

**Oracle® Tuxedo**  
Using Oracle Jolt  
12c Release 2 (12.2.2)

April 2016

Copyright © 1996, 2016, Oracle and/or its affiliates. All rights reserved.

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

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

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

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

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

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

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

# Contents

## 1. Introducing Oracle Jolt

Oracle Jolt Components .....	1-2
Key Features .....	1-3
How Oracle Jolt Works .....	1-5
Jolt Servers and Repository Servers .....	1-6
Jolt Class Library .....	1-7
JoltBeans .....	1-9
Jolt Server and Jolt Client Communication .....	1-9
Oracle Tuxedo Service Metadata Repository .....	1-10
Jolt Internet Relay .....	1-11
Creating a Jolt Client to Access Oracle Tuxedo Applications .....	1-11

## 2. Bulk Loading Oracle Tuxedo Services

Using the Bulk Loader .....	2-1
Activating the Bulk Loader .....	2-2
The Bulk Load File .....	2-2
Syntax of the Bulk Loader Data Files .....	2-3
Guidelines for Using Keywords .....	2-3
Keyword Order in the Bulk Loader Data File .....	2-4
Using Service-Level Keywords and Values .....	2-5
Using Parameter-Level Keywords and Values .....	2-6
Troubleshooting .....	2-8

Sample Bulk Load Data . . . . .	2-9
---------------------------------	-----

## 3. Configuring the Oracle Jolt System

Quick Configuration . . . . .	3-1
Editing the UBBCONFIG File . . . . .	3-2
Configuring the Tuxedo Service Metadata Repository . . . . .	3-3
Initializing Services That Use Oracle Tuxedo and the Repository Editor . . . . .	3-3
Configuring the Oracle Tuxedo TMUSREVT Server for Event Subscription . . . . .	3-3
Configuring Jolt Relay . . . . .	3-4
Jolt Background Information . . . . .	3-5
Jolt Server . . . . .	3-5
Starting the JSL . . . . .	3-6
Shutting Down the JSL . . . . .	3-6
Restarting the JSL . . . . .	3-7
Configuring the JSL . . . . .	3-7
JSL Command-line Options . . . . .	3-7
Security and Encryption . . . . .	3-12
Jolt Relay . . . . .	3-12
Jolt Relay Failover . . . . .	3-14
Jolt Relay Process . . . . .	3-14
JRLY Command-line Options for Windows 2003 . . . . .	3-15
JRLY Command-line Option for UNIX . . . . .	3-18
JRLY Configuration File . . . . .	3-18
Jolt Relay Adapter . . . . .	3-20
JRAD Configuration . . . . .	3-20
Network Address Configurations . . . . .	3-23
Oracle Tuxedo Service Metadata Repository . . . . .	3-23
Initializing Services By Using Oracle Tuxedo and the Repository Editor . . . . .	3-23

Event Subscription . . . . .	3-24
Configuring for Event Subscription . . . . .	3-24
Filtering Oracle Tuxedo FML or VIEW Buffers. . . . .	3-25
Oracle Tuxedo Background Information . . . . .	3-26
Configuration File . . . . .	3-26
Creating the UBBCONFIG File . . . . .	3-27
Sample Applications in Oracle Jolt Online Resources. . . . .	3-36

## 4. Using the Jolt Class Library

Class Library Functionality Overview . . . . .	4-2
Java Applications Versus Java Applets . . . . .	4-2
Jolt Class Library Features . . . . .	4-3
Error and Exception Handling . . . . .	4-3
Jolt Client/Server Relationship . . . . .	4-4
Jolt Object Relationships . . . . .	4-7
Jolt Class Library Walkthrough . . . . .	4-8
Logon and Logoff . . . . .	4-8
Synchronous Service Calling. . . . .	4-8
Transaction Begin, Commit, and Rollback . . . . .	4-9
Using Oracle Tuxedo Buffer Types with Jolt. . . . .	4-14
Using the STRING Buffer Type . . . . .	4-15
Using the CARRAY Buffer Type . . . . .	4-20
Using the FML Buffer Type . . . . .	4-22
Using the VIEW Buffer Type . . . . .	4-26
Using the XML Buffer Type . . . . .	4-30
Using the MBSTRING Buffer Type . . . . .	4-33
Multithreaded Applications . . . . .	4-35
Threads of Control. . . . .	4-35

Using Jolt with Non-Preemptive Threading . . . . .	4-36
Using Threads for Asynchronous Behavior . . . . .	4-36
Using Threads with Jolt . . . . .	4-37
Event Subscription and Notifications. . . . .	4-41
Event Subscription Classes . . . . .	4-41
Notification Event Handler . . . . .	4-42
Connection Modes . . . . .	4-43
Notification Data Buffers . . . . .	4-43
Oracle Tuxedo Event Subscription . . . . .	4-44
Using the Jolt API to Receive Oracle Tuxedo Notifications . . . . .	4-45
Clearing Parameter Values. . . . .	4-46
Reusing Objects . . . . .	4-49
Deploying and Localizing Jolt Applets . . . . .	4-54
Deploying a Jolt Applet . . . . .	4-54
Client Considerations . . . . .	4-54
Web Server Considerations . . . . .	4-55
Localizing a Jolt Applet . . . . .	4-55
Using SSL . . . . .	4-56

## 5. Using JoltBeans

Overview of Jolt Beans . . . . .	5-2
JoltBeans Terms . . . . .	5-3
Adding JoltBeans to Your Java Development Environment. . . . .	5-4
Using Development and Run-time JoltBeans . . . . .	5-4
Basic Steps for Using JoltBeans. . . . .	5-5
JavaBeans Events and Oracle Tuxedo Events . . . . .	5-5
Using Oracle Tuxedo Event Subscription and Notification with JoltBeans . . . . .	5-6
How JoltBeans Use JavaBeans Events. . . . .	5-7

The JoltBeans Toolkit . . . . .	5-8
JoltSessionBean . . . . .	5-8
JoltServiceBean . . . . .	5-9
JoltUserEventBean . . . . .	5-10
Jolt-Aware GUI Beans . . . . .	5-10
JoltTextField . . . . .	5-11
JoltLabel . . . . .	5-11
JoltList . . . . .	5-11
JoltCheckbox . . . . .	5-12
JoltChoice . . . . .	5-12
Using the Property List and the Property Editor to Modify the JoltBeans Properties .	5-12
JoltBeans Class Library Walkthrough . . . . .	5-15
Building the Sample Form . . . . .	5-16
Wiring the JoltBeans Together . . . . .	5-23
Using the Oracle Tuxedo Service Metadata Repository and Setting the Property Values . .	5-41
JoltBeans Programming Tasks . . . . .	5-44
Using Transactions with JoltBeans . . . . .	5-45
Using Custom GUI Elements with the JoltService Bean . . . . .	5-46

## 6. Using Servlet Connectivity for Oracle Tuxedo

What Is a Servlet? . . . . .	6-2
How Servlets Work with Jolt . . . . .	6-2
The Jolt Servlet Connectivity Classes . . . . .	6-2
Writing and Registering HTTP Servlets . . . . .	6-3
Jolt Servlet Connectivity Sample . . . . .	6-5
Viewing the Sample Servlet Applications . . . . .	6-5
SimpApp Sample . . . . .	6-5

BankApp Sample .....	6-8
Admin Sample.....	6-10
Additional Information on Servlets .....	6-11

## 7. Migrating from Jolt Repository to Oracle Tuxedo Service Metadata Repository

Replacing JREPSVR with TMMETADATA in UBBCONFIG .....	7-2
Loading Jolt Repository to Oracle Tuxedo Service Metadata Repository.....	7-2
Sample: joltapp Migration .....	7-3

### A. Oracle Jolt Exceptions

# Introducing Oracle Jolt

Oracle Jolt is a Java-based interface to the Oracle Tuxedo system that extends the functionality of existing Oracle Tuxedo applications to include Intranet- and Internet-wide availability. Using Jolt, you can now easily transform any Oracle Tuxedo application so that its services are available to customers using an ordinary browser on the Internet. Jolt interfaces with existing and new Oracle Tuxedo applications and services to allow secure, scalable, intranet/Internet transactions between client and server. Jolt enables you to build client applications and applets that can remotely invoke existing Oracle Tuxedo services, such as application messaging, component management, and distributed transaction processing.

