       throws TPException, TPReplyException, RemoteException;
```

To develop an EJB-based service application using the `TuxedoService.service()` interface, you must do the following steps:

1. Retrieve input data and perform task.
2. Create Typed Buffer for output data.
3. Setup the output data to be returned to caller.
4. Configure the EJB deployment descriptor.

The code examples in this section show how to:

- use the `TuxedoService` interface.
- configure the EJB in the Tuxedo JCA Adapter configuration file to expose the service.
- configure the Oracle Tuxedo GWTDOMAIN gateway to import the service.

The service name is `TOLOWER` and requires configuration. For more information, see Configuration File Examples. The EJB service uses a STRING Typed Buffer for input and output.

Listing 28 shows an example of how input data is retrieved using `TPServiceInformation` (shown in Listing 27).

Listing 28 EJB Input Data Retrieved from TPServiceInformation

```
TypedString data;

...
```

```
data = (TypedString)mydata.getServiceData();
```

The input data is converted to lower case as shown in Listing 29

Listing 29 EJB Input Data Converted to Lower Case

```
String lowered;

...

lowered = data.toString().toLowerCase();
```

When the output data is available, it must be wrapped in an Oracle Tuxedo Typed Buffer. Listing 30 shows the how the output data is wrapped using the TypedString Typed Buffer.

Listing 30 EJB Output Data Wrapped in TypedString Typed Buffer

```
TypedString return_data;

...

return_data = new TypedString(lowered);
```

The output Typed Buffer is then transported back to the caller (using the `TPServiceInformation` object) as shown in Listing 31.

Listing 31 EJB Output Typed Buffer Transported to Caller

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

In order for the Tuxedo JCA Adapter to successfully invoke the EJB service, the EJB deployment descriptor must be configured using the following information:

- `home interface: weblogic.wtc.jatmi.TuxedoServiceHome`
- `remote interface: weblogic.wtc.jatmi.TuxedoService`
- `jndi name: tuxedo.services.TolowerHome`

The required prefix (`tuxedo.services`) and the EJB name (`TolowerHome`) are configured in the `<EXPORT> <SOURCE>` element of the Tuxedo JCA Adapter configuration file as shown in Listing 32.

Listing 32 EJB "TOLOWER"Configuration

```
<Export name="TOLOWER">
  <RemoteName>TOLOWER</RemoteName>
  <SessionName>session_1</SessionName>
  <Type>EJB</Type>
  <Source>tuxedo.services.TolowerHome</Source>
</Export>
```

To successfully compile the Java code, the client application must import the following packages:

- `weblogic.wtc.jatmi`
- `com.oracle.tuxedo.adapter.tdom`

Listing 33 shows an EJB client application program example.

Listing 33 EJB Client Application Program Example

```
package test.tuxedo.simpserv;

import javax.ejb.CreateException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.naming.Context;
import javax.naming.InitialContext;
```

```
import javax.naming.NamingException;

import weblogic.wtc.jatmi.TPException;
import weblogic.wtc.jatmi.TypedString;
import weblogic.wtc.jatmi.Reply;

import com.oracle.tuxedo.adapter.tdom.TPServiceInformation;

...

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

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

  mydata.setReplyBuffer(return_data);

  return mydata;
}

...
```

Inbound POJO Service Programming

You can use the Tuxedo JCA Adapter to access Plain Old Java Object (POJO)-based Oracle Tuxedo client services.

In order for the Tuxedo JCA Adapter to invoke a POJO service, the POJO service must provide a method with same name as the exported name. This method must take two fixed arguments: `TPServiceInformation` and `TPRequestAsyncReply`.

To develop an POJO-based service application using the `TuxedoService.service()` interface, you must do the following steps:

1. Retrieve input data and perform a task.
2. Create Typed Buffer for output data.
3. Setup the output data to be returned to caller.
4. Inform Tuxedo JCA Adapter POJO handler of success or failure
5. Configure the POJO deployment descriptor.

The code examples in this section show how to:

- use an ordinary Java class to provide a service to an Oracle Tuxedo C/C++ client.
- configure the POJO in the Tuxedo JCA Adapter configuration file to expose the service
- configure an Oracle Tuxedo GWTDOMAIN gateway to import the service

The service name is `MYTOLOWER` and requires configuration. For more information, see Configuration File Examples. The POJO service requires a STRING Typed Buffer for input and output.

Listing 34 shows an example of how input data is retrieved using `TPServiceInformation` (shown in Listing 27).

Listing 34 POJO Input Data Retrieved from Input TPServiceInformation

```
TypedString typedstr;

...

data = (TypedString)svcinfo.getServiceData();
```

The input data is converted to lower case as shown in Listing 35

Listing 35 POJO Input Data Converted into Lower Case

```
String lower;

...

