Developer's Guide

iPlanet Application Server Enterprise Connector for Tuxedo

Version 6.0

806-5510-02
October 2000

Printed in the United States of America. 00 99 98 5 4 3 2 1

Contents

List of Figures

Preface

The *iPlanet Application Server Enterprise Connector for Tuxedo Developer's Guide* describes how to develop Java 2 Enterprise Edition (J2EE) compliant applications that access the BEA Tuxedo® services.

This guide assumes some understanding of:

- iPlanet Application Server administration
- iPlanet Application Server programming concepts
- The Internet and World Wide Web
- System management knowledge of BEA Tuxedo
- Familiarity with BEA Tuxedo programming
- Java 2 Enterprise Edition APIs (EJB, JSP, and Servlets)
- iPlanet Application Server Enterprise Connector for Tuxedo configuration concepts

How This Guide Is Organized

Chapter 1, "Introduction" describes Unified Integration Framework (UIF) and iPlanet Application Server Enterprise Connector for Tuxedo architecture and tools.

Chapter 2, "Programming for Tuxedo" describes how to develop J2EE applications which access Tuxedo services using the iPlanet Application Server Enterprise Connector for Tuxedo.

Documentation Conventions

File and directory paths are provided in Windows format with backslashes separating directory names. For Unix versions, the directory paths are the same, except slashes should be substituted in place of backslashes.

This guide uses URLs of the form http://*server.domain*/*path*/*file*.html

Where:

- *server* is the name of the server where you are running your application.
- *domain* is your Internet domain name.
- *path* is the directory structure on the server.
- *file* is an individual filename.

This guide uses the following font conventions:

- The `monospace` font is used for sample code and code listings, API and language elements (such as function names and class names), file names, pathnames, directory names, and HTML tags.
- *Italic* type is used for book titles, emphasis, variables and placeholders, and words used in the literal sense.

User Roles

This section describes the various user roles and their tasks. People with a variety of skills are involved with the setup of iPlanet Application Server Enterprise Connector for Tuxedo. Some of these users are listed below:

System Administrator	This person is responsible for the Tuxedo Enterprise Connector installation.
Systems or Business Analyst	This person uses the Tuxedo Management Console to configure the Tuxedo Enterprise Connector and import Tuxedo data types and services into UIF Repository.
Applications Programmer	This person who writes Servlets, JavaServer Pages (JSPs) and Enterprise Java Beans (EJBs) that call the UIF API. This person also uses the UIF Repository Browser to determine the available Function Objects (Tuxedo Services).
Tuxedo Programmer	This person develops Tuxedo services using the Application to Transaction Manager Interface (ATMI) interface.

Online Guides

You can find the *iPlanet Application Server Enterprise Connector for Tuxedo Developer's Guide* online in PDF and HTML formats at:

http://docs.iplanet.com/docs/manuals/

Related Information

In addition to this Developer's Guide, the Tuxedo Enterprise Connector comes with a Administrator's Guide. The Administrator's guide explains how to install and configure the Tuxedo Enterprise Connector.

The installer copies these publications to *ias/APPS/docs/tux* subdirectory of the root installation directory of the iPlanet Application Server.

Also refer to the *iPlanet Unified Integration Framework Developer's Guide* under the *ias/APPS/docs/bsp* subdirectory of the root installation of iPlanet Application Server for detailed information about the UIF API and Repository Browser.

In addition to these guides, there is additional information for administrators, users and developers. Use the following URL to view the related documentation:

http://docs.iplanet.com/docs/manuals/

The following lists the additional documents:

- *iPlanet Application Server Release Notes*
- *iPlanet Application Server Installation Guide*
- *iPlanet Application Server Overview Guide*
- *iPlanet Administration and Deployment Guide*
- *iPlanet Java Programmer's Guide*
- *iPlanet Application Builder Release Notes*
- *iPlanet Application Builder Installation Guide*
- *iPlanet Application Builder User's Guide*

Third Party Publications

The following BEA publications may be useful:

- BEA Tuxedo : Administrating the BEA Tuxedo System
- BEA Tuxedo : Application Development Guide
- BEA Tuxedo : Programmer's Guide
- BEA Tuxedo : Workstation Guide
- BEA Tuxedo : Reference Manual

For more information about Tuxedo technology, refer to the following books:

- Building Client/Server Applications Using Tuxedo - Carl L. Hall
- The TUXEDO System - Andrade, Carges, Dwyer, Felts

Chapter 1

Introduction

The iPlanet Application Server Enterprise Connector for Tuxedo extends the BEA Tuxedo system for Java 2 Enterprise Edition (J2EE) e-commerce applications. Using a consistent Java Application Programming Interface (API), along with the iPlanet Application Server Unified Integration Framework (UIF), the enterprise connector allows you to develop, deploy, and manage application solutions that leverage the Tuxedo transactions (services) in real time without having to learn the Tuxedo Application to Transaction Manager Interface (ATMI).

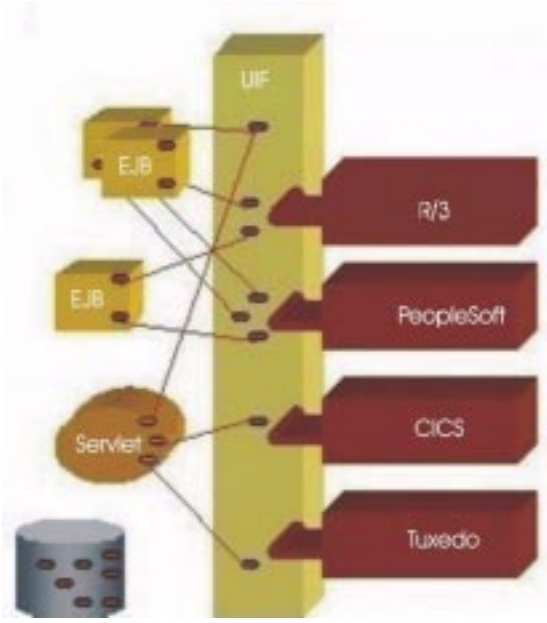
This chapter covers the following topics:

- Unified Integration Framework
- Tuxedo Enterprise Connector Architecture
- Tuxedo Enterprise Connector Tools

Unified Integration Framework

The UIF is an application programming framework that provides a single API to access different Enterprise Information Systems (EIS) using the iPlanet Application Server. As shown in Figure 1-1 an enterprise connector is developed for each EIS to allow communication between the UIF and the EIS.

Figure 1-1 The Unified Integration Framework



The framework dramatically reduces development effort by providing a consistent access layer to disparate EISs. The framework provides support for the following features:

- Connection pooling
- Thread management
- Communication and life-cycle management
- Exception management

The framework is multi-threaded to enable high-performance and fault tolerant integration. Application developers, and system integrators can easily build e-solutions accessing the various EISs using the Java Programming Language.

A universal metadata repository is also part of UIF and is used to hold information about EIS data types, business functions, and connection parameters. The EIS system administrator populates the repository using the management console provided with each connector.

Additionally, a universal repository browser allows the application developer to view business functions available and associated data types.

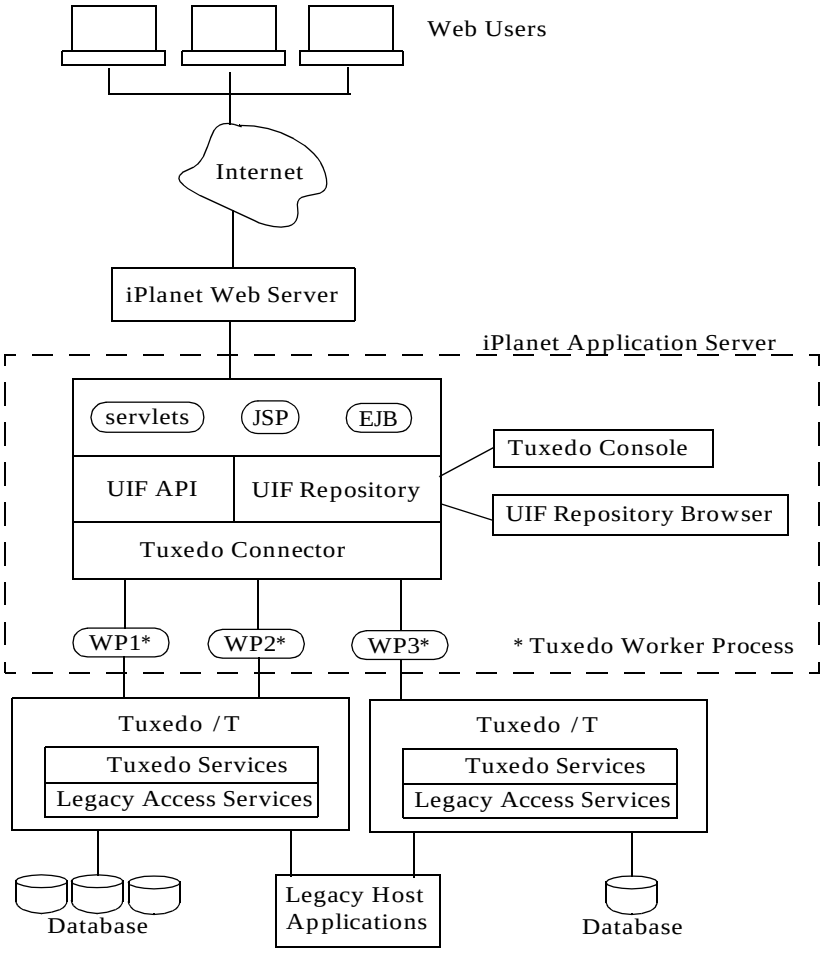
Tuxedo Enterprise Connector Architecture

The Tuxedo Enterprise Connector connects Java clients to applications built using the BEA Tuxedo system. The Tuxedo system provides a set of modular services, each offering specific functionality related to the application as a whole. For example, a simple banking application might have services like INQUIRY, WITHDRAW, TRANSFER, and DEPOSIT.

Typically, service requests are implemented in C or COBOL as a sequence of calls to a program library. In order to access these services the Tuxedo client libraries for the specific operating system on each client machine must be installed. The Tuxedo Enterprise Connector acts as a proxy for the Java clients invoking the Tuxedo services on behalf of the client. The Tuxedo Enterprise Connector accepts requests from the Java clients and translates the Java-based request into a Tuxedo request which is forwarded to the Tuxedo system. The Tuxedo system processes the request and returns the information to the Tuxedo Enterprise Connector which translates it back to the Java client.

Figure 1-2 shows the Tuxedo Enterprise Connector architecture.

Figure 1-2 The Tuxedo Enterprise Connector Architecture



Tuxedo Enterprise Connector Tools

The following tools are available with the Tuxedo Enterprise Connector:

- Tuxedo Management Console
- UIF Repository Browser

The Tuxedo Management Console is a Java-based Graphical User Interface (GUI) tool which allows browsing and configuring of the Tuxedo Enterprise Connector.