Because you develop your applications with the Jolt API, which use Oracle Tuxedo and the Java programming language, the Jolt documentation is written with the assumption that you are familiar with Oracle Tuxedo and Java programming. This documentation is intended for system administrators, network administrators, and developers.

This topic includes the following sections:

- [Oracle Jolt Components](#)
- [Key Features](#)
- [How Oracle Jolt Works](#)
- [Creating a Jolt Client to Access Oracle Tuxedo Applications](#)

# Oracle Jolt Components

Oracle Jolt is a Java class library and API that provides an interface to Oracle Tuxedo from remote Java clients. Oracle Jolt consists of the following components for creating Java-based client programs that access Oracle Tuxedo services:

- **Jolt Servers and Repository Servers**—one or more Jolt servers listen for network connections from clients, translate Jolt messages, multiplex multiple clients into a single process, and submit and retrieve requests to and from Oracle Tuxedo-based applications running on one or more Oracle Tuxedo servers.
- **Jolt Class Library**—the Jolt class library is a Java package containing the class files that implement the Jolt API. These classes enable Java applications and applets to invoke Oracle Tuxedo services. The Jolt class library includes functionality to set, retrieve, manage, and invoke communication attributes, notifications, network connections, transactions, and services.
- **JoltBeans**—Oracle JoltBeans provides a JavaBeans-compliant interface to Oracle Jolt. JoltBeans are Beans components that you can use in JavaBeans-enabled integrated development environments (IDEs) to construct Oracle Jolt clients. Jolt Beans consists of two sets of Java Beans: JoltBeans toolkit (a JavaBeans-compliant interface to Oracle Jolt that includes the JoltServiceBean, JoltSessionBean, and JoltUserEventBean) and Jolt GUI beans, which consist of Jolt-aware Abstract Window Toolkit (AWT) and Swing-based beans.
- **Jolt Internet Relay**—the Jolt Internet Relay is a component that routes messages from a Jolt client to a Jolt Server Listener (JSL) or Jolt Server Handler (JSH). This component eliminates the need for the JSH and Oracle Tuxedo to run on the same machine as the Web server. The Jolt Internet Relay consists of the Jolt Relay (JRLY) and the Jolt Relay Adapter (JRAD).
- **Jolt ECID** —The Jolt call process is as follows: `JOLT client --> JSL/JSH --> tuxedo server --> service`

1. Jolt Connection Pool supports ECID propagation between Weblogic and Oracle Tuxedo. ECID is propagated and inserted into a request message if java option

"-Dtuxedo.ECID\_ENABLE" or environment variable "ECID\_ENABLE" is set to 'y'. Java option "-Dtuxedo.ECID\_ENABLE" has higher priority than environment variable "ECID\_ENABLE". ECID also can be sent back within a reply message from Oracle Tuxedo.

2. ECID relies on DMS (Dynamic Monitoring Service), ECID is not supported for standalone jolt client by default. If you want ECID to be created, DMS should be installed first.

3. Jolt ECID does not impact JRLY and JRAD. The format for ECID in jolt client trace is as follows:

```
000915:30475@s1c05are:1: ECID <0000KqPMc659XbHpIsT4if1LPfwR000001>:
atmi: { JoltSession.send(len 132)
```

## Key Features

With Oracle Jolt, you can leverage existing Oracle Tuxedo services and extend your transaction environment to the corporate intranet or world-wide Internet. The key feature of Jolt architecture is its simplicity. You can build, deploy, and maintain robust, modular, and scalable electronic commerce systems that operate over the Internet.

Oracle Jolt includes the following features:

- **Java-based API for simplified development**—with its Java-based API, Oracle Jolt simplifies application design by providing well-designed object interfaces. Jolt supports the Java 2 Software Development Kit (SDK) and is fully compatible with Java threads. Jolt enables Java programmers to build graphical front-ends that use the Oracle Tuxedo application and transaction services without having to understand detailed transactional semantics or rewrite existing Oracle Tuxedo applications.
- **Pure Java client development**—using Jolt, you can build a pure Java client that runs in any Java-enabled browser. Jolt automatically converts from Java to native Oracle Tuxedo data types and buffers, and from Oracle Tuxedo back to Java. As a pure Java client, your applet or application does not need resident client-side libraries or installation; thus, you can download client applications from the network.
- **Easy access to Oracle Tuxedo services through Oracle Tuxedo Service Metadata Repository**—the Oracle Tuxedo Service Metadata Repository facilitates Java application development by managing and presenting Oracle Tuxedo service definitions that you can use in your Java client. A bulk loading utility lets you quickly integrate your existing Oracle Tuxedo services into the Jolt development environment. Jolt and Oracle Tuxedo simplify network and application scalability, while encouraging the reuse of application components. You can also use `tmloadrepos` to create or update the binary Tuxedo Service Metadata Repository file and load it with service parameter information. See [tmloadrepos\(1\)](#) for more information.
- **GUI-Based maintenance and distribution of Oracle Tuxedo services**—the GUI lets you manage Oracle Tuxedo service definitions such as service names, inputs and outputs. The GUI provides support for different input and output names for services defined in the Metadata Repository.

- **Encryption for secure transaction processing**—Oracle Jolt allows you to encrypt data transmitted between Jolt clients and the JSL/JSH. Jolt encryption helps ensure secure Internet transaction processing.
- **Added security through Internet Relay**—network administrators can use the Oracle Jolt Internet Relay component to separate their Web server and Oracle Tuxedo application server. Web servers are generally considered insecure because they often exist outside a corporate firewall. Using the Jolt Internet Relay, you can locate your Oracle Tuxedo server in a secure location or environment on your network, yet still handle transactions from Jolt clients on the Internet.
- **Event Subscription Support**—Jolt Event Subscription enables you to receive event notifications from Oracle Tuxedo services and Oracle Tuxedo clients. Jolt Event Subscription lets you subscribe to two types of Oracle Tuxedo application events:
  - Unsolicited Event Notifications—a Jolt client can receive these notifications when an Oracle Tuxedo client or service subscribes to unsolicited events and an Oracle Tuxedo client issues a broadcast or a directly targeted message.
  - Brokered Event Notifications—the Jolt client receives these notifications through the Oracle Tuxedo Event Broker. The Jolt client receives these notifications only when it subscribes to an event and any Oracle Tuxedo client or server posts an event.
- **Jolt Trace**—If java option `-Dtuxedo.TMTRACE` is set, JOLT client prints out trace as the trace format. JOLT trace format is the same as Oracle Tuxedo. Four trace categories, "atmi", "inet", "trace" and "\*" are supported.
  - **atmi**  
Trace points for explicit application calls to the ATMI and TX interfaces.
  - **inet**  
Trace points related to network.
  - **trace**  
Trace points related to the tracing feature itself, including message dyeing.
  - **\***  
All trace points.

**Listing 1-1 Jolt Trace Category Example**

---

```
-Dtuxedo.TMTRACE=trace_spec  atmi/inet:jtrace
-Dtuxedo.JTRACEPATH=path_to_trace_file  (optional, if not set, use
user.dir)
-Dtuxedo.ECID_ENABLE=y|n (optional)
```

Usage:

```
atmi+inet+trace:jtrace:dye
atmi+inet:jtrace:dye
*:jtrace:dye
atmi-inet:jtrace:undy
```

Output file name:

```
Jtrace.yyyymmdd
```

---

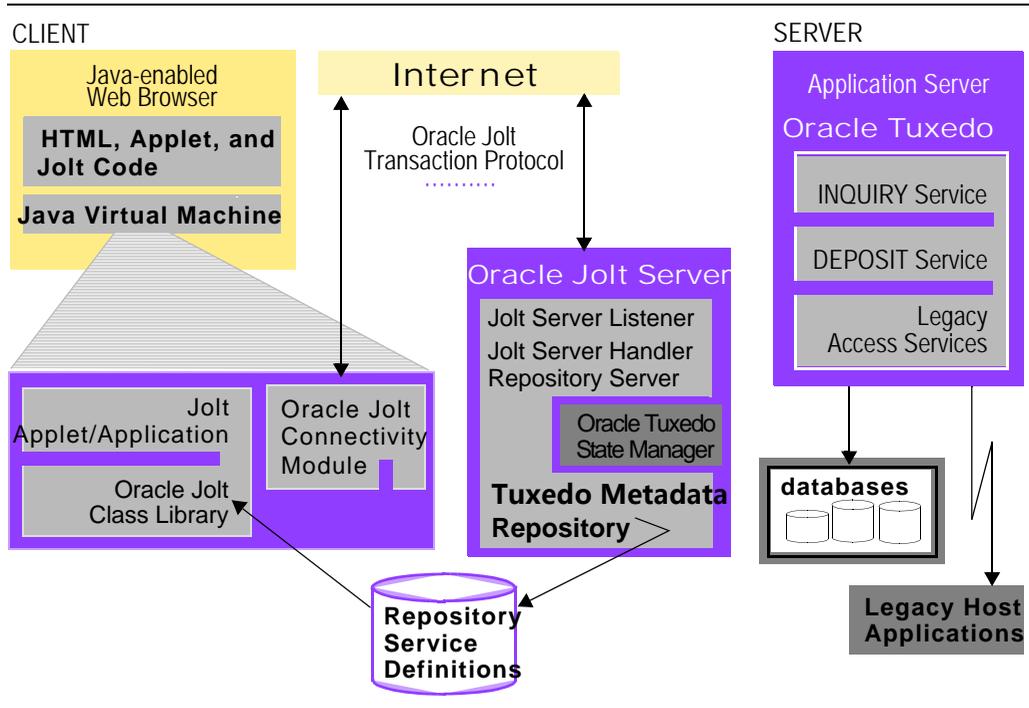
## How Oracle Jolt Works

Oracle Jolt connects Java clients to applications that are built using the Oracle Tuxedo system. The Oracle Tuxedo system provides a set of modular services, each offering specific functionality related to the application as a whole.

The end-to-end view of the Oracle Jolt architecture, as well as related Oracle Tuxedo components and their interactions, is illustrated in the figure [“Oracle Jolt Architecture” on page 1-6](#).

Using [Figure 1-1](#) as an example, a simple banking application might have services such as INQUIRY, WITHDRAW, TRANSFER, and DEPOSIT. Typically, service requests are implemented in C or COBOL as a sequence of calls to a program library. Accessing a library from a native program means installing the library for the specific combination of CPU and operating system release on the client machine, a situation that Java was expressly designed to avoid. The Jolt Server implementation acts as a proxy for the Jolt client, invoking the Oracle Tuxedo service on behalf of the client. The Oracle Jolt Server accepts requests from the Jolt clients and maps those requests into Oracle Tuxedo service requests.

**Figure 1-1 Oracle Jolt Architecture**



## Jolt Servers and Repository Servers

### Jolt Servers

The following Jolt Server components act in concert to pass Jolt client transaction processing requests to the Oracle Tuxedo application.

- **Jolt Server Listener (JSL)**

The JSL handles the initial Jolt client connection, and assigns a Jolt client to the Jolt Server Handler.

- **Jolt Server Handler (JSH)**

The JSH manages network connectivity, executes service requests on behalf of the client and translates Oracle Tuxedo buffer data into the Jolt buffer, as well as Jolt buffer data into the Tuxedo buffer.

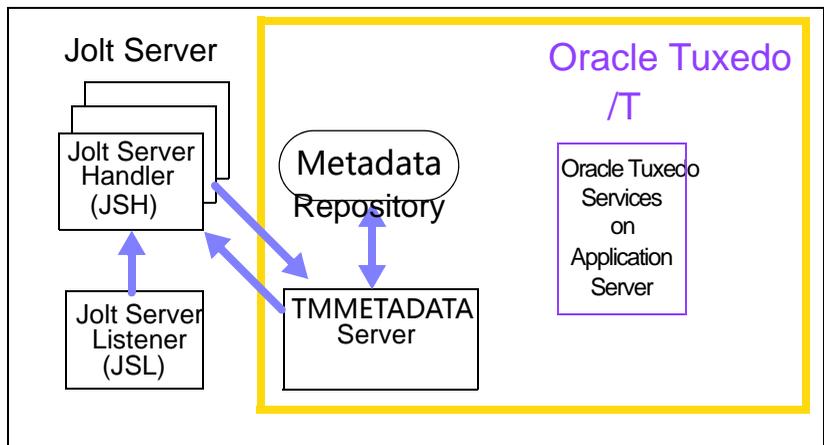
## Repository Servers

- **TMMETADATA Server**

TMMETADATA server retrieves Jolt service definitions from the Tuxedo Service Metadata Repository and returns the service definitions to the JSH. The TMMETADATA server also updates or adds Jolt service definitions.

Figure 1-2 illustrates the Jolt-related server and repository components.

**Figure 1-2 Jolt-Related Server and Repository Components**



## Jolt Class Library

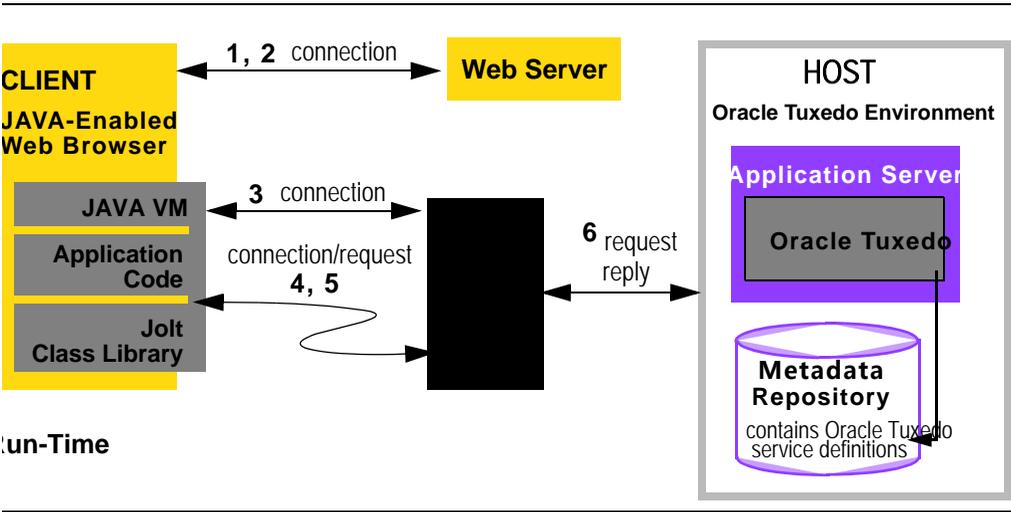
The Oracle Jolt Class Library is a set of classes that you can use in your Java application or applet to make service requests to the Oracle Tuxedo system from a Java-enabled client. You access Oracle Tuxedo transaction services by using Jolt class objects.

When developing a Jolt client application, you only need to know about the classes that Jolt provides and the Oracle Tuxedo services that are exported by the Metadata Repository. Jolt hides the underlying application details. To use Jolt and the Jolt Class Library, you do **not** need to understand: the underlying transactional semantics, the language in which the services were coded, buffer manipulation, the location of services, or the names of databases used.

The Jolt API is a Java class library and has the benefits that Java provides: applets are downloaded dynamically and are only resident during run time. As a result, there is no need for client installation, administration, management, or version control. If services are changed, the client application notes the changