Oracle® Tuxedo

Programming an Oracle Tuxedo Application Using Java

12*c* Release 1 (12.1.1)

June 2012

ORACLE

Oracle Tuxedo Programming an Oracle Tuxedo Application Using Java, 12*c* Release 1 (12.1.1)

Contents

4. Implementing Services in Oracle Tuxedo Java Server

5. Reference

CHAPTER 1

Introduction to Oracle Tuxedo Java Programming

This topic includes the following sections:

Overview

An Oracle Tuxedo service can be developed using pure Java language. The service implemented with Java language functions the same as other Tuxedo services. You can call the services advertised by the Tuxedo Java server (`TMJAVASVR`) using ATMI interfaces from client/ Tuxedo server, and similarly, you can call the services advertised by the Tuxedo server using TJATMI interfaces from the java-implemented service.

Besides, you can call java-implemented services from any type of Tuxedo clients, such as native clients, /WS clients, and Jolt clients.

It is supported to use a variety of mainstream Java technologies like TJATMI interface, JATMI TypedBuffers, POLO java object, and so on to implement Tuxedo services.

Programing Guidelines

The following guidelines are basic instructions for Java service development.

- Java server class, which implements Java services, should inherit the `TuxedoJavaServer` class; Java server class also should have a default constructor.
- In Java server class, Java method, which will be advertised as Java service, should take the `TPSVCINFO` interface as the only input argument and should be declared to public.
- Java server class should implement `tpsvrinit()` method, which will be called when Tuxedo Java server starts up.
- Java server class should implement `tpsvrdone()` method, which will be called when Tuxedo Java server shuts down.
- Java service could use Tuxedo Java ATMI (e.g, `tpcall`, `tpbegin`, etc).
- Java service could return result to client by using `tpreturn` and exit by throwing exception.

Tuxedo Java Server Threads and Java Class Instance Model

- Tuxedo Java server uses traditional Tuxedo multithread model and must be running in multithread mode.
- Once started, Tuxedo Java server creates one global object (instance) for each class defined in the configuration file and then the working threads share the global object (instance) when handling the Java service.

Tuxedo Java Server tpsvrinit()/tpsvrdone() Handling

tpsvrinit() Handling

Users need to implement the `tpsvrinit()` method. Given that `tpsvrinit()` will be called when server starts up, it's recommended to put the class scope initialization in this method. If one class' `tpsvrinit()` fails, a warning message will be reported in user log and the Java server will continue its execution.

tpsvrdone() Handling

Users need to implement the `tpsvrdone()` method. Given that `tpsvrdone()` will be called when the server shuts down, it's recommended to put the class scope cleanup actions in this method.

Tuxedo Java Server tpreturn() Handling

The `tpreturn()` in Java service does not immediately disrupt the Java service method's execution but provide the return result to Tuxedo Java server.

How does `tpreturn()` behave in Java service is different from how does `tpreturn()` behave in the existing Tuxedo system.

- When a `tpreturn()` is called in the existing Tuxedo system, the flow control is transferred to Tuxedo automatically.
- When a `tpreturn()` is called in Java service, statements after `tpreturn()` will still be executed. Users must make sure `tpreturn()` is the last execution statement in Java service - if not, we suggest to use a following Java `return` invocation after `tpreturn()`; otherwise, `tpreturn()` will not transfer the flow control to the Tuxedo system automatically.

Note: The use of a Java `return` statement in Java service without a previous `tpreturn()` statement is not suggested - such use will make the Java server return `TPFAIL` with rcode setting 0 to the corresponding client.

Tuxedo Java Server Exception Handling

- Java service can throw any exception during execution and exit the Java service. In this case the Java server will return `TPFAIL` with rcode setting to 0 for this service to its client.
- All the exception information is recorded into the `$APPDIR/stderr` file.

CHAPTER 2

Programming Environment

This topic includes the following sections:

- Updating the UBB Configuration File

Updating the UBB Configuration File

You need to configure the path of where the Tuxedo Java server finds the configuration file for the Java-implemented services in CLOPT.