lower = typedstr.toString().toLowerCase();
```

When the output data is available, it must be wrapped in an Oracle Tuxedo Typed Buffer. Listing 36 shows the how the output data is wrapped using the TypedString Typed Buffer.

Listing 36 POJO Output Data Wrapped in TypedString Buffer

```
TypedString return_data;

...

return_data = new TypedString(lower);
```

The output Typed Buffer is then transported back to the caller (using the `TPServiceInformation` object) as shown in Listing 31.

Listing 37 POJO Output Typed Buffer Transported to Caller

```
mydata.setReplyBuffer(return_data);
areply.success(svcinfo);
```

In order for the Tuxedo JCA Adapter to successfully invoke a POJO service, the POJO deployment descriptor must be configured. The POJO method name must be configured in the `<EXPORT>` section of the Tuxedo JCA Adapter configuration file as shown in Listing 38.

The <SOURCE> element contains the fully qualified class name, and the <SourceLocation> element contains the full path name to the .JAR file that contains the class. The .JAR file must be configured in the CLASSPATH.

Listing 38 POJO "TOLOWER_POJO" Configuration

```
<Export name="TOLOWER_POJO">
  <RemoteName>TOLOWER_POJO</RemoteName>
  <SessionName>session_1</SessionName>
  <Type>POJO</Type>
  <Source>com.oracle.tuxedo.test.MyTolower</Source>
  <SourceLocation>c:\tuxedo\jca\test\myapp.jar</SourceLocation>
</Export>
```

To successfully compile the Java code, the client application must import the following packages.

- weblogic.wtc.jatmi
- com.oracle.tuxedo.adapter.tdom

Listing 39 shows a POJO client application program example.

Listing 39 POJO Client Application Program Example

```
package com.oracle.tuxedo.test;

import javax.ejb.CreateException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import weblogic.wtc.jatmi.TPException;
import weblogic.wtc.jatmi.TypedString;
import weblogic.wtc.jatmi.Reply;
import weblogic.wtc.jatmi.TPRequestAsyncReply;
```

```
import com.oracle.tuxedo.adapter.tdom.TPServiceInformation;

...

public void TOLOWER_POJO(TPServiceInformation svcinfo, TPRequestAsyncReply
areply) throws TPException
{
  TypedString   typedstr;
  String              lower;
  TypedString    returned_data;

  typedstr = (TypedString)svcinfo.getServiceData();
  lower = typedstr.toString().toLowerCase();
  returned_data = new TypedString(lower);

  svcinfo.setReplyBuffer(return_data);
  areply.success(svcinfo);
}

...
```

AUTOTRAN Programming

The Tuxedo JCA Adapter supports adapter managed AUTOTRAN transactions. For AUTOTRAN transactions, the Tuxedo JCA Adapter will start a transaction if the service request is not already part of an existing transaction. The Tuxedo JCA Adapter will commit or rollback the transaction before returning the reply or exception to client program. Whether the Tuxedo JCA Adapter commits or rolls back transactions depends on the reply and session connection status. If the reply indicates a failure or the session connection error, then it rolls back the transaction, otherwise it commits the transaction.

To develop a synchronous CCI-based Tuxedo JCA Adapter AUTOTRAN client program, the procedure and program is exactly the same as developing an ordinary synchronous CCI-based client without transactions as described in CCI Client Programming. The only differences is configuration.

Listing 40 shows an example that enables AUTOTRAN on imported resource TOUPPER with transaction timeout set to 15 seconds.

Listing 40 AUTOTRAN Configuration Using DMCONFIG File

```
....
<Import name="TOUPPER" autotran=true trantime=15>
  <RemoteName>TOUPPER</RemoteName>
  <SessionName>session_1</SessionName>
  <LoadBalancing>RoundRobin</LoadBalancing>
</Import>
....
```

Beside using the Tuxedo JCA Adapter Configuration file to configure the AUTOTRAN for an individual imported resource, you can also enable the adapter-wise AUTOTRAN using property in the Resource Adapter Deployment Descriptor.

Listing 41 shows an example that enables AUTOTRAN on every imported resource with transaction timeout set to 15 seconds.

Listing 41 Enable AUTOTRAN Using Property in ra.xml

```
...
<config-property>
  <config-property-name>appManagedLocalTxTimeout</config-property-name>
  <config-property-type>java.lang.Integer</config-property-type>
  <config-property-value>15</config-property-value>
</config-property>
<config-property>
   <config-property-name>autoTran</config-property-name>
   <config-property-type>java.lang.Boolean</config-property-type>
   <config-property-value>true</config-property-value>
</config-property>
...
```

Configuration File Examples

To run the Tuxedo JCA Adapter, you must configure the following files:

- Tuxedo JCA Adapter configuration file
- Oracle Tuxedo UBBCONFIG and DMBCONFIG files

Oracle Tuxedo JCA Adapter Configuration Examples

The Tuxedo JCA Adapter configuration file is a formal-syntax XML file. The location of this file is configured in the ra.xml file in the resource adapter configuration property. For more information, see the Oracle JCA Users Guide and the Oracle Tuxedo JCA Adapter Reference Guide.

Listing 42 shows an example ra.xml file snippet that links the configuration file with the Tuxedo JCA Adapter.

Listing 42 ra.xml File Example

```
<config-property>
  <config-property-name>dmconfig</config-property-name>
  <config-property-type>java.lang.String</config-property-type>
<config-property-value>c:/myJcaApp/adapter/bdmconfig.xml</config-property-
value>
</config-property>
```

Listing 43 shows an Tuxedo JCA Adapter DMCONFIG file example that accesses:

- A single Oracle Tuxedo application domain.
- Oracle Tuxedo TOUPPER_V32 and TOUPPER services.

  Note: TOUPPER_V32 is a VIEW32 Typed Buffer service. TOUPPER is a STRING Typed Buffer service.

The Local Access Point defines the Tuxedo JCA Adapter listening end point. The Remote Access Point defines the Oracle Tuxedo GWTDOMAIN gateway listening end point.

Listing 43 Tuxedo JCA Adapter Configuration File

```
<?xml version="1.0" encoding="UTF-8"?><TuxedoConnector>
  <Resources>
    <ViewFile32Classes>tuxedo.test.simpapp.View32</ViewFile32Classes>
  </Resources>
  <LocalAccessPoint name="JDOM">
    <AccessPointId>JDOM_ID</AccessPointId>
    <NetworkAddress>//localhost:10801</NetworkAddress>
  </LocalAccessPoint>
  <RemoteAccessPoint name="TDOM1">
    <AccessPointId>TDOM1_ID</AccessPointId>
    <NetworkAddress>//localhost:12478</NetworkAddress>
  </RemoteAccessPoint>
  <SessionProfile name="profile_1">
    <Security>NONE</Security>
    <BlockTime>30000</BlockTime>
    <Interoperate>false</Interoperate>
    <ConnectionPolicy>ON_STARTUP</ConnectionPolicy>
    <ACLPolicy>local</ACLPolicy>
    <CredentialPolicy>local</CredentialPolicy>
    <RetryInterval>60</RetryInterval>
    <MaxRetries>1000</MaxRetries>
    <CompressionLimit>1000000</CompressionLimit>
  </SessionProfile>
  <Session name="session_1">
    <LocalAccessPointName>JDOM</LocalAccessPointName>
    <RemoteAccessPointName>TDOM1</RemoteAccessPointName>
    <ProfileName>profile_1</ProfileName>
  </Session>
  <Import name="TOUPPER">
    <RemoteName>TOUPPER</RemoteName>
    <SessionName>session_1</SessionName>
    <LoadBalancing>RoundRobin</LoadBalancing>
  </Import>
  <Import name="TOUPPER_V32">
    <RemoteName>TOUPPER_V32</RemoteName>
    <SessionName>session_1</SessionName>
```

```
    <LoadBalancing>RoundRobin</LoadBalancing>
  </Import>
<Export name="TOLOWER">
    <RemoteName>TOLOWER</RemoteName>
    <SessionName>session_1</SessionName>
    <Type>EJB</Type>
    <Source>tuxedo.services.TolowerHome</Source>
  </Export>
  <Export name="TOLOWER_POJO">
    <RemoteName>TOLOWER_POJO</RemoteName>
    <SessionName>session_1</SessionName>
    <Type>POJO</Type>
    <Source>com.oracle.tuxedo.test.MyTolower</Source>
    <SourceLocation>c:\tuxedo\jca\test\MyApp.jar</SourceLocation>
  </Export>
</TuxedoConnector>
```

JCA Adapter AUTOTRAN Configuration

Listing 44 provides an AUTOTRAN configuration example.

Listing 44 AUTOTRAN Configuration Example

```
<?xml version="1.0" encoding="UTF-8"?>
<connector xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/connector_1_5.xsd"
    version="1.5">
  <display-name>Tuxedo JCA Adapter</display-name>
  <vendor-name>Oracle</vendor-name>
  <eis-type>Tuxedo</eis-type>
  <resourceadapter-version>12c(12.1.1)</resourceadapter-version>
  <license>
      <description>Tuxedo SALT license</description>
      <license-required>false</license-required>
```