Through the Tuxedo Management Console you can:

- Create a datasource
- Edit a datasource
- Set connection pool parameters
- Set the Tuxedo authentication context
- Set the Tuxedo workstation environment variables

The GUI is also used to import the Tuxedo services definition, FML fields and VIEWs defined in the Tuxedo system into the UIF Repository. Typically, the Tuxedo administrator (domain expert) uses the Management Console to configure the enterprise connector for one or more Tuxedo systems.

The UIF Repository Browser is a Java-based GUI tool, which allows browsing of data in the UIF Repository. You can view the available business functions (Tuxedo Services), configuration parameters, and connection pools defined for a datasource. Typically an application developer uses this information while developing Java code to access the Tuxedo services.

Chapter 2

Programming for Tuxedo

The iPlanet Application Server Enterprise Connector for Tuxedo is a pre-built Java-based enterprise integration solution. The Tuxedo Enterprise Connector allows access to Tuxedo services from a J2EE compliant application.

This chapter describes how to develop Java programs using the Tuxedo Enterprise Connector.

The following topics are covered:

- About UIF Data Objects
- Mapping Tuxedo Data Types to UIF Data Types
- Developing Applications Using the Tuxedo Enterprise Connector
- Using Tuxedo Buffers with the Tuxedo Enterprise Connector
- Using the STRING Tuxedo Buffer Type
- Using the CARRAY Tuxedo Buffer Type
- Using the X_OCTET Tuxedo Buffer Type
- Using the FML Tuxedo Buffer Type
- Using the VIEW Tuxedo Buffer Type
- TransactionFO Function Object
- Developing Client-side Transactions
- Using propertySet Parameters
- Using Tuxedo User Management
- Tuxedo Application Return Code
- Error and Exception Handling

- Developing International Applications
- Tuxedo SimpApp Sample Using Servlet
- Tuxedo Online Bank Sample Using JSP & EJB

About UIF Data Objects

A UIF data object is a hierarchical data representation object and is somewhat like C/C++ structures. It can contain structures, arrays and lists, and is nested to arbitrary levels. Data objects are self describing and introspectable. Data objects are used to pass information and data across API boundaries between the application and UIF, and between UIF and the enterprise connector.

The UIF supports two types of data objects:

- Primitive Data Objects: wraps a single primitive value, such as an integer, a null terminated string, or a binary value.
- Complex Data Objects: includes structure, list, and array data objects.

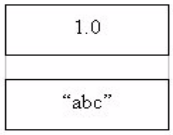
A data object can also have a type object associated with it. This is known as a `TypeInfo` object and is used internally by UIF during object creation.

Primitive Data Objects

Figure 2-1 shows a primitive data object which contains a single value of one of the following types:

- Integer
- Float
- Double
- Fixed length string (FString)
- Variable length string (String)
- Fixed size binary (Binary)
- Variable size binary (VBinary)

Figure 2-1 Primitive Data Object



Integer, float, and double

Integer, float, and double data type objects hold a value whose type corresponds to the Java data type.

Fixed length string and variable length string

Strings correspond to the Java String data type. A fixed length string has a maximum length. A variable length string has no length restriction.

Fixed size byte array and variable size byte array

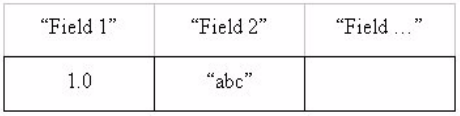
A fixed size byte array has a maximum size. A variable size byte array has no size restriction.

The maximum length of a fixed length string and the maximum size of a fixed size byte array are set when the initial value is specified.

Structure Objects

Figure 2-2 shows a structure object which contains other data objects or primitive values as fields. Each object within the structure object is referred to by a string that represents the field name. Field names have a maximum length of 64 characters. A structure’s fields are heterogeneous.

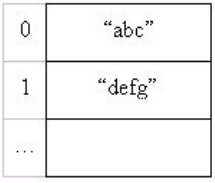
Figure 2-2 Structure Object



List Objects

Figure 2-3 shows a list object which contains data objects or primitive values as list elements and can be heterogeneous. Each element within a list object is referred to by an integer that specifies its position in the list object.

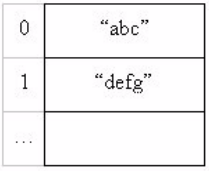
Figure 2-3 List Object



Array Objects

Figure 2-4 shows an array object which contains data objects or primitive values as array elements. Array objects inherit from list objects. The difference between an array object and a list object is that array elements must be homogeneous. Each element within the array object is referred to by an integer that specifies its position in the array object.

Figure 2-4 Array Object



Refer to the *iPlanet Unified Integration Framework Developer's Guide* for details on data objects.

Mapping Tuxedo Data Types to UIF Data Types

The BEA Tuxedo system provides the following nine built-in buffer types:

- `STRING`: a character array terminated by a null character.
- `CARRAY`: an undefined character array, any of which can be a null character. The `CARRAY` is not self describing and the length must always be provided during transmission.
- `X_OCTET`: is equivalent to a `CARRAY`.
- `FML`: a proprietary BEA Tuxedo self defining buffer where each data field carries its own identifier, an occurrence number, and possibly a length indicator. It provides great flexibility but at the expense of processing overhead because all data manipulation is done via `FML` function calls rather than native C statements.
- `FML32`: is similar to `FML` but allows for larger character fields, more fields, and larger overall buffers.
- `VIEW`: a C structure that the application defines and requires a view description file. VIEW type buffers must have subtypes which designate individual data structures.
- `VIEW32`: is similar to a `VIEW` buffer but allows for larger character fields, more fields, and larger overall buffers.
- `X_COMMON`: is similar to a `VIEW` buffer but is used with both COBOL and C programs where field types are limited to short, long, and string.
- `X_C_TYPE`: is equivalent to a `VIEW` buffer.

Tuxedo also allows custom buffers which plug into your application. Currently custom built buffer types are not supported by the Tuxedo Enterprise Connector.

Refer to BEA Tuxedo documentation for details about Tuxedo buffer types.

Tuxedo buffer types are mapped to UIF data types as follows:

Tuxedo Buffer Type	UIF Data Type
STRING	String (variable length string)
CARRAY	VBinary
X_OCTET	VBinary
FML	Struct
FML32	Struct
VIEW	Struct
VIEW32	Struct
X_COMMON	Struct
X_C_TYPE	Struct

FML/VIEW Field Type	UIF Data Type
char	Integer
int	Integer
short	Integer
long	Integer
string	String
carray	VBinary
float	Float
double	Double
dec_t	Double
multiple occurrence of FML field	Array

Developing Applications Using the Tuxedo Enterprise Connector

This section describes the basic steps involved in developing J2EE compliant applications which access the Tuxedo Service(s) using the Tuxedo Enterprise Connector.

To invoke a Tuxedo Service from a Servlet, JavaServer Page (JSP), or Enterprise JavaBeans (EJB) you must do the following:

1. Acquire a Runtime Object
2. Create a Service Provider Object
3. Create a Function Object
4. Prepare and Execute a Function Object

Acquire a Runtime Object

The runtime object is the entry point into the UIF. It is both the object factory and the access point for creating other objects. You must acquire a reference to a runtime object using the static method `getcBSPRuntime()` provided with `netscape.bsp.runtime.access_cBSPRuntime` class. This method takes three parameters; `IContext` object, `ISession2` object, and `AppLogic` object. Currently the second and third parameters are not used and null must be passed.

The following code fragment shows how to acquire a runtime object from a Servlet:

```
private IBSPRuntime getRuntime()
{
   com.kivasoft.IContext ctx =
((com.netscape.server.servlet.platformhttp.PlatformServletContex
t) getServletContext()).getContext();
   netscape.bsp.runtime.IBSPRuntime rt =
     access_cBSPRuntime.getcBSPRuntime(ctx, null, null);
   return rt;
}
```

Create a Service Provider Object

A service provider object is the logical representation of a connection to an EIS. Typically, the service provider is not bound to a physical connection until it is needed. It contains configuration information such as, host name, port number, application password, user name, client name, and data to create a connection. A service provider also needs the service provider type to manage the connection. A service provider must be enabled before being used.

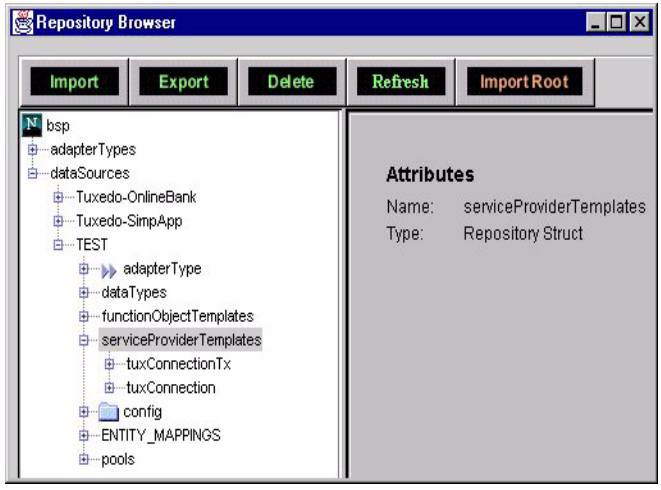
The `createServiceProvider()` method on `IBSPRuntime` interface is used to create the service provider and takes two parameters:

- the datasource name
- the service provider type

The Tuxedo Enterprise Connector supports the following two service provider templates, as shown in Figure 2-5:

- `tuxConnectionTx`
  - A virtual connection to Tuxedo to perform a *transactional* service invocation
- `tuxConnection`
  - A virtual connection to Tuxedo to perform a *non transactional* service invocation

Figure 2-5 Service Provider Types



The following code fragment illustrates how to create a service provider object:

```
private IBSPServiceProvider getServiceProvider(IBSPRuntime
runtime)
{
    if (runtime != null)
    {
        return runtime.createServiceProvider("Tuxedo-OnlineBank",
"tuxConnection");
    }

    return null;
}
```

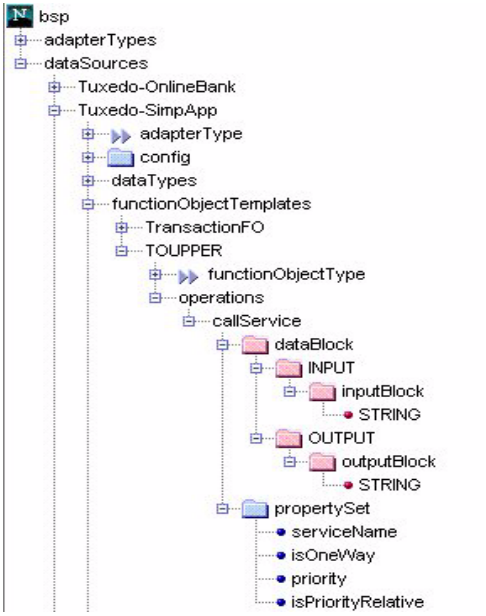
Where: `Tuxedo-OnlineBank` is the datasource name.