As the ATMI Java server is a multithread server, you also need to specify the limit of dispatching threads. For more information about the multithread server configuration, see Defining the Server Dispatch Threads.

Listing 2-1 shows an example of UBB configuration for ATMI Java Server.

Listing 2-1 UBB Configuration for ATMI Java Server

```
*SERVERS
TMJAVASVR SRVGRP=TJSVRGRP SRVID=3
CLOPT="-- -c /home/oracle/app/javaserver/TJSconfig.xml"
MINDISPATCHTHREADS=2 MAXDISPATCHTHREADS=3
```

Note: The minimum value of `MAXDISPATCHTHREADS` you specify in `UBBCONFIG` for the Java server should be 2.

See Also

- *Setting Up an Oracle Tuxedo Application*
- `UBBCONFIG(5)` in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*

CHAPTER 3

ATMI Java Server User Interfaces

This topic includes the following sections:

TuxedoJavaServer

`TuxedoJavaServer` is an abstract class, which should be inherited by all the user-defined classes that implement the services.

Table 3-1 TuxedoJavaServer Interfaces

Function	Description
`tpsvrinit`	An abstract method, which should be implemented by child class to do some initialization works
`tpsvrdone`	An abstract method, which should be implemented by child class to do some cleanup works
`getTuxAppContext`	Use to retrieve the current attached Tuxedo application Java context.

Oracle Tuxedo Java Context

To access the TJATMI primitives provided by Oracle Tuxedo Java server, you need to get a `TuxAppContext` object that implements all the TJATMI primitives.

Because the service class inherits from `TuxedoJavaServer`, you can call `getTuxAppContext()` in the service to get the context object. However, you cannot get `TuxAppContext` in `tpsvrinit()` because the `TuxAppContext` is not ready at this time. If you try to get the `TuxAppContext` object in `tpsvrinit()`, `tpsvrinit()` will fail and throw an exception.

TJATMI Primitives for Tuxedo Java Applications

TJATMI is a set of primitives that provides communication between clients and servers, such as calling the services, starting and ending transactions, getting the connection to DataSource, logging, and etc.

For more information, please refer to Jave Server Javadoc.

Table 3-2 TJATMI Primitives

Name	Operation
`tpcall`	Use for synchronous invocation of an Oracle Tuxedo service during request/response communication.
`tpreturn`	Use to set the return value and data in Tuxedo Java Server.
`tpbegin`	Use to begin a transaction.
`tpcommit`	Use to commit the current transaction
`tpabort`	Use to abort the current transaction
`tpgetlev`	Use to check if a transaction is in progress
`getConnection`	Use to get a connection to the configured DataSource
`userlog`	Use to print the user log in Tuxedo user log file

Note: The service continues running after `tpreturn` ends execution. It is recommended put `tpreturn()` as the last executive statement in the service.

TypedBuffers for Tuxedo Java Applications

ATMI Java server reuses the Oracle WebLogic Tuxedo Connector TypedBuffers that corresponds to Oracle Tuxedo typed buffers. Messages are passed to servers in typed buffers. The ATMI Java server provides the following buffer types in Table 3-3:

Table 3-3 TypedBuffers

Buffer Type	Description
`TypedString`	Buffer type used when the data is an array of characters that terminates with the null character. Oracle Tuxedo equivalent: `STRING`.
`TypedCArray`	Buffer type used when the data is an undefined array of characters (byte array), any of which can be null. Oracle 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. Oracle Tuxedo equivalent: `FML`.
`TypedFML32`	Buffer type similar to TypeFML but allows for larger character fields, more fields, and larger overall buffers. Oracle Tuxedo equivalent: `FML32`.
`TypedXML`	Buffer type used when data is an XML based message. Oracle 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. Oracle Tuxedo equivalent: `VIEW`
`TypedView32`	Buffer type similar to View but allows for larger character fields, more fields, and larger overall buffers. Oracle Tuxedo equivalent: `VIEW32`.