```
  </license>
  <resourceadapter>
     <resourceadapter-class>com.oracle.tuxedo.adapter.
       TuxedoClientSideResourceAdapter</resourceadapter-class>
     <config-property>
        <config-property-name>xaAffinity</config-property-name>
        <config-property-type>java.lang.String</config-property-type>
        <config-property-value>true</config-property-value>
     </config-property>
     <config-property>
         <config-property-name>appManagedLocalTxTimeout
            </config-property-name>
         <config-property-type>java.lang.Integer</config-property-type>
          <config-property-value>30</config-property-value>
     </config-property>
     <config-property>
         <config-property-name>autoTran</config-property-name>
         <config-property-type>java.lang.Boolean</config-property-type>
         <config-property-value>true</config-property-value>
    </config-property>
<!--
Uncomment this if you are running Tuxedo version before 11.1.1.2.0

     <config-property>
         <config-property-name>localAccessPointSpec</config-property-name>
         <config-property-type>java.lang.String</config-property-type>
         <config-property-value>//localhost:10801/domainId=JDOM_ID
           </config-property-value>
     </config-property>

-->
     <config-property>
         <config-property-name>viewFile32Classes</config-property-name>
         <config-property-type>java.lang.String</config-property-type>
         <config-property-value>tuxedo.test.simpapp.View32
            </config-property-value>
         </config-property>
     <config-property>
```

```
        <config-property-name>remoteAccessPointSpec</config-property-name>
        <config-property-type>java.lang.String</config-property-type>
        <config-property-value>//localhost:12478/domainId=TDOM1_ID
          </config-property-value>
    </config-property>
    <outbound-resourceadapter>
       <connection-definition>
         <managedconnectionfactory-class>com.oracle.tuxedo.adapter.spi.Tu
          xedoManagedConnectionFactory</managedconnectionfactory-class>
         <connectionfactory-interface>javax.resource.cci.ConnectionFactor
            y</connectionfactory-interface>
         <connectionfactory-impl-class>com.oracle.tuxedo.adapter.cci.Tuxe
            doConnectionFactory</connectionfactory-impl-class>
         <connection-interface>javax.resource.cci.Connection</connection-
           interface>
         <connection-impl-class>com.oracle.tuxedo.adapter.cci.TuxedoJCACo
           nnection</connection-impl-class>
    </connection-definition>
<!--
    <transaction-support>NoTransaction</transaction-support>
    <transaction-support>LocalTransaction</transaction-support>
-->
    <transaction-support>XATransaction</transaction-support>
    <authentication-mechanism>
        <authentication-mechanism-type>BasicPassword</authentication-mecha
         nism-type>
        <credential-interface>javax.resource.spi.security.PasswordCredenti
         al</credential-interface>
    </authentication-mechanism>
    <reauthentication-support>false</reauthentication-support>
  </outbound-resourceadapter>
 </resourceadapter>
</connector>
```

Listing 45 shows an example of a property that enables AUTOTRAN.

Note: This configuration can also be done through the console in a WebSphere environment.

Listing 45 Property Enabling AUTOTRAN Example

```
<config-property>
  <config-property-name>autoTran</config-property-name>
  <config-property-type>java.lang.Boolean</config-property-type>
  <config-property-value>true</config-property-value>
  </config-property>
```

Oracle Tuxedo UBBCONFIG and BDMCONFIG Examples

In addition to configuring the Tuxedo JCA Adapter configuration file, the Oracle Tuxedo `UBBCONFIG` and `BDMCONFIG` configuration files must include the Tuxedo JCA Adapter configuration in order to enable the application.

Listing 46 and Listing 47 show `UBBCONFIG` and `BDMCONFIG` file snippet examples required to expose services inside an Oracle Tuxedo Application Domain and inter-domain requests.

Listing 46 UBBCONFIG File Snippet Example

```
*SERVICES
TOUPPER
TOUPPER_V32
...
```

Listing 47 BMDCONFIG File Snippet Example

```
*DM_LOCAL_SERVICES
TOUPPER
TOUPPER_V32
...

*DM_REMOTE_SERVICES
```

```
TOLOWER
TOLOWER_POJO
```

Oracle Tuxedo JCA Adapter MDB Programming

Interface TuxedoMDBService

The Tuxedo JCA Adapter provides an **EJB MDB** interface that you must implement in your **EJB** application code.

Note: The **MDB** interface is similar to the existing **EJB** supported by Tuxedo JCA Adapter; however, they are not the same.

Listing 48 shows the interface listing.

Listing 48 Interface Listing

```
package com.oracle.tuxedo.adapter.intf;

import weblogic.wtc.jatmi.Reply;

import com.oracle.tuxedo.adapter.tdom.TPServiceInformation;

import com.oracle.tuxedo.adapter.TuxedoReplyException;

public interface TuxedoMDBService {

public Reply service(TPServiceInformation service) throws
TuxedoReplyException;

}
```

This is different from a *JMS*-based **MDB**; it uses the `service()` interface instead of the `onMessage()` interface. Listing 49 shows an **MDB** code example that implements the "*Tolower*" service for an Oracle Tuxedo client.

Listing 49 MDB Code Example

```
package ejbs;
import com.oracle.tuxedo.adapter.TuxedoReplyException;
import com.oracle.tuxedo.adapter.intf.TuxedoMDBService;
import com.oracle.tuxedo.adapter.tdom.TPServiceInformation;
import javax.ejb.MessageDrivenBean;
import javax.ejb.MessageDrivenContext;
import javax.jms.Message;
import weblogic.wtc.jatmi.Reply;
import weblogic.wtc.jatmi.TypedString;

public class TolowerMDBBeanBean
    implements MessageDrivenBean, TuxedoMDBService
{

    public TolowerMDBBeanBean()
    {
    }

    public MessageDrivenContext getMessageDrivenContext()
    {
        return fMessageDrivenCtx;
    }

    public void setMessageDrivenContext(MessageDrivenContext ctx)
    {
        fMessageDrivenCtx = ctx;
```

```
    }

    public void ejbCreate()
    {
    }

    public void onMessage(Message message)
    {
    }

    public void ejbRemove()
    {
    }

    public Reply service(TPServiceInformation mydata)
        throws TuxedoReplyException
    {
        TypedString data = (TypedString)mydata.getServiceData();
        String lowered = data.toString().toLowerCase();
        TypedString return_data = new TypedString(lowered);
        mydata.setReplyBuffer(return_data);
        return mydata;
    }
  private static final long serialVersionUID = 1L;
  private MessageDrivenContext fMessageDrivenCtx;
}
```

Creating an Inbound Connector-Based MDB Using IBM ASTK

This procedure creates a Connector-Based EJB 2.1 MDB using WebSphere ASTK 6.1. The simple EJB MDB echoes the input string back to the Oracle Tuxedo Client. The name of the project is called EchoMDB.

Use J2EE Perspective

If you are not already in "J2EE" perspective, do the following to change to "J2EE" perspective.

From menu *Window* select ***Open Perspective***, and then select ***J2EE***.

Create EJB Project

From menu "***File***" select ***New***, then select ***Project…***. Expand **EJB** by clicking it, and then highlight ***EJB Project***. Click ***Next***.

In ***EJB Project*** menu fill in ***Project Name*****" with "*****EchoMDB***. Click ***Next.*** The "*Select Project Facets*" menu will be shown.

In "*Select Project Facets*" menu, make sure "*EJB Module*" version is "*2.1*", "*Java*" version is "*5.0*", and "WebSphere EJB (Extended" version is "*6.1*", and make sure these three are selected. Click on "*Next*".

In "*EJB Module*" menu uncheck "*create an EJB Client JAR module to hold the client interface and classes*" since inbound EJB is invoked by Tuxedo JCA Adapter, it is not required. Click on "*Finish*".

Setup Build Environment

Right click on project **EchoMDB** in the **Project Explorer**. Select **Properties** from the context menu, **the Properties for EchoMDB** window appears as shown in Figure 1.

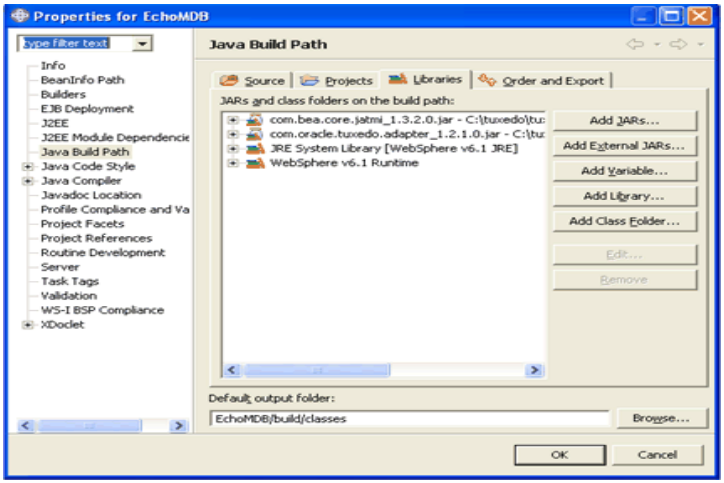
Select **Add External JARs…** from the "Java Build Path". Add the following two Jar files from Tuxedo JCA Adapter RAR file. (If you have not unzipped the RAR file, do so now.)

```
com.bea.core.jatmi_1.3.2.0.jar
```

```
com.oracle.tuxedo.adapter_1.2.1.0.jar
```

Click **OK**.

Figure 1 EchoMDB Properties Window

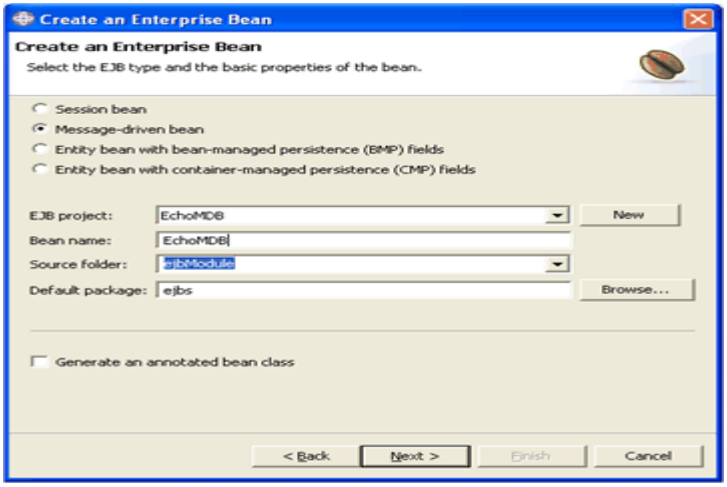


Create Message-Driven Bean

In the left Window pane under the **Project Explorer,** expand the newly created MDB project EchoMDB. Right click **EchoMDB,** select **New**, and then select **Other**. Select **Enterprise Bean** and click **Next**. The **Create an Enterprise Bean** popup window appears as shown in Figure 2.

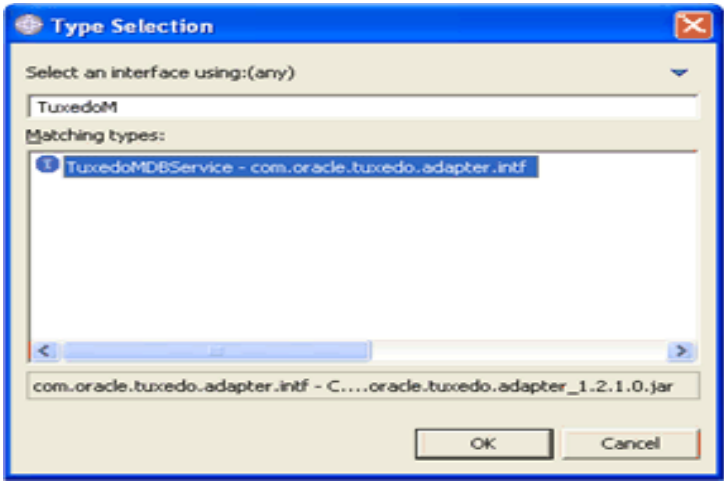
In the popup window select **Message-driven bean.** Enter the Bean name with value **EchoMDB**.

Figure 2 Create an Enterprise Bean Window



Click **Next**. The “Message Driven Bean type” popup window appears as shown in Figure 3. Select **Other Type** and then click **Browse**. Enter **TuxedoMDBService** and select from the list shown in Figure 3, then click **OK**.

Figure 3 Message Driven Bean Type Window

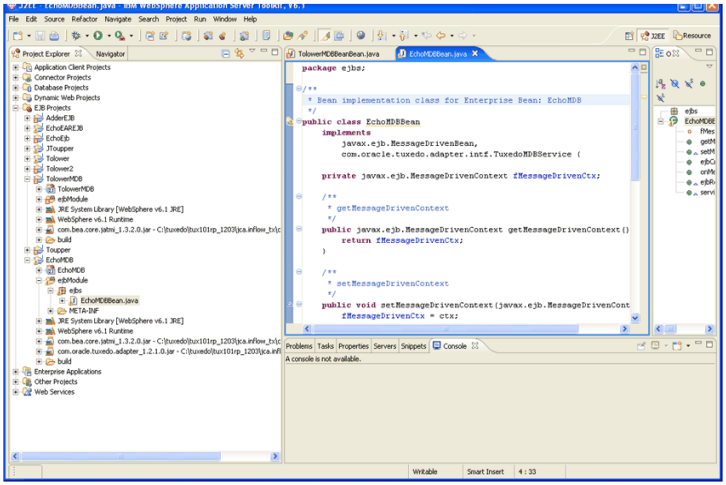


The **Message Driven Bean type** popup window appears. Click **Finish**.

Modify EchoBean

Expand **ejbModule** in the left window pane until you see **EchoMDBBean.java**. EchoMDBBean.java must be modified to perform the ECHO service. Double click **EchoMDBBean.java** and the edit window pane with default editor appears as shown in Figure 4.

Figure 4 Edit Window



Add the following lines shown in Listing 1at the top of the class file.

Listing 1 Add New Lines

```
import weblogic.wtc.jatmi.Reply;
import weblogic.wtc.jatmi.TPException;
import weblogic.wtc.jatmi.TPReplyException;
import weblogic.wtc.jatmi.TypedString;

import com.oracle.tuxedo.adapter.TuxedoReplyException;
import com.oracle.tuxedo.adapter.intf.TuxedoMDBService;
import com.oracle.tuxedo.adapter.tdom.TPServiceInformation;
```

Edit the method `service()` at the end of the class file as shown in Listing 2.

Listing 2 service()

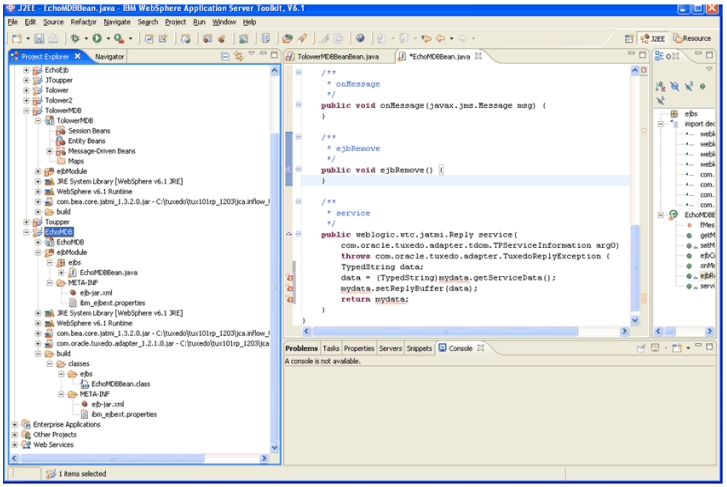
```
public weblogic.wtc.jatmi.Reply service(
```

```
        com.oracle.tuxedo.adapter.tdom.TPServiceInformation mydata)
         throws com.oracle.tuxedo.adapter.TuxedoReplyException {
  TypedString data;
  data = (TypedString)mydata.getServiceData();
  mydata.setReplyBuffer(data);
  return mydata;
}
```

Build

Right click project **EchoMDB** in the Project Explorer, and then select **Deploy** as shown in Figure 5. This compiles it into class in the build directory.

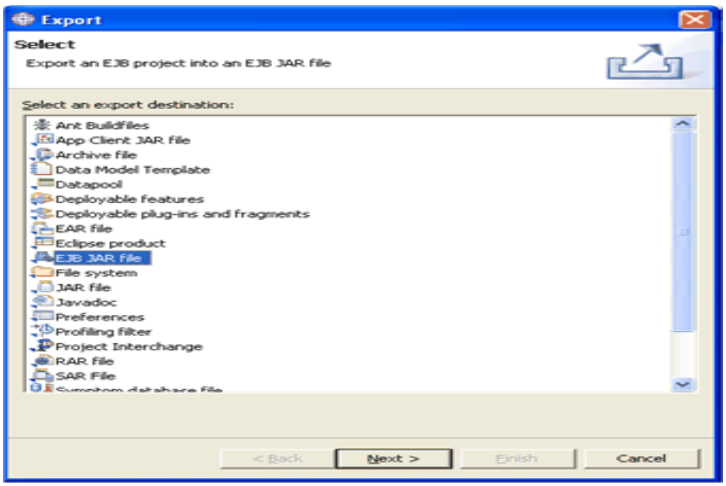
Figure 5 Compile



Create EJB JAR File

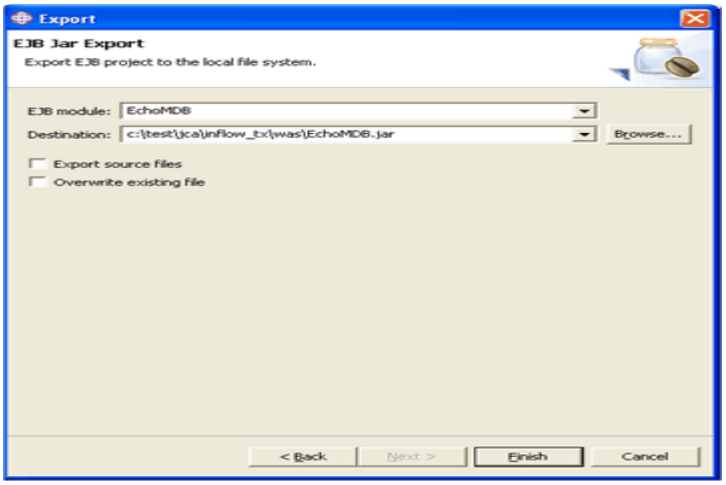
Right click the project **EchoMDB** in the Project Explorer and select **Export.** The **Export** menu popup appears as shown in Figure 6.

Figure 6 Export Popup Window



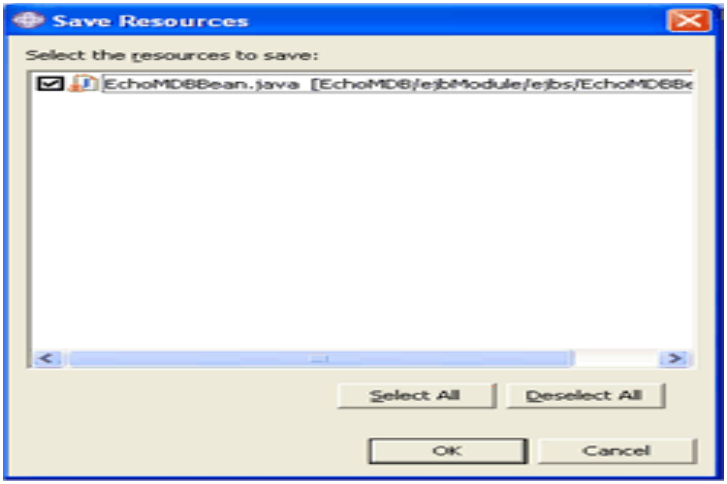
In the Export popup window, select **EJB JAR file**. Click **Next**. The **EJB Jar Export** popup window appears as shown in Figure 7. Select **EchoMDB** from the drop down menu, and enter the complete path of the jar file name in the **Destination:** text field. Click **Finish**.

Figure 7 EJB Jar Export Popup Window



The **Save Resources** popup window appears as shown in Figure 8 click “OK”.

Figure 8 Save Resources Popup Window



For Tuxedo JCA Adapter dispatching-based MDB, you must add `activation-config-property` to its `ejb-jar.xml` file using one of two ways.

1. The first method is to unzip the jar file. After the jar file is unzipped, modify the `META-INF/ejb-jar.xml`, and then re-jar the bean jar file. Listing 1 shows an example `ejb-jar.xml` file suitable to this type of MDB.

Listing 1 ejb-jar.xml File Example

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar id="ejb-jar_ID" version="2.1"
xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd">
   <display-name>EchoMDB</display-name>
   <enterprise-beans>
      <!-- message driven descriptor -->
      <message-driven id="EchoMDB">
         <ejb-name>EchoMDB</ejb-name>
```

```
         <ejb-class>ejbs.EchoMDBBean</ejb-class>
         <!-- message listener interface -->

<messaging-type>com.oracle.tuxedo.adapter.intf.TuxedoMDBService</messag
ing-type>
         <transaction-type>Container</transaction-type>
         <!-- the values for the Activation Spec JavaBean -->
         <activation-config>
            <activation-config-property>

<activation-config-property-name>source</activation-config-property-name>

<activation-config-property-value>eis/echo</activation-config-property-val
ue>
            </activation-config-property>
         </activation-config>
      </message-driven>
   </enterprise-beans>
</ejb-jar>
```

Where `eis/echo` is the JNDI name of `EchoMDB`.

2. Similarly, the second method is to modify `ejb-jar.xml` file directly to add `activation-config-property` using ASTK before the MDB is being deployed and exported.

Oracle Tuxedo Conversational Server Programming Tips

When a Java conversational client initiates a `tpconnect()` call, the Oracle Tuxedo "C" language conversational server service routine is invoked; however, before the service routine is invoked

the Oracle Tuxedo system implements a RECEIVE command implicitly and puts all the data and events into the input argument of the service routine.

If the Java client initiates tpconnect (TPSENDONLY), the Oracle Tuxedo server will have data in TPSVCINFO->data, and TPSVCINFO->len. The TPSENDONLY flag is translated to TPRECVONLY and conveyed in the TPSVCINFO->flags field. In this case, the TPRECVONLY flag can be ignored by the server service routine, and the server should continue initiating tprecv() until it receives a TPEV_SENDONLY event.

If the Java client initiates tpconnect(TPRECVONLY), the Oracle Tuxedo server will have both event and data in TPSVCINFO. The event is in the TPSVCINFO->flags field, and user data is in TPSCVINFO->data, the data length is in TPSVCINFO->len. The server should start initiating tpsend() until it either finishes the conversation with tpreturn(), or returns control to the client with tpsend(TPRECVONLY).

WARNING: You can also choose not to send data, in this case TPSVCINFO->data will be NULL.

The following is a typical Oracle Tuxedo conversational server code example that handles tpconnect().

Listing 2

```
void
CONV_TOUPPER(TPSVCINFO rqst)
{
  if (rqst->data != NULL && rqst->len > 0) {
     /* received meaningful data from client's tpconnect(). */
     …
  }
  If (rqst->flags & TPSENDONLY) {
    /* client does a tpconnect(TPRECVONLY) so server can start sending data
immediately */
    …
  }
```

```
  else {
   /* client does a tpconnect(TPSENDONLY) so server just have to tprecv()
until it gains control */
  }
…
}
```

It is permissible for client to initiate `tpconnect()` with no data, but just the flags. The server should expect the possibility of not receiving data when the service routine is invoked.

See Also

- Oracle Tuxedo JCA Adapter Users Guide
- Oracle Tuxedo JCA Adapter Reference Guide
- Oracle Tuxedo Metadata Repository Documentation