`tuxConnection` is the service provider type.

Create a Function Object

A function object represents the logical business operations available on a datasource. The function object represents a Tuxedo Service definition in the BEA Tuxedo system. The function object needs to be set up and associated with a service provider before it can be executed. Figure 2-6 shows a function object.

Figure 2-6 Function Object



The following table shows the key components of a function object:

Component	Description
`operations`	List of operations available in the function object
`callService`	The Tuxedo service is mapped to a function object with one operation `callService`
`dataBlock`	The input and output data block definition for the service
`INPUT`	The input definition for the service
`inputBlock`	(Optional) Holds input parameters for this service
`OUTPUT`	The output definition for the service

`outputBlock`	(Optional) Holds output values for this service

`propertySet`	The supported properties set used for service invocation, are as follows:
	`serviceName` (read-only)
	`isOneWay`
	`priority`
	`isPriorityRelative`

The `createFunctionObject()` method on `IBSPRuntime` interface is used to create the function object and takes two parameters: the datasource name and the function object name.

The following code fragment illustrates how to create a function object:

```
...
IBSPServiceProvider sp = getServiceProvider(runtime);
IBSPFunctionObject fn =
runtime.createFunctionObject("Tuxedo-OnlineBank", "OPEN_ACCT");
....
```

Where: `Tuxedo-OnlineBank` is the datasource name.

`OPEN_ACCT` is the function object name.

Prepare and Execute a Function Object

Each function object needs a connection to an EIS before it can be executed. This connection is specified by associating a service provider with function object. The function object may have multiple operations, but must be prepared for a specific operation before it can be executed. Currently, function objects other than `TransactionFO` supports only one operation `callService`.

To set up and execute a function object:

1. Enable the service provider and associate it with a function object.

2. Prepare the function object, set up the property set, and set up the input parameters in the function object's data block.
3. Execute the function object.
4. Retrieve the output parameters from the function object's data block.
5. Disable the service provider.

The following code fragment illustrates how to prepare and execute the function object:

```
// Create a function object
fn = runtime.createFunctionObject("Tuxedo-OnlineBank",
"INQUIRY");
// Enable the service provider
sp.enable();
// Associate a service provider with a function object
fn.useServiceProvider(sp);
// Prepare the function object for operation 'callService'
fn.prepare("callService");
// Set inputs
data = fn.getDataBlock();
data.setAttrInt("INPUT.inputBlock.ACCOUNT_ID", acctnum);
// Execute the function object
fn.execute();
// Get output
String outputBalance =
data.getAttrString("OUTPUT.outputBlock.SBALANCE");
// Disable the service provider
sp.disable();
```

Sample Code Walk-through

```
import java.io.*;
import java.util.*;
```

```
import javax.servlet.*;
import javax.servlet.http.*;
// Import the following UIF Packages
import netscape.bsp.*;
import netscape.bsp.runtime.*;
import netscape.bsp.dataobject.*;
public class TuxSamples extends HttpServlet {
private IBSPRuntime         rt = null;
private IBSPServiceProvider sp = null;
private IBSPFunctionObject  fo = null;
private IBSPDataObject data   = null;
private IBSPDataObject prop   = null;
private IBSPDataObjectStructure config = null;
private com.kivasoft.IContext ctx = null;
private String strOperName   = "callService";
private String strDataSource = "TEST";
private String strFOName     = "";
public void init (ServletConfig config) throws ServletException {
   super.init(config);
   ctx =
((com.netscape.server.servlet.platformhttp.PlatformServletContext)
getServletContext()).getContext();
}
public void doGet(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException  {
   // add logic to call TOUPPER() method
}
private boolean TOUPPER() {
   String inputStr  = "tuxedo";
   String outputStr = "";
   boolean bRetCode = true;
```

```
    try {
        // Get the runtime instance
        rt = access_cBSPRuntime.getcBSPRuntime(ctx, null, null);
        // Create the service provider and enable it
        sp = rt.createServiceProvider("TEST", "tuxConnection");
        sp.enable();
        // Create the function object and prepare
        fo = rt.createFunctionObject("TEST", "TOUPPER");
        fo.useServiceProvider(sp);
        fo.prepare(strOperName);
        // Set input data
        data = fo.getDataBlock();
        data.setAttrString("INPUT.inputBlock.INPUTSTRING", inputStr);
        // Execute the function object
        fo.execute();
        // Get output and process
        data = fo.getDataBlock();
        outputStr =
data.getAttrString("OUTPUT.outputBlock.OUTPUTSTRING");
        if (!(inputStr.toUpperCase()).equals(outputStr)) {
            System.out.println("ERROR : DATA MISMATCH");
            bRetCode = false;
        }
    } catch (BspException e) {
        // Handle exceptions
        System.out.println("ERROR : " + e);
        bRetCode = false;
    } finally {
        // Disable service provider
        if (sp != null)
            sp.disable();
```

```
    }
    return bRetCode;
    }
}
```

Using Tuxedo Buffers with the Tuxedo Enterprise Connector

The Tuxedo Enterprise Connector supports the following built-in Tuxedo buffer types:

- STRING
- CARRAY
- X_OCTET
- FML
- FML32
- VIEW
- VIEW32
- X_COMMON
- X_C_TYPE

Using the Tuxedo Enterprise Connector, you can develop J2EE applications which accesses Tuxedo services using any of the above buffer types. Please refer to BEA Tuxedo documentation for information about the Tuxedo buffer types.

The rest of the chapter provides sample code to illustrate the Tuxedo Enterprise Connector capabilities. The program examples are only code fragments used to illustrate a specific functionality. They are not intended to be compiled and run as provided, and additional code is required to be fully functional.

Using the STRING Tuxedo Buffer Type

The STRING buffer type is a character array terminated by a null character. This buffer type is useful for transmitting a character string.

The following example for TOUPPER Tuxedo Service, illustrates how the Tuxedo Enterprise Connector works with a service whose buffer type is STRING. The TOUPPER Tuxedo Service is available in the Tuxedo simpapp example. The service converts the input string to uppercase and returns it to the client.

The service definition must be imported into the UIF repository using the Tuxedo Management Console. The service definition is:

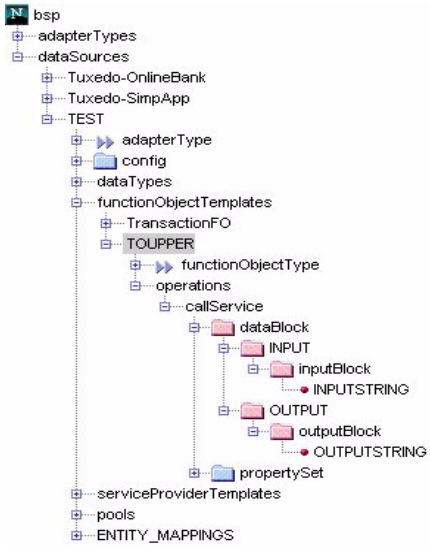
```
interface SAMPLE {

    void TOUPPER(
        [in]    STRING INPUTSTRING
        [out]   STRING OUTPUTSTRING
    );
};
```

Where: INPUTSTRING is the input parameter.

OUTPUTSTRING is the output parameter to the TOUPPER Tuxedo Service. The Tuxedo buffer type STRING is mapped to String data type in UIF.

Figure 2-7 shows how the service is mapped in the UIF repository:

Figure 2-7 TOUPPER Function Object



The following code fragment illustrates how the Tuxedo Enterprise Connector works with a service whose buffer type is `STRING`:

```
IBSPServiceProvider sp = null;

try {

    IContext ctx =
((com.netscape.server.servlet.platformhttp.PlatformServletContext)
getServletContext()).getContext();

    IBSPRuntime runtime = access_cBSPRuntime.getcBSPRuntime(ctx,
null, null);

    sp = runtime.createServiceProvider("TEST", "tuxConnection");

    IBSPFunctionObject fn = runtime.createFunctionObject("TEST",
"TOUPPER");

    sp.enable();

    fn.useServiceProvider(sp);

    fn.prepare("callService");

    IBSPDataObject data = fn.getDataBlock();

    // set the input string

    data.setAttrString("INPUT.inputBlock.INPUTSTRING", inputstr);
```

```
    fn.execute();
    // get the result string back
    resultstr =
data.getAttrString("OUTPUT.outputBlock.OUTPUTSTRING");
} catch (BspException e) {
    // handle exceptions
} finally {
    if (sp != null)
        sp.disable();
}
```

Using the CARRAY Tuxedo Buffer Type

The CARRAY buffer type is an array of characters, any of which can be a null character. The application defines the array semantics; because the semantics are not interpreted by the BEA Tuxedo system. This buffer type is used to handle data opaquely. Unlike all other built-in buffer types, the CARRAY is not self describing.

Consider a Tuxedo Service CARRAYSAMPLE whose input Tuxedo buffer type is CARRAY and output is also CARRAY. This service takes a buffer and passes it back to the client.

The service definition must be imported into the UIF repository using the Tuxedo Management Console. The service definition is:

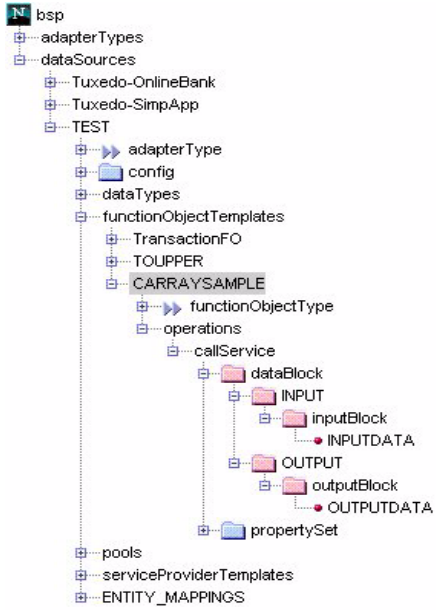
```
interface SAMPLE {

     void CARRAYSAMPLE(
          [in]  CARRAY INPUTDATA
          [out] CARRAY OUTPUTDATA
     );
};
```

Where: INPUTDATA is the input parameter.

OUTPUTDATA is the output parameter to Tuxedo Service CARRAYSAMPLE. The CARRAY Tuxedo buffer type is mapped to VBinary in UIF.