For more information about TypedBuffers, please see the Package of "weblogic.wtc.jatmi".

Additionally, "Using FML with Oracle Tuxedo Java Server" and "Using VIEW with Oracle Tuxedo Java Server" in Reference are useful for you to use TypedFML/TypedFML32 and/or TypedView/TypedView32 in Java server class.

Limitations for Typedbuffer Support

- `Fldid()/Fname()` for the `TypedFML32` which is embedded in another `TypedFML32` cannot work. To work around this issue, you can use the `fieldtable` class instead for name/id transferring.
- The `weblogic.wtc.gwt.XmlViewCnv/XmlFmlCnv` class is not available for the present.

Get/Set Service Information

Use the `TPSVCINFO` class to get/set service information sent by the Oracle Tuxedo client.

Table 3-4 Getter Functions

Function	Description
`getServiceData`	Use to return the service data sent from the Oracle Tuxedo Client.
`getServiceFlags`	Use to return the service flags sent from the Oracle Tuxedo Client.
`getServiceName`	Use to return the service name that was called.
`getAppKey`	Use to get the application authentication client key.
`getClientID`	Use to get the client identifier for originating client.

Use `TuxATMIReply` to get the reply data and meta-data from a service invocation.

Table 3-5 Getter Functions for Reply

Function	Description
`getReplyBuffer`	Return the (possibly null) typed buffer returned from a service
`gettpurcode`	Return the `tpurcode` returned from a service

Exception

You need to catch the exception thrown by JATMI primitives in the service, such as `tpcall()`. There are two types of exceptions that JATMI can throw:

- `TuxATMITPException`: Exception thrown that represents a TJATMI failure.

- `TuxATMITPReplyException`: Exception thrown if there was a service failure (TPESVCFAIL or TPSVCERROR) and user data may be associated with the exception.

For more information, please refer to Jave Server Javadoc.

Trace

You also need to export `TMTRACE=atmi:ulog` as you have done for traditional Tuxedo ATMI. The TJATMI API traces are written into ULOG as other ATMI traces.

CHAPTER 4

Implementing Services in Oracle Tuxedo Java Server

This topic includes the following sections:

Typical Procedures

Typical steps of implementing the services in Oracle Tuxedo Java server are as follows.

1. Define a class that inherits from `TuxedoJavaServer`
2. Provide a default constructor
3. Implement the `tpsvrinit()` and `tpsvrdone()` method
4. Implement the service method which should use `TPSVCINFO` as its only argument parameter, as follows:
   a. Get the `TuxAppContext` object using `getTuxAppContext()` method
   b. Get the client request data using `TPSVCINFO.getServiceData()` method from `TPSVCINFO` object
   c. If you have configured a DataSource, get a connection to the DataSource using `TuxAppContext.getConnection()` method

d. Do the business logic, such as call some other services using `TuxAppContext.tpcall()`, manipulate the database, etc.

e. Allocate a new TypedBuffer and put a reply data in the TypedBuffer

f. Call `TuxAppContext.tpreturn()` to return the reply data to client

Example: Implementing Java Service without Transaction

Following is a simple example that implements the `TOUPPER` service. It includes three steps:

1. Defining Java Classes: Listing 4-1
2. Creating Java Server Configuration File: Listing 4-2
3. Updating UBB Configuration File: Listing 4-3

Defining Java Classes

Listing 4-1 Java Class Definition

```
import weblogic.wtc.jatmi.TypedBuffer;
import weblogic.wtc.jatmi.TypedString;
import com.oracle.tuxedo.tjatmi.*;
public class MyTuxedoJavaServer extends TuxedoJavaServer {
    public MyTuxedoJavaServer()
    {
        return;
    }

    public int tpsvrinit() throws TuxException
    {
        System.out.println("MyTuxedoJavaServer.tpsvrinit()");
        return 0;
```

```
}

public void tpsvrdone()
{
    System.out.println("MyTuxedoJavaServer.tpsvrdone()");
    return;
}

public void JAVATOUPPER(TPSVCINFO rqst) throws TuxException {
    TypedBuffer svcData;
    TuxAppContext   myAppCtxt = null;
    TuxATMIReply    myTuxReply = null;
    TypedBuffer     replyTb = null;

    /* Get TuxAppContext first */
    myAppCtxt = getTuxAppContext();

    svcData = rqst.getServiceData();
    TypedString TbString = (TypedString)svcData;
    myAppCtxt.userlog("Handling in JAVATOUPPER()");
    myAppCtxt.userlog("Received string is:" + TbString.toString());
    String newStr = TbString.toString();
    newStr = newStr.toUpperCase();
    TypedString replyTbString = new TypedString(newStr);
    /* Return new string to client */
    myAppCtxt.tpreturn(TPSUCCESS, 0, replyTbString, 0);
}
```

```
    public void JAVATOUPPERFORWARD(TPSVCINFO rqst) throws TuxException {
        TypedBuffer svcData;
        TuxAppContext   myAppCtxt = null;
        TuxATMIReply  myTuxReply = null;
        TypedBuffer replyTb = null;
        long flags = TPSIGRSTRT;
        /* Get TuxAppContext first */
        myAppCtxt = getTuxAppContext();
        svcData = rqst.getServiceData();
        TypedString TbString = (TypedString)svcData;

        myAppCtxt.userlog("Handling in JAVATOUPPERFORWARD()");
        myAppCtxt.userlog("Received string is:" + TbString.toString());
      /* Call another service "TOUPPER" which may be implemented by another
Tuxedo Server */
        try {
            myTuxReply = myAppCtxt.tpcall("TOUPPER", svcData, flags);
            /* If success, get reply buffer */
            replyTb = myTuxReply.getReplyBuffer();
            TypedString replyTbStr = (TypedString)replyTb;
            myAppCtxt.userlog("Replied string from TOUPPER:" +
replyTbStr.toString());
            /* Return the replied buffer to client */
            myAppCtxt.tpreturn(TPSUCCESS, 0, replyTb, 0);
        } catch (TuxATMITPReplyException    tre) {
            myAppCtxt.userlog("TuxATMITPReplyException:" + tre);
            myAppCtxt.tpreturn(TPFAIL, 0, null, 0);
        } catch (TuxATMITPException    te) {
```

```
            myAppCtxt.userlog("TuxATMITPException:" + te);
            myAppCtxt.tpreturn(TPFAIL, 0, null, 0);
        }
    }

}
```

Creating Java Server Configuration File

Listing 4-2 shows an configuration example that exports `MyTuxedoJavaServer.JAVATOUPPER()` method as Tuxedo service name `JAVATOUPPER` and `MyTuxedoJavaServer.JAVATOUPPERFORWARD()` method as Tuxedo service name `JAVATOUPPERFORWARD`.

Listing 4-2 Java Server Configuration File

```
<?xml version="1.0" encoding="UTF-8"?>
<TJSconfig>
    <TuxedoServerClasses>
        <TuxedoServerClass name="MyTuxedoJavaServer"></TuxedoServerClass>
    </TuxedoServerClasses>
</TJSconfig>
```

Updating UBB Configuration File

Listing 4-3 UBB Config File Configuration

```
*GROUPS
TJSVRGRP  LMID=simple GRPNO=2
```

```
*SERVERS
TMJAVASVR SRVGRP= TJSVRGRP  SRVID=4CLOPT="-- -c TJSconfig.xml"
       MINDISPATCHTHREADS=2 MAXDISPATCHTHREADS=2
```

Example: Implementing Java Service with Transaction

Listing 4-4 shows an example that implements the `WRITEDB_SVCTRN_COMMIT` service which inserts the user request string into the table: `TUXJ_TRAN_TEST`.

It includes three steps:

1. Defining Java Classes: Listing 4-4
2. Creating Java Server Configuration File: Listing 4-5
3. Updating UBB Configuration File: Listing 4-6

Defining Java Classes

Listing 4-4 Class Definition

```
import weblogic.wtc.jatmi.TypedBuffer;
import weblogic.wtc.jatmi.TypedString;
import com.oracle.tuxedo.tjatmi.*;
import java.sql.SQLException;
/* MyTuxedoTransactionServer is user defined class */
public class MyTuxedoTransactionServer extends TuxedoJavaServer{
    public MyTuxedoTransactionServer ()
    {
        return;
    }
    public int tpsvrinit() throws TuxException
```

```
    {
        System.out.println("In MyTuxedoTransactionServer.tpsvrinit()");
        return 0;
    }

    public void tpsvrdone()
    {
        System.out.println("In MyTuxedoTransactionServer.tpsvrdone()");
        return;
    }
public void WRITEDB_SVCTRN_COMMIT(TPSVCINFO rqst) throws TuxException {
        TuxAppContext   myAppCtxt;
        TypedBuffer     rplyBuf = null;
        String          strType = "STRING";
        String          ulogMsg;
        TypedString     rqstMsg;
        Connection      connDB = null;
        Statement       stmtDB = null;
        String          stmtSQL;
        int             trnLvl, trnStrtInSVC;
        int             trnRtn;
        int             rc = TPSUCCESS;

        rqstMsg = (TypedString)rqst.getServiceData();
        myAppCtxt = getTuxAppContext();
        myAppCtxt.userlog("JAVA-INFO: Request Message Is \"" +
rqstMsg.toString() + "\"");
```

```
        rplyBuf = new TypedString("This Is a Simple Transaction Test from
Tuxedo Java Service");
        long trnFlags = 0;
        try {
            trnStrtInSVC = 0;
            trnLvl = myAppCtxt.tpgetlev();
            if (0 == trnLvl) {
                long trnTime = 6000;
                myAppCtxt.userlog("JAVA-INFO: Start a transaction...");
                trnRtn = myAppCtxt.tpbegin(trnTime, trnFlags);
                myAppCtxt.userlog("JAVA-INFO: tpbegin return " + trnRtn);
                trnStrtInSVC = 1;
            }
            connDB = myAppCtxt.getConnection();
            if (null != connDB) {
                myAppCtxt.userlog("JAVA-INFO: Get connection: (" +
                                  connDB.toString() + ").");
            }
            stmtDB = connDB.createStatement();
            if (null != stmtDB) {
                myAppCtxt.userlog("JAVA-INFO: Create statement: (" +
                                  stmtDB.toString() + ").");
            }
            stmtSQL = "INSERT INTO TUXJ_TRAN_TEST VALUES ('" +
                      rqstMsg.toString() + "')";
           myAppCtxt.userlog("JAVA-INFO: Start to execute sql (" + stmtSQL
+ ")...");
            stmtDB.execute(stmtSQL);
```

```
            myAppCtxt.userlog("JAVA-INFO: End to execute sql (" + stmtSQL +
").");
             if (1 == trnStrtInSVC) {
                     myAppCtxt.userlog("JAVA-INFO: tpcommit current
transaction...");
                     trnRtn = myAppCtxt.tpcommit(trnFlags);
                  myAppCtxt.userlog("JAVA-INFO: tpcommit return " + trnRtn);
                     trnStrtInSVC = 0;
                     if (-1 == trnRtn ) {
                       rc = TPFAIL;
    }
             }
        } catch (TuxATMIRMException e) {
           String errMsg = "ERROR: TuxATMIRMException: (" + e.getMessage()
+ ").";
             myAppCtxt.userlog("JAVA-ERROR: " + errMsg);
    rc = TPFAIL;
        } catch (TuxATMITPException e) {
           String errMsg = "ERROR: TuxATMITPException: (" + e.getMessage()
+ ").";
             myAppCtxt.userlog("JAVA-ERROR: " + errMsg);
            rc = TPFAIL;
        } catch (SQLException e) {
           String errMsg = "ERROR: SQLException: (" + e.getMessage() + ").";
             myAppCtxt.userlog("JAVA-ERROR: " + errMsg);
            rc = TPFAIL;
        } catch (Exception e) {
             String errMsg = "ERROR: Exception: (" + e.getMessage() + ").";
```

```
            myAppCtxt.userlog("JAVA-ERROR: " + errMsg);
           rc = TPFAIL;
        } catch (Throwable e) {
            String errMsg = "ERROR: Throwable: (" + e.getMessage() + ").";
            myAppCtxt.userlog("JAVA-ERROR: " + errMsg);
            rc = TPFAIL;
       } finally {
            if (null != stmtDB) {
              try {
                stmtDB.close();
              } catch (SQLException e) {}
        }
       myAppCtxt.tpreturn(rc, 0, rplyBuf, 0);
    }
}
```

Creating Java Server Configuration File

Listing 4-5 Java Server Configuration File

```
<?xml version="1.0" encoding="UTF-8"?>
<TJSconfig>
     <ClassPaths>

<ClassPath>/home/oracle/app/oracle/product/11.2.0/dbhome_2/ucp/lib/ucp.jar
</ClassPath>
```

```
<ClassPath>/home/oracle/app/oracle/product/11.2.0/dbhome_2/jdbc/lib/ojdbc6
.jar</ClassPath>
     </ClassPaths>


        <DataSources>
           <DataSource name="oracle">

<DriverClass>oracle.jdbc.xa.client.OracleXADataSource</DriverClass>
            <JdbcDriverParams>

<ConnectionUrl>jdbc:oracle:thin:@//10.182.54.144:1521/javaorcl</Connection
Url>
            </JdbcDriverParams>
          </DataSource>
        </DataSources>
       <TuxedoServerClasses>
   <TuxedoServerClass name=" MyTuxedoTransactionServer">
          </TuxedoServerClass>
       </TuxedoServerClasses>
</TJSconfig>
```

Updating UBB Configuration File

Listing 4-6 UBB Conf File Configuration

```
*GROUPS
ORASVRGRP  LMID=simple GRPNO=1
```

```
OPENINFO="Oracle_XA:Oracle_XA+Acc=P/scott/triger+SesTm=120+MaxCur=5+LogDir
=.+SqlNet=javaorcl"

TMSNAME=TMSORA TMSCOUNT=2

*SERVERS

TMJAVASVR SRVGRP=ORASVRGRP SRVID=3

       CLOPT="-- -c TJSconfig.xml"

       MINDISPATCHTHREADS=2 MAXDISPATCHTHREADS=4
```

CHAPTER 5

Reference

This topic includes the following sections:

Using FML with Oracle Tuxedo Java Server

Overview of FML

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.

For more information about using FML, see Programming a Tuxedo ATMI Application Using FML.

The Oracle WebLogic Tuxedo Connector FML API

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 Oracle 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 an Oracle Tuxedo system using FML, the following steps are required:

1. Copy the field tables from the Oracle Tuxedo system to Oracle Tuxedo Java server environment.

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

   ```
   # 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/mkfldclass32 utility supplied in the weblogic.wtc.jatmi package. This class is a utility function that reads a FML/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 Oracle Tuxedo Java server uses the class file to add, retrieve and delete field entries from an FML field.

4. Add the field table class to <Resources> section in Tuxedo Java server's configuration file (Also make sure it is also included in <ClassPath> of Tuxedo Java server's configuration file).

   For example:

   ```
   <Resources>
   <FieldTable16Classes>bankflds</FieldTable16Classes>
   </Resources>
   ```

5. Restart your Tuxedo Java server to load the field table class definitions.

Using the DynRdHdr Property for mkfldclass32 Class

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.
2. `java -DDynRdHdr=Path_to_Your_FML_Table`

   `weblogic.wtc.jatmi.mkfldclass32 userTable`

The arguments for this command are defined as follows:

Table 5-1 Arguments for the Commend to Use the DynRdHdr Property

Attribute	Description
`-DDynRdHdr`	Oracle 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.
`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.

3. Compile the `userTable` file using the following command:

   `javac userTable.java`

4. Add the field table class to <Resources> section in Tuxedo Java server's configuration file(Also make sure it is also included in <ClassPath> of Tuxedo Java server's configuration file).

   For example:

   `<Resources>`

   `<FieldTable32Classes>userTable</FieldTable32Classes>`

   `</Resources>.`

5. Restart your Tuxedo Java server to load the field table class definitions.

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 Java server is started, Java server will load the updated FML table.

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.

Gaining TypedFML32 Performance Improvements

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".

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.

Using VIEW with Oracle Tuxedo Java Server

Overview of VIEW Buffers

Oracle Tuxedo Java server allows you to use a Java `VIEW` buffer type analogous to an Oracle Tuxedo `VIEW` buffer type derived from an independent C structure. This allows Oracle Tuxedo

Java server classes and Oracle Tuxedo applications to pass information using a common structure.

For more information on Oracle Tuxedo `VIEW` buffers, see "Using a VIEW Typed Buffer" in Programming a Tuxedo ATMI Application Using C.

How to Create a VIEW Description File

Your Oracle Tuxedo Java server class and your Oracle 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 viewj compiler and the Oracle 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 5-2 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 5-2 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 an Oracle Tuxedo application. The file name for this VIEW is infoenc.

Listing 5-1 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
```

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) How to Use the viewj CompilerHow to Use the viewj Compileras a placeholder in these fields.

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 5-3 Arguments for the Commands for viewj Compiler

Argument	Description
options	• `-associated_fields:` Use to set `AssociatedFieldHandling` to true. This allows set and get 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 `AssociatedFieldHandling` is false. • `-bean_names:` Use to create set and get accessor names that follow `JavaBeans` naming conventions. The first character of the field name is changed to upper case before the set or get prefix is added. The signature of indexed set accessors for array fields changes from the default signature of void `setAfield` (T value, int index) to void `setAfield` (int index, T value). • `-compat_names:` Use to create set and get 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 `-compat_names` if `-bean_names` or `-compat_names` is not specified. • `-modify_strings:` Use to generate different Java code for encoding strings sent to Oracle Tuxedo and decoding strings received from Oracle Tuxedo. Encoding code adds a null character to the end of each string. Decoding code truncates each string at the first null character received. • `-xcommon:` Use to generate output class as extending `TypedXCommon` instead of `TypedView`. • `-xtype:` Use to generate output class as extending `TypedXCType` instead of `TypedView`. **Note:** `-compat_names and -bean_names are mutually exclusive options.`

Table 5-3 Arguments for the Commands for viewj Compiler

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.javaserver.atmi.simpview infoenc
```

- A `VIEW32` buffer is compiled as follows:

```
java weblogic.wtc.jatmi.viewj32 -compat_names -modify_strings
examples.javaserver.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

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

1. Create a `VIEW` description file for your application as described above.
2. Compile the `VIEW` description file as described above.
3. Use the set and get accessor methods to pass information to and receive information from a `VIEW` buffer as described above.
4. Import the output class of the `VIEW` compiler into your source code.
5. If necessary, compile the `VIEW` description file for your Oracle Tuxedo application and include the output in your C source file as described in "Using a VIEW Typed Buffer" in Programming a Tuxedo ATMI Application Using C.
6. Configure the fully qualified class name of the compiled Java `VIEW` description file in <Resources> section in Tuxedo Java server configuration. The class of the compiled Java `VIEW` description file should also be included in <ClassPath> of your configuration file.

   For example: (for `VIEW32`)

   ```
   <Resources>

   <ViewFile32Classes>
   examples.javaserver.atmi.simpview</ViewFile32Classes>

   </Resources>
   ```

7. Launch your Oracle Tuxedo Java Server.

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 using the `ViewFile32Classes` attribute in the <Resources> section of the Tuxedo Java server 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 5-2 Example: 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 5-2.

Listing 5-3 Example FML Description

```
*base 20000
#name    number  type   flags   comments
FLD0     10      string -       -
FLD1     20      long   -       -
FLD2     30      float  -       -
myview   50      view32 -       defined in View description file
```