Figure 2-8 shows how the service is mapped in the UIF repository:

Figure 2-8 CARRAYSAMPLE Function Object



The following code fragment illustrates how the Tuxedo Enterprise Connector works with a service whose buffer type is CARRAY:

```
byte[] inputdata = new byte[100];
// code to fill inputdata
.....
try {
   // code to get service provider and runtime
   ......
   IBSPFunctionObject fn = null;
   IBSPDataObject data = null;
   fn = runtime.createFunctionObject("TEST", "CARRAYSAMPLE");
   sp.enable();
   fn.useServiceProvider(sp);
```

```
    fn.prepare("callService");
    data = fn.getDataBlock();
    // set the input binary data
    data.setAttrVBinary("INPUT.inputBlock.INPUTDATA", inputdata);
    fn.execute();
    // get the result binary data
    byte[] resultbytes =
data.getAttrVBinary("OUTPUT.outputBlock.OUTPUTDATA");
} catch (BspException e) {
    // handle exceptions
} finally {
    if (sp != null)
        sp.disable();
}
```

Using the X_OCTET Tuxedo Buffer Type

The X_OCTET buffer type is defined as an alias for CARRAY to support XATMI.

Consider a Tuxedo Service XOCTETSAMPLE whose input Tuxedo buffer type is X_OCTET and output is also X_OCTET. This service takes a buffer and passes it back to the client.

The service definition must be imported into the UIF repository using the Tuxedo Management Console. The service definition is:

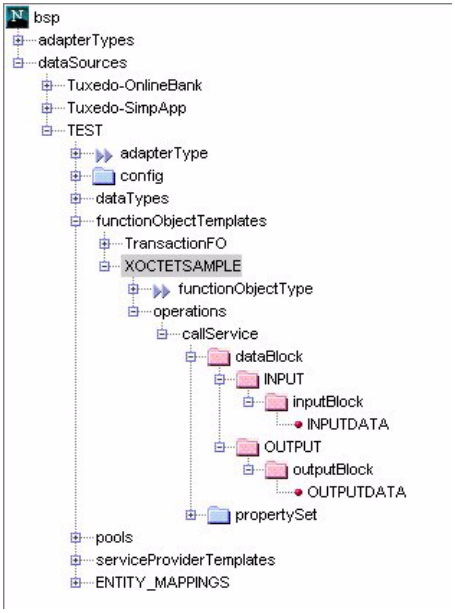
```
interface SAMPLE {

    void XOCTETSAMPLE (
        [in]  X_OCTET INPUTDATA
        [out] X_OCTET OUTPUTDATA
    );
};
```

Where: INPUTDATA is the input parameter.

OUTPUTDATA is the output parameter to Tuxedo Service XOCTETSAMPLE. The X_OCTET Tuxedo buffer type is mapped to VBinary in UIF.

Figure 2-9 shows how the service is mapped in the UIF repository:

Figure 2-9 XOCTETSAMPLE Function Object



The following code fragment illustrates how the Tuxedo Enterprise Connector works with a service whose buffer type is X_OCTET:

```
try {
    byte[] inputdata = new byte[100];
    // code to fill inputdata
    .....
    // code to get service provider and runtime
    ....
    IBSPFunctionObject fn = null;
    IBSPDataObject data = null;
    fn = runtime.createFunctionObject("TEST", "XOCTETSAMPLE");
    sp.enable();
```

```
    fn.useServiceProvider(sp);
    fn.prepare("callService");
    data = fn.getDataBlock();
    data.setAttrVBinary("INPUT.inputBlock.INPUTDATA", inputdata);
    hr = fn.execute();
    byte[] resultbytes =
data.getAttrVBinary("OUTPUT.outputBlock.OUTPUTDATA");
} catch (BspException e) {
     // handle exceptions
} finally {
    // disable service provider
     if (sp != null)
         sp.disable();
}
```

Using the FML Tuxedo Buffer Type

The FML buffer type is a self describing buffer in which each data field carries its own identifier, an occurrence number, and possibly a length indicator. The FML32 buffer is similar to FML but allows for larger character fields and more fields and larger overall buffers. The individual fields in the FML buffer can be of data types float, double, long, short, char, string, and carray.

If a field in FML buffer has multiple occurrences, then each occurrence is accessed by an index. Such a field is mapped to an array data type in UIF.

FML Sample 1

The following example for TRANSFER Tuxedo Service, illustrates how the Tuxedo Enterprise Connector works with a service whose buffer type is FML. The TRANSFER service is available in the Tuxedo bankapp example. This service uses FML as an input and output buffer. The input FML buffer has ACCOUNT_ID field of data type long with multiple occurrence and SAMOUNT field with string data type. The output FML buffer has a multiple occurrence string field SBALANCE.

Before calling this service, FML Field Table and service definitions must be imported into the UIF repository using the Tuxedo Management Console.

The service definition is:

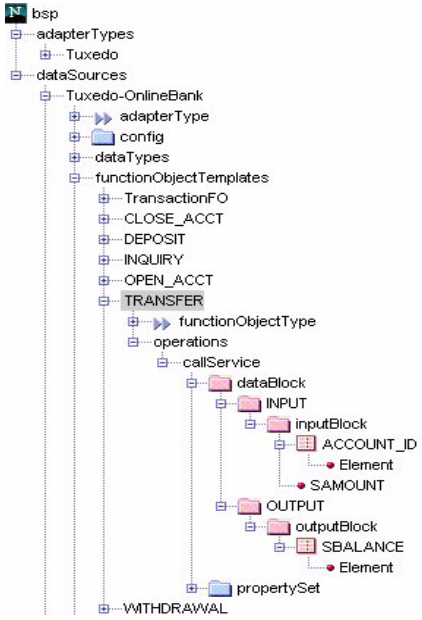
```
interface SAMPLE {

    void TRANSFER(
        [in,FML]    LONG   ACCOUNT_ID[]
        [in,FML]    STRING SAMOUNT
        [out,FML]   STRING SBALANCE[]
    );
};
```

This service uses the following FML field table definition file provided with Tuxedo bankapp application:

```
# name                          number  type    flags   comments
ACCOUNT_ID                      110     long    -       -
ACCT_TYPE                       112     char    -       -
ADDRESS                         109     string  -       -
AMOUNT                          117     float   -       -
BALANCE                         105     float   -       -
BRANCH_ID                       104     long    -       -
FIRST_NAME                      114     string  -       -
LAST_ACCT                       106     long    -       -
LAST_NAME                       113     string  -       -
LAST_TELLER                     107     long    -       -
MID_INIT                        115     char    -       -
PHONE                           108     string  -       -
SSN                             111     string  -       -
TELLER_ID                       116     long    -       -
SBALANCE                        201     string  -       -
SAMOUNT                         202     string  -       -
XA_TYPE                         203     short   -       -
CURS                            204     string  -       -
SVCHG                           205     string  -       -
VIEWNAME                        206     string  -       -
OPEN_CR                         207     char    -       -
TYPE_CR                         208     char    -       -
STATLIN                         209     string  -       -
```

The input FML field ACCOUNT_ID has multiple occurrences and hence mapped to an array in UIF. This array contains elements of type integer, which holds the account ID. The single occurrence of input FML field SAMOUNT is mapped to string data type in UIF. The output FML field SBALANCE is of string data type with multiple occurrences and hence mapped to an array data type with string elements. Figure 2-10 illustrates the definition of the TRANSFER service in the UIF repository:

Figure 2-10 TRANSFER Function Object



The following code fragment illustrates how the Tuxedo Enterprise Connector works with a service whose buffer type is FML:

```
try {
    ....
    fn = runtime.createFunctionObject("Tuxedo-OnlineBank",
"TRANSFER");
    sp.enable();
    fn.useServiceProvider(sp);
    fn.prepare("callService");
```

```
    data = fn.getDataBlock();
    // populate the input data
    data.setAttrString("INPUT.inputBlock.SAMOUNT", strAmount);
    IBSPDataObjectArray arrayObj = (IBSPDataObjectArray)
data.getAttrDataObject("INPUT.inputBlock.ACCOUNT_ID");
    arrayObj.addElemInt(Integer.parseInt(frmAccountNumber));
    arrayObj.addElemInt(Integer.parseInt(toAccountNumber));
    // call the service
   fn.execute();
    // read the output results
    arrayObj = (IBSPDataObjectArray)
data.getAttrDataObject("OUTPUT.outputBlock.SBALANCE");
    String frmBalanceStr = arrayObj.getElemString(0);
    String toBalanceStr = arrayObj.getElemString(1);
    ....
} catch (BspException e) {
    // handle exceptions
} finally {
    if (sp != null)
        sp.disable();
}
```

FML Sample 2

Consider a more complex service `FMLSAMPLE2`. This service reads in an input `FML` buffer, creates a new `FML` buffer to store the data, and passes that buffer back to the client.

Before calling this service, FML Field Table and service definitions must be imported into the UIF repository using the Tuxedo Management Console.

The service definition is as follows:

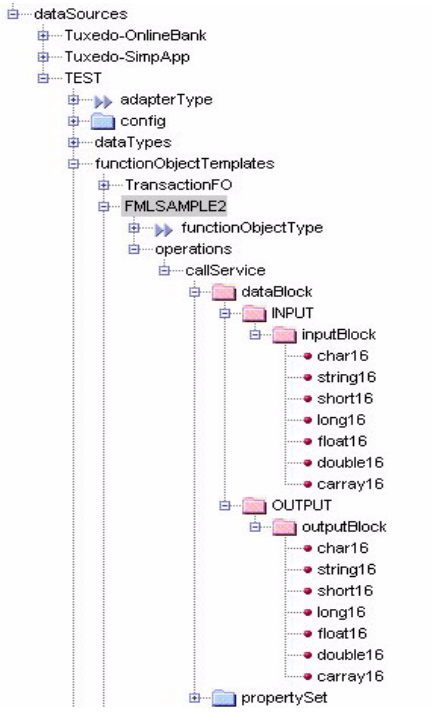
```
interface SAMPLE {

    void FMLSAMPLE2(
        [in, FML]  int    char16
        [in, FML]  String string16
        [in, FML]  short  short16
        [in, FML]  long   long16
        [in, FML]  float  float16
        [in, FML]  double double16
        [in, FML]  carray carray16

        [out, FML]  int    char16
        [out, FML]  String string16
        [out, FML]  short  short16
        [out, FML]  long   long16
        [out, FML]  float  float16
        [out, FML]  double double16
        [out, FML]  carray carray16
    );
};
```

The above service uses the following FML field table definition file:

```
# name        number  type    flags   comments
*base 1000
char16         1       char    -       this is a char field
string16       2       string  -       this is a string field
short16        3       short   -       this is a short field
long16         4       long    -       this is a long field
float16        5       float   -       this is a float field
double16       6       double  -       this is a double field
carray16       7       carray  -       this is a carray field
```

Figure 2-11 shows how the service is mapped in the UIF repository:

Figure 2-11 FMLSAMPLE2 Function Object



The following code fragment illustrates how the Tuxedo Enterprise Connector works with a service whose buffer type is FML:

```
try {
    ....
    sp.enable();
    fn = runtime.createFunctionObject("TEST", "FMLSAMPLE2");
    fn.useServiceProvider(sp);
    fn.prepare("callService");
    data = fn.getDataBlock();
    // populate the input data
data.setAttrInt("INPUT.inputBlock.char16", (int) 'a');
data.setAttrString("INPUT.inputBlock.string16", "Hello World");
data.setAttrInt("INPUT.inputBlock.long16", 9876);
```

```
data.setAttrInt("INPUT.inputBlock.short16", 8888);

data.setAttrFloat("INPUT.inputBlock.float16", 2123.1212f);

data.setAttrDouble("INPUT.inputBlock.double16", 234.234);

data.setAttrVBinary("INPUT.inputBlock.carray16", "Hello
World".getBytes());

// call the service

fn.execute();

// read the output results

String outStr1 = data.getAttrString("OUTPUT.outputBlock.string16");

long   outLong1 = data.getAttrInt("OUTPUT.outputBlock.long16");

short  outShort1 = (short)
data.getAttrInt("OUTPUT.outputBlock.short16");

float  outFloat1 = data.getAttrFloat("OUTPUT.outputBlock.float16");

double outDouble1 = data.getAttrDouble("OUTPUT.outputBlock.double16"
);

int outChar1 = data.getAttrInt("OUTPUT.outputBlock.char16");

} catch (BspException e) {

   // handle exceptions

} finally {

   if (sp != null)

      sp.disable();

}
```

FML Sample 3

Consider another service `FMLSAMPLE3`. This service reads in an input `FML` buffer, creates a new `FML` buffer to store the same data with multiple occurrences, and passes that buffer back to the client. The input `FML` buffer contains `char16`, `string16`, `short16`, `long16`, `float16`, `double16`, and `carray16` fields with a single occurrence. The service returns `FML` buffer with multiple occurrences of the same input fields.

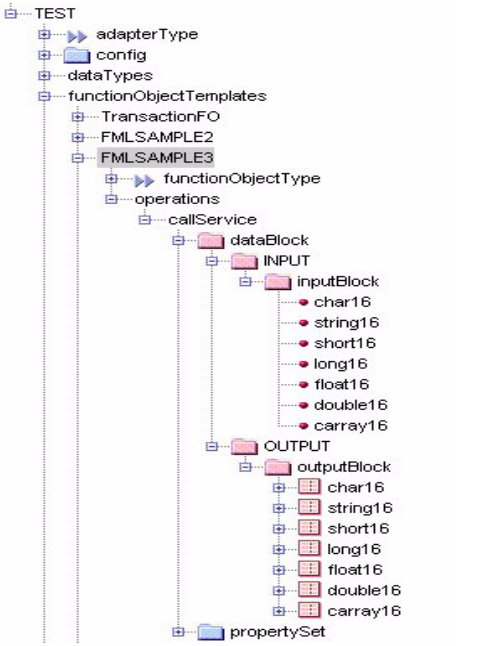
Before calling this service, FML Field Table and service definitions must be imported into the UIF repository using the Tuxedo Management Console.

The service definition is as follows:

```
interface SAMPLE {
    void FMLSAMPLE3(
        [in, FML]  int    char16
        [in, FML]  String string16
        [in, FML]  short  short16
        [in, FML]  long   long16
        [in, FML]  float  float16
        [in, FML]  double double16
        [in, FML]  carray carray16

        [out, FML]  int    char16[]
        [out, FML]  String string16[]
        [out, FML]  short  short16[]
        [out, FML]  long   long16[]
        [out, FML]  float  float16[]
        [out, FML]  double double16[]
        [out, FML]  carray carray16[]
    );
};
```

This service uses the same FML field table definition file, defined for FMLSAMPLE2 sample.

Figure 2-12 FMLSAMPLE3 Function Object



The following code fragment illustrates how the Tuxedo Enterprise Connector works with a service whose buffer type is FML:

```
// ... populating input data
fn.execute();
// read the output results
IBSPDataObjectArray stringdo = (IBSDataObjectArray)
   data.getAttrDataObject("OUTPUT.outputBlock.string16");
String outStr[] = new String[stringdo.getElemCount()];
for (int i = 0; i < outStr.length; i++)
   outStr[i] = stringdo.getElemString(i);
IBSPDataObjectArray longdo = (IBSDataObjectArray)
   data.getAttrDataObject("OUTPUT.outputBlock.long16");
long outLong[] = new long[longdo.getElemCount()];
for (int i = 0; i < outLong.length; i++)
   outLong[i] = longdo.getElemInt(i);
```

```
IBSPDataObjectArray doubledo = (IBSDataObjectArray)
   data.getAttrDataObject(“OUTPUT.outputBlock.double16”);
double outDouble[] = new double[doubledo.getElemCount()];
for (int i = 0; i < outDouble.length; i++)
   outDouble[i] = doubledo.getElemDouble(i);
// similar code for the other output fields.
.....
```

FML32 Sample

This sample illustrates how the Tuxedo Enterprise Connector works with a service whose buffer type is `FML32`. This service reads in an input `FML32` buffer, creates a new `FML32` buffer to store the data, and passes that buffer back to the client. The input `FML32` buffer contains `char32`, `string32`, `short32`, `long32`, `float32`, `double32`, and `carray32` fields with single occurrence. The service returns `FML32` buffer with single occurrence of same fields.

Before calling this service, `FML32` Field Table and service definitions must be imported into the UIF repository using the Tuxedo Management Console.

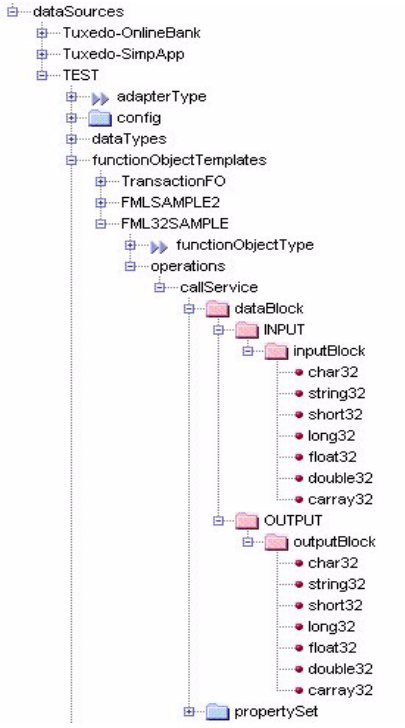
The service definition is:

```
interface SAMPLE {

    void FML32SAMPLE(
        [in, FML32]  int    char32
        [in, FML32]  String string32
        [in, FML32]  short  short32
        [in, FML32]  long   long32
        [in, FML32]  float  float32
        [in, FML32]  double double32
        [in, FML32]  carray carray32

        [out, FML32]  int    char32
        [out, FML32]  String string32
        [out, FML32]  short  short32
        [out, FML32]  long   long32
        [out, FML32]  float  float32
        [out, FML32]  double double32
        [out, FML32]  carray carray32
    );
};
```

This service uses the following FML32 field table definition file:

```
# name        number  type    flags   comments
*base 1000
char32         1       char    -       this is a char field
string32       2       string  -       this is a string field
short32        3       short   -       this is a short field
long32         4       long    -       this is a long field
float32        5       float   -       this is a float field
double32       6       double  -       this is a double field
carray32       7       carray  -       this is a carray field
```

Figure 2-13 FML32SAMPLE Function Object



The following code fragment illustrates how the Tuxedo Enterprise Connector works with a service whose buffer type is FML32:

```
try {
.....
fn = runtime.createFunctionObject("TEST", "FML32SAMPLE");
sp.enable();
fn.useServiceProvider(sp);
fn.prepare("callService");
data = fn.getDataBlock();
// populate the input data
data.setAttrInt("INPUT.inputBlock.char32", (int) 'a');
data.setAttrString("INPUT.inputBlock.string32", "Hello World");
data.setAttrInt("INPUT.inputBlock.long32", 9876);
```

```
data.setAttrInt("INPUT.inputBlock.short32", 8888);

data.setAttrFloat("INPUT.inputBlock.float32", 2123.1212f);

data.setAttrDouble("INPUT.inputBlock.double32", 234.234);

data.setAttrVBinary("INPUT.inputBlock.carray32", "Hello
World".getBytes());

// call the service

fn.execute();

// read the output results

String outStr1 = data.getAttrString("OUTPUT.outputBlock.string32");

long outLong1 = data.getAttrInt("OUTPUT.outputBlock.long32");

short outShort1 = (short)
data.getAttrInt("OUTPUT.outputBlock.short32");

float outFloat1 = data.getAttrFloat("OUTPUT.outputBlock.float32");

double outDouble1 = data.getAttrDouble("OUTPUT.outputBlock.double32"
);

int outChar1 = data.getAttrInt("OUTPUT.outputBlock.char32");

} catch (BspException e) {

   // handle exceptions

} finally {

   if (sp != null)

       sp.disable();

}
```

Using the VIEW Tuxedo Buffer Type

`VIEW` is a built-in Tuxedo typed buffer. The `VIEW` buffer provides a way to use C structures and COBOL records with the Tuxedo system. The `VIEW` buffer enables the Tuxedo runtime system to understand the format of C structures and COBOL records, that are based on the view description read at runtime. When allocating a `VIEW` buffer, your application specifies a `VIEW` buffer type and a subtype that matches the name of the view (the name that appears in the view description file). The individual fields in `VIEW` structure can be of data types `char`, `string`, `carray`, `long`, `short`, `int`, `float`, `double`, and `dec_t`.

The VIEW32 buffer type is similar to VIEW but allows for larger character fields, more fields, and larger overall buffers. The X_COMMON buffer type is similar to VIEW but is used for both COBOL and C programs so field types should be limited to short, long, and string. The X_C_TYPE buffer type is equivalent to VIEW.

If a field in VIEW buffer is an array, it is mapped to an array data type in UIF and each element of the array is of corresponding UIF data type for the field type.

If your J2EE application calls any Tuxedo Service with VIEW, VIEW32, X_COMMON or X_C_TYPE buffer types. The Tuxedo environment variables VIEWFILES, VIEWDIR, VIEWFILES32, and VIEWDIR32 must be set before starting the iPlanet Application Server.

VIEW Sample

The following sample illustrates how the Tuxedo Enterprise Connector works with a service whose buffer type is VIEW. This sample uses a Tuxedo service VIEWSAMPLE with input and output VIEW buffer. This service accepts a VIEW buffer with subtype v16test1 as an input and outputs the same data as VIEW buffer with subtype v16test2.

Before calling this service, VIEW and service definitions must be imported into the UIF repository using the Tuxedo Management Console.

The service definition is:

```
interface SAMPLE {

     void VIEWSAMPLE(
         [in, VIEW16  v16test1]  VIEW16 inputBlock
         [out,VIEW16  v16test2]  VIEW16 outputBlock
     );
};
```

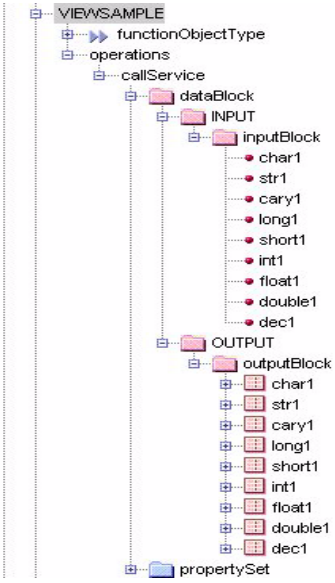
The definition of views v16test1 and v16test2 used in this service are as follows:

```
VIEW v16test1
# type cname   fbname  count flag  size null
char   char1   -       1     -     -    -
string str1    -       1     -     100  -
carray cary1   -       1     -     100  -
```

```
long   long1   -       1     -     -    -
short  short1  -       1     -     -    -
int    int1    -       1     -     -    -
float  float1  -       1     -     -    -
double double1 -       1     -     -    -
dec_t  dec1    -       1     -     9,2  -
END
VIEW v16test2
# type cname   fbname  count flag  size null
char   char1   -       5     -     -    -
string str1    -       5     -     100  -
carray cary1   -       5     -     100  -
long   long1   -       5     -     -    -
short  short1  -       5     -     -    -
int    int1    -       5     -     -    -
float  float1  -       5     -     -    -
double double1 -       5     -     -    -
dec_t  dec1    -       5     -     9,2  -
END
```

Figure 2-14 shows how the service is represented in the UIF repository.

Figure 2-14 VIEWSAMPLE Function Object



The following code fragment illustrates how the Tuxedo Enterprise Connector works with a service whose buffer type is `VIEW`:

```
try {
....
fn = runtime.createFunctionObject("TEST", "VIEWTEST1");
sp.enable();
fn.useServiceProvider(sp);
hr = fn.prepare("callService");
data = fn.getDataBlock();
// populate the input data
data.setAttrInt("INPUT.inputBlock.char1", (int) 'a');
data.setAttrString("INPUT.inputBlock.str1", "Hello World");
data.setAttrInt("INPUT.inputBlock.long1", 9876);
data.setAttrInt("INPUT.inputBlock.short1", 8888);
data.setAttrInt("INPUT.inputBlock.int1", 9999);
data.setAttrFloat("INPUT.inputBlock.float1", 2123.1212f);
```

```
data.setAttrDouble("INPUT.inputBlock.double1", 234.234);

data.setAttrVBinary("INPUT.inputBlock.cary1", "Hello
World".getBytes());

data.setAttrDouble("INPUT.inputBlock.dec1", 2123.12);

// call the service

fn.execute();

// Get the output results

for (int i = 0; i < 5; i ++) {

   String outStr1 =

   data.getAttrString("OUTPUT.outputBlock.str1" + ".["+i+"]");

   long   outLong1 =

   data.getAttrInt("OUTPUT.outputBlock.long1" + ".["+i+"]");

   short  outShort1 = (short)

   data.getAttrInt("OUTPUT.outputBlock.short1" + ".["+i+"]");

   int    outInt1 =

   data.getAttrInt("OUTPUT.outputBlock.int1" + ".["+i+"]");

   float  outFloat1 =

   data.getAttrFloat("OUTPUT.outputBlock.float1" + ".["+i+"]");

   double outDouble1 =

   data.getAttrDouble("OUTPUT.outputBlock.double1" + ".["+i+"]");

   .....

}

.....
```

VIEW32 Sample

This sample illustrates how the Tuxedo Enterprise Connector works with a service whose buffer type is `VIEW32`. This sample uses a Tuxedo service `VIEW32SAMPLE` with input and output `VIEW32` buffer. This service accepts a `VIEW32` buffer with subtype `v32test1` as an input and outputs the same data as `VIEW32` buffer with subtype `v32test2`.

Before calling this service, `VIEW32` and service definitions must be imported into the UIF repository using the Tuxedo Management Console.

The service definition is as follows:

```
interface SAMPLE {

     void VIEW32SAMPLE(
         [in,  VIEW32  v32test1]  VIEW32 inputBlock
         [out, VIEW32  v32test2]  VIEW32 outputBlock
     );
};
```

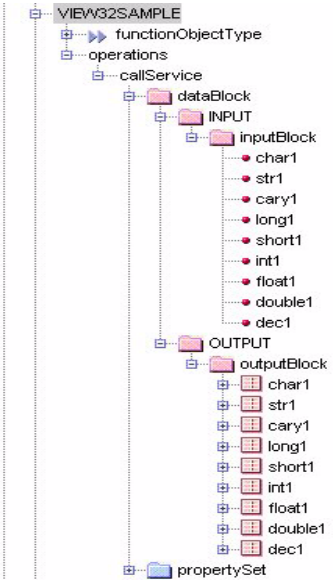
The definition of views `v32test1` and `v32test2` are as follows:

```
VIEW v32test1
# type cname   fbname  count flag  size null
char   char1   -       1     -     -    -
string str1    -       1     -     100  -
carray cary1   -       1     -     100  -
long   long1   -       1     -     -    -
short  short1  -       1     -     -    -
int    int1    -       1     -     -    -
float  float1  -       1     -     -    -
double double1 -       1     -     -    -
dec_t  dec1    -       1     -     9,2  -
END

VIEW v32test2
# type cname   fbname  count flag  size null
char   char1   -       5     -     -    -
string str1    -       5     -     100  -
carray cary1   -       5     -     100  -
long   long1   -       5     -     -    -
short  short1  -       5     -     -    -
int    int1    -       5     -     -    -
```

```
float  float1  -       5     -     -    -
double double1 -       5     -     -    -
dec_t  dec1    -       5     -     9,2  -
END
```

Figure 2-15 shows how the service is represented in the UIF repository.

Figure 2-15 VIEW32SAMPLE Function Object



The following code fragment illustrates how the Tuxedo Enterprise Connector works with a service whose buffer type is `VIEW32`:

```
try {
    fn = runtime.createFunctionObject("TEST", "VIEW32TEST");
    sp.enable();
    fn.useServiceProvider(sp);
    fn.prepare("callService");
    data = fn.getDataBlock();
// populate the input data
data.setAttrInt("INPUT.inputBlock.char1", (int) 'a');
```

```
data.setAttrString("INPUT.inputBlock.str1", "Hello World");
data.setAttrInt("INPUT.inputBlock.long1", 9876);
data.setAttrInt("INPUT.inputBlock.short1", 8888);
data.setAttrInt("INPUT.inputBlock.int1", 9999);
data.setAttrFloat("INPUT.inputBlock.float1", 2123.1212f);
data.setAttrDouble("INPUT.inputBlock.double1", 234.234);
data.setAttrVBinary("INPUT.inputBlock.cary1", "Hello
World".getBytes());
data.setAttrDouble("INPUT.inputBlock.dec1", 2123.12);
// call the service
fn.execute();
// get the output results
for (int i = 0; i < 5; i ++) {
   int outChar1 =
   data.getAttrInt("OUTPUT.outputBlock.char1" + ".["+i+"]");
   String outStr1 =
   data.getAttrString("OUTPUT.outputBlock.str1" + ".["+i+"]");
   long   outLong1 =
   data.getAttrInt("OUTPUT.outputBlock.long1" + ".["+i+"]");
   short  outShort1 = (short)
   data.getAttrInt("OUTPUT.outputBlock.short1" + ".["+i+"]");
   int    outInt1 =
   data.getAttrInt("OUTPUT.outputBlock.int1" + ".["+i+"]");
   float  outFloat1 =
   data.getAttrFloat("OUTPUT.outputBlock.float1" + ".["+i+"]");
   double outDouble1 =
   data.getAttrDouble("OUTPUT.outputBlock.double1" + ".["+i+"]");
   double outDec1 =
   data.getAttrDouble("OUTPUT.outputBlock.dec1" + ".["+i+"]");
.....
}
```

.....

X_COMMON Sample

This sample illustrates how the Tuxedo Enterprise Connector works with a service whose buffer type is X_COMMON. This sample uses a service XCOMMONSAMPLE with input and output X_COMMON buffer type. This service takes the data from the client X_COMMON buffer with subtype xcomtest1, creates a new X_COMMON buffer with subtype xcomtest2, populates the structure xcomtest2 and passes it back to the client.

Before calling this service, VIEW and service definitions must be imported into the UIF repository using the Tuxedo Management Console.

The service definition is as follows:

```
interface SAMPLE {
     void XCOMMONSAMPLE(
         [in,  X_COMMON xcomtest1]  X_COMMON inputBlock
         [out, X_COMMON xcomtest2]  X_COMMON outputBlock
     );
};
```

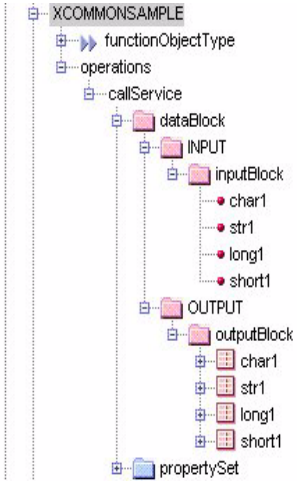
The following listing shows the VIEW field definitions used in this service:

```
#
# view def for X_COMMON with count 1
#
VIEW xcomtest1
# type cname   fbname  count flag  size null
char   char1   -       1     -     -    -
string str1    -       1     -     100  -
long   long1   -       1     -     -    -
short  short1  -       1     -     -    -
END
#
# view def for X_COMMON with count 10
```

```
#
VIEW xcomtest2
# type cname   fbname  count flag  size null
char   char1   -       10 -     -    -
string str1    -       10 -     100  -
long   long1   -       10 -     -    -
short  short1  -       10 -     -    -
END
```

Figure 2-16 shows how the service is represented in the UIF repository.

Figure 2-16 XCOMMONSAMPLE Function Object



The code to populate the input structure `xcomtest1` calls the service and gets the results from output `xcomtest2` structure as shown below.

```
fn = runtime.createFunctionObject("TEST","XCOMMONSAMPLE");
sp.enable();
fn.useServiceProvider(sp);
fn.prepare("callService");
data = fn.getDataBlock();
// populate the input data
```

```
data.setAttrInt("INPUT.inputBlock.char1", (int) 'a');
data.setAttrString("INPUT.inputBlock.str1", "Hello World");
data.setAttrInt("INPUT.inputBlock.long1", 9876);
data.setAttrInt("INPUT.inputBlock.short1", 8888);
// call the service
fn.execute();
// Get the output results
for (int i = 0; i < 10; i ++) {
   int outChar1 =
      data.getAttrInt("OUTPUT.outputBlock.char1" + ".["+i+"]");
   String outStr1 =
      data.getAttrString("OUTPUT.outputBlock.str1" + ".["+i+"]");
   long outLong1 =
      data.getAttrInt("OUTPUT.outputBlock.long1" + ".["+i+"]");
   short outShort1 = (short)
      data.getAttrInt("OUTPUT.outputBlock.short1" + ".["+i+"]");
   .....
}
....
```

X_C_TYPE Sample

This sample illustrates how the Tuxedo Enterprise Connector works with a service whose buffer type is `X_C_TYPE`. This sample uses a service `XCTYPESAMPLE` with input and output `X_C_TYPE` buffer type. This service takes the data from the client X_C_TYPE with subtype `xctest1`, creates a new `X_C_TYPE` buffer with subtype `xctest2`, populates the structure `xctest2` and passes it back to the client.

Before calling this service, VIEW and service definitions must be imported into the UIF repository using the Tuxedo Management Console.

The definition of the service is as follows:

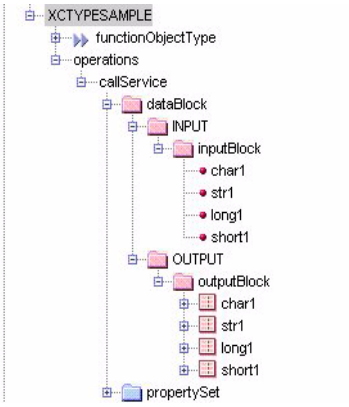
```
interface SAMPLE{
     void XCTYPESAMPLE(
         [in,  X_C_TYPE xctest1]  X_C_TYPE inputBlock
         [out, X_C_TYPE xctest2]  X_C_TYPE outputBlock
     );
};
```

This service uses the following views:

```
#
# view def for X_C_TYPE with count 1
#
VIEW xctest1
# type cname   fbname  count flag  size null
char   char1   -       1     -     -    -
string str1    -       1     -     100  -
long   long1   -       1     -     -    -
short  short1  -       1     -     -    -
END
#
# view def for X_C_TYPE with count 10
#
VIEW xctest2
# type cname   fbname  count flag  size null
char   char1   -       10     -     -    -
string str1    -       10     -     100  -
long   long1   -       10     -     -    -
short  short1  -       10     -     -    -
END
```

Figure 2-17 shows how the service is represented in the UIF repository.

Figure 2-17 XCTYPESAMPLE Function Object



The code to populate the input structure `xctest1` calls the service and gets the results from output `xctest2` structure as shown below.

```
try {
....
fn = runtime.createFunctionObject("TEST", "XCTYPESAMPLE");
sp.enable();
fn.useServiceProvider(sp);
fn.prepare("callService");
data = fn.getDataBlock();
// populate the input data
data.setAttrInt("INPUT.inputBlock.char1", (int) 'a');
data.setAttrString("INPUT.inputBlock.str1", "Hello World");
data.setAttrInt("INPUT.inputBlock.long1", 9876);
data.setAttrInt("INPUT.inputBlock.short1", 8888);
// call the service
fn.execute();
// get the output results
for (int i = 0; i < 10; i ++) {
   int outChar1 =
data.getAttrInt("OUTPUT.outputBlock.char1" + ".["+i+"]");
```

```
String outStr1 =
data.getAttrString("OUTPUT.outputBlock.str1" + ".["+i+"]");
long outLong1 =
data.getAttrInt("OUTPUT.outputBlock.long1" + ".["+i+"]");
short outShort1 = (short)
data.getAttrInt("OUTPUT.outputBlock.short1" + ".["+i+"]");
....
}
....
```

TransactionFO Function Object

The `TransactionFO` is a special function object present with each Tuxedo datasource. It is created automatically when you create a new Tuxedo datasource. This special function object provides support to develop Java programs which calls ATMI functions `tpbegin()`, `tpcommit()`, and `tpabort()` from client process.

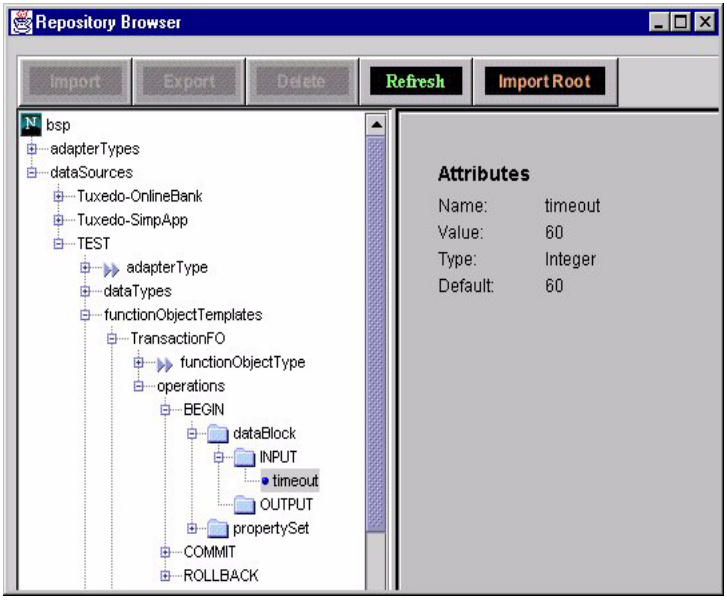
The `TransactionFO` function object supports the following operations:

- `BEGIN`
- `COMMIT`
- `ROLLBACK`

The above operations are mapped to `tpbegin()`, `tpcommit()`, and `tpabort()` of ATMI primitives. In the Tuxedo Enterprise Connector these operations are used to define client-side global transactions.

Figure 2-18 shows how the `TransactionFO` is represented in the UIF repository.

Figure 2-18 TransactionFO Function Object



The transaction is started by executing the `BEGIN` operation. Optionally, you can specify a *timeout* parameter. The timeout specifies the amount of time in seconds a transaction has before timing out. If the *timeout* parameter is not set the Tuxedo Enterprise Connector uses 60 seconds as the default value.

Developing Client-side Transactions

A transaction is bounded by a begin transaction and an end transaction. Between the begin transaction and end transaction, the application is said to be in transaction mode. This can be controlled either in Tuxedo clients or services. Client controlled transactions are known as client-side global transactions.

The following sample illustrates how to use client-side transactions in Tuxedo Enterprise Connector.

This sample uses two user-defined Tuxedo services, `WITHDRAWL` and `DEPOSIT` to perform `TRANSFER` transaction. If `WITHDRAWL` operation fails, a rollback is performed. Otherwise, a `DEPOSIT` is performed and a commit completes the transaction.

The tuxConnectionTx service provider type must be used while creating the service provider object.

The following code fragment illustrates how to use the TransactionFO function object:

```
.....


IContext            ctx = null;
IBSPRuntime         rt  = null;
IBSPServiceProvider sp  = null;
IBSPFunctionObject  fn  = null;
TuxTransaction      tx  = null;


try {
    // get runtime
    ctx =
((com.netscape.server.servlet.platformhttp.PlatformServletContext)
getServletContext()).getContext();
    rt = access_cBSPRuntime.getcBSPRuntime(ctx, null, null);
    // create service provider using tuxConnectionTx type
    sp = runtime.createServiceProvider("TEST", "tuxConnectionTx");
    sp.enable();
    // instantiate TuxTransaction
    tx = new TuxTransaction(rt, sp, "TEST");
    // begin transaction
    tx.begin();
    // call WITHDRAWL Service
    fn = rt.createFunctionObject("TEST", "WITHDRAWL");
    fn.useServiceProvider(sp);
    fn.prepare("callService");
    // set inputs
    ....
    //execute WITHDRAWL service
```

```
    fn.execute();
   // Check for errors
   if (error) {
      // call rollback
      tx.rollback();
      return;
   }
   // call DEPOSIT Service
    fn = rt.createFunctionObject("TEST", "DEPOSIT");
    fn.useServiceProvider(sp);
    fn.prepare("callService");
    // set inputs
    ....
    //execute DEPOSIT service
    fn.execute();
   // commit transaction
    tx.commit();
} catch (BspException e) {
    // handle exceptions
} finally {
    if (sp != null)
        sp.disable();
}
```

TuxTransaction Class - Abstracts TransactionFO operations

```
package TuxBank;
import netscape.bsp.*;
import netscape.bsp.runtime.*;
import netscape.bsp.dataobject.*;

public class TuxTransaction {
```

```
    IBSPFunctionObject txfn = null;


    public TuxTransaction(IBSPRuntime rt, IBSPServiceProvider sp,
String ds) throws BspException
    {
        txfn = rt.createFunctionObject(ds, "TransactionFO");
        txfn.useServiceProvider(sp);
    }
    public void begin() throws BspException
    {
        txfn.prepare("BEGIN");
        txfn.execute();
    }
    public void commit() throws BspException
    {
        txfn.prepare("COMMIT");
        txfn.execute();
    }
    public void rollback() throws BspException
    {
        txfn.prepare("ROLLBACK");
        txfn.execute();
    }
}
```

Using propertySet Parameters

`propertySet` is used to describe the operational parameters associated with a function object. The following parameters are defined:

- `serviceName`: The Tuxedo Service name to be invoked. This is same as the function object template name. It is a read-only parameter.

- `isOneWay`: If this flag is set, the enterprise connector calls the Tuxedo service without expecting a reply back from Tuxedo system. This is equivalent to setting `TPNOREPLY` parameter with function `tpacall()` in Tuxedo.

  The default value is false.

- `priority`: Sets the priority of the Tuxedo request. The priority affects how the request is dequeued by the server. The interpretation of this value is dependent on the parameter, `isPriorityRelative`.

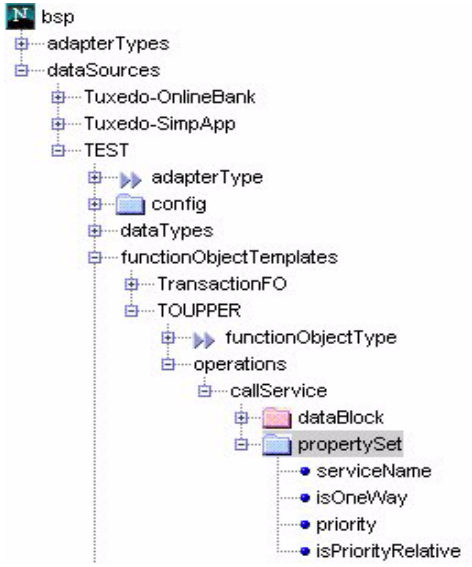
- `isPriorityRelative`: Indicates whether priority level is relative or absolute. This parameter indicates how the priority value to be interpreted.

  If the flag is set to relative, the current priority value of the service, will be incremented or decremented based on sign of the priority value set.

  Using the absolute method, you can set a request's priority to an absolute value. The absolute value of priority must be in the range of 1 to 100, 100 being the highest priority value.

Figure 2-19 shows how the propertySet is represented in the UIF repository.

Figure 2-19 Function Object propertySet



The following code fragment illustrates how to set the `priority`:

```
try {
```

```
    IBSPDataObject prop = null;
    sp = rt.createServiceProvider(inputs.strAdapterName,
inputs.strSPTemplate);
    sp.enable();
     fo = rt.createFunctionObject(inputs.strAdapterName, strFOName);
     fo.useServiceProvider(sp);
     fo.prepare(strOperName);
     data = fo.getDataBlock();
     prop = fo.getProperties();
     prop.setAttrInt("priority", 20);
     prop.setAttrInt("isPriorityRelative", 1);
     fo.execute();
     data = fo.getDataBlock();
    .....
} catch (BspException e) {
    // handle exceptions
} finally {
     if (sp != null)
        sp.disable();
}
```

Using Tuxedo User Management

If your Tuxedo system is configured with security level `TPAPPAUTH (USER_AUTH/ACL/MANDATORY_ACL)`, you must configure the `Entities` (Tuxedo authentication context) and `WebUsers` using the Tuxedo Management Console.

The application programmer provides a `WebUserId` to the Tuxedo Enterprise Connector, which determines the Tuxedo authentication context to be used to process the request. The `WebUserId` must be set with the configuration structure of service provider before enabling by calling the `enable()` method.

The following code fragment illustrates how to set `webuser-test` as the `WebUserId` before enabling the Service Provider:

```
// Create Service Provider
....
// Get Service Provider Config structure
config = (IBSPDataObjectStructure) sp.getConfig();
// Set WebUserId
if (config != null) {
   config.setAttrString("WebUserId", "webuser-test");
}
// Enable Service Provider
sp.enable();
```

Tuxedo Application Return Code

The Tuxedo application return code is an application defined value. It can be returned from a Tuxedo service to the client along with the reply, regardless of whether or not the service completed successfully. The return code is defined as an integer and is used to carry a code that means something to the client. This code is passed using the second argument, `rcode`, in `tpreturn()` function. The Tuxedo clients can access via the global variable, `tpurcode`.

The Tuxedo Enterprise Connector provides a facility to access this application return code within Java programs. This is stored with `OUTPUT.urcode` field.

The following code fragment illustrates how to get the Tuxedo application return code:

```
try {
    sp = rt.createServiceProvider(inputs.strAdapterName,
inputs.strSPTemplate);
   sp.enable();
   fo = rt.createFunctionObject(inputs.strAdapterName, strFOName);
   fo.useServiceProvider(sp);
   fo.prepare(strOperName);
   data = fo.getDataBlock();
   fo.execute();
   data = fo.getDataBlock();
```

```
    int code = data.getAttrInt("OUTPUT.urcode");
    .....
} catch (BspException e) {
    // handle exception
} finally {
    if (sp != null)
        sp.disable();
}
```

Error and Exception Handling

The Tuxedo Enterprise Connector returns connector and Tuxedo system errors as exceptions using the `netscape.bsp.BspException` class. Always enclose the application logic within a try or catch block, and attempt to deal with an exception appropriately.

The `BspException` class instance carries additional information if the exception is thrown due to a Tuxedo system error. Obtain the additional *info object* by calling `getInfo()` method on the exception object. This returns `IBSPDataObjectStructure` object with fields `errno` and `errstr`. These two fields are mapped to Tuxedo variables `tperrno` and `tpstrerror`, respectively. Then call `getAttrInt("errno")` and `getAttrString("errstr")` methods on the *info object* to get the Tuxedo system error number and the message associated with it.

The following code fragment demonstrates how to handle exceptions:

```
IBSPDataObjectStructure info = null;
.....
catch (BspException e) {
    info = e.getInfo();
    if (info != null) {
        errno = info.getAttrInt("errno");
        strError = info.getAttrString("errstr");
        ..............
    }
}
```

The `netscape.tux.TuxError` class has constants which can be used to compare with the *errno* retuned from exception object.

The following constants are available:

- `TuxError.TPEMATCH`
- `TuxError.TPEABORT`
- `TuxError.TPEBADDESC`
- `TuxError.TPEBLOCK`
- `TuxError.TPEDIAGNOSTIC`
- `TuxError.TPED_CLIENTDISCONNECTED`
- `TuxError.TPED_DOMAINUNREACHABLE`
- `TuxError.TPED_MAXVAL`
- `TuxError.TPED_MINVAL`
- `TuxError.TPED_NOCLIENT`
- `TuxError.TPED_NOUNSOLHANDLER`
- `TuxError.TPED_SVCTIMEOUT`
- `TuxError.TPED_TERM`
- `TuxError.TPEEVENT`
- `TuxError.TPEHAZARD`
- `TuxError.TPEHEURISTIC`
- `TuxError.TPEINVAL`
- `TuxError.TPEITYPE`
- `TuxError.TPELIMIT`
- `TuxError.TPEMIB`
- `TuxError.TPENOENT`
- `TuxError.TPEOS`
- `TuxError.TPEOTYPE`
- `TuxError.TPEPERM`
- `TuxError.TPEPROTO`

- `TuxError.TPERELEASE`
- `TuxError.TPERMERR`
- `TuxError.TPESVCERR`
- `TuxError.TPESVCFAIL`
- `TuxError.TPESYSTEM`
- `TuxError.TPETIME`
- `TuxError.TPETRAN`

Developing International Applications

The Enterprise Connector for Tuxedo supports developing J2EE compliant international applications using various character sets supported by the iPlanet Application Server. To use the Tuxedo connector in international mode, the iPlanet Application Server must operate in the INTERNATIONAL mode. Refer to the iPlanet Application Server documentation on how to run the iPlanet Application Server in INTERNATIONAL mode.

All connector related error messages are formatted and logged in the operating system locale and character set in which the iPlanet Application Server is running. The character set used to pass the `WebUserId` to a Servlet, JSP, or EJB must be set correctly within your application. For details on how to specify character set and write J2EE compliant international applications, refer to iPlanet Application Server documentation.

The use of the `CARRAY` Tuxedo buffer type is recommended to transmit multi-byte data. Please contact BEA System for details on support of multi-byte and double byte character data in the BEA Tuxedo software.

Tuxedo SimpApp Sample Using Servlet

The `simpapp` sample demonstrates how a Servlet may connect to the Tuxedo system and call one of its services using the Tuxedo Enterprise Connector. In this sample the service is called `TOUPPER`, available in Tuxedo `simpapp` example. You may find the Tuxedo server *simpapp* application is created in the *<tuxedo root dir>/apps/simpapp* directory. The Java source code is available in *<iAS root dir>/ias/APPS/TuxSimpApp* directory.

The TuxSimpApp sample directory contains:

File Name	Description
SimpAppServlet.java	Servlet source code that calls the TOUPPER service
SimpAppServlet.class	Servlet Class file
TuxSimpApp.xml	Deployment descriptor XML file
ias-TuxSimpApp.xml	Deployment descriptor XML file
application.xml	Deployment descriptor XML file
createear.sh	Sample script to create ear file
tuxsimpapp.war	Web Application module
TuxSimpApp.ear	Application **E**nterprise **Ar**chive (.ear) file

Tuxedo Online Bank Sample Using JSP & EJB

The bankapp sample application illustrates the Tuxedo connectivity of the Tuxedo Enterprise Connector to the bankapp application that comes with BEA Tuxedo system. The source code for the Tuxedo server bankapp application is located in the *<tuxedo root dir>/apps/bankapp* directory. The Tuxedo server bankapp application contains the services WITHDRAWAL, DEPOSIT, TRANSFER, INQUIRY, CLOSE_ACCT, and OPEN_ACCT.

The bankapp sample application demonstrates techniques to use Servlet, EJB, and JSP J2EE components to develop a web-based application which accesses the Tuxedo services. The source code for this sample Java application is available in the *<iAS root dir>/ias/APPS/TuxBank* directory.

The TuxBank example directory contains:

File Name	Description
IBankManager.java	Remote interface for BankManager Session Enterprise Java Bean
IBankManagerHome.java	Home interface for BankManager Session Enterprise Java Bean
BankManagerBean.java	Enterprise-bean class for BankManager Session Enterprise Java Bean
BankServlet.java	Servlet that calls the BankManager EJB which in turn calls Tuxedo Services.

`TuxTransaction.java`	Abstracts TransactionFO function object to perform client-side transactions
`AccountData.java`	Application specific data class
`ApplicationException.java`	Application defined exception class
`ErrorHandler.java`	Util class to handle errors
`BankManagerUtil.java`	Application util class
`*.class files`	Java Class files
`jsp/MainMenu.jsp`	Displays BankApp main menu
`jsp/AccountOpenForm.jsp`	Account Open input form
`jsp/AccountCloseForm.jsp`	Account Close input form
`jsp/BalanceQueryForm.jsp`	Balance Query input form
`jsp/DepositAmountForm.jsp`	Deposit input form
`jsp/TransferAmountForm.jsp`	Transfer input form
`jsp/WithdrawAmountForm.jsp`	Withdraw input form
`jsp/ShowAccountOpen.jsp`	Response form for Account Open operation
`jsp/ShowAccountClose.jsp`	Response form for Account Close operation
`jsp/ShowBalance.jsp`	Response form for Balance Enquiry operation
`jsp/ShowDeposit.jsp`	Response form for Deposit operation
`jsp/ShowTransfer.jsp`	Response form for Transfer operation
`jsp/ShowWithdrawl.jsp`	Response form for Withdraw operation
`TuxBank.xml`	Deployment descriptor XML file
`ias-TuxBank.xml`	Deployment descriptor XML file
`TuxBankEjb.xml`	Deployment descriptor XML file
`ias-TuxBankEjb.xml`	Deployment descriptor XML file
`application.xml`	Deployment descriptor XML file
`createear.sh`	Sample script to create ear file
`TuxBank.war`	Web Application module
`TuxBank.ear`	Application **E**nterprise **Ar**chive (`.ear`) file

`XXXForm.jsp` displays a form for input. When the user submits the form, it calls the Servlet `BankServlet`, which calls the EJB `BankManager` and forwards the request to the Tuxedo system using the UIF API. The response is taken and sent back to the Servlet which calls a JSP `ShowXXX.jsp` to display the results. In case of an error, `ErrorPage.jsp` is shown with the appropriate error message.

Index

F

G

H

I

J

L

M

N

O

W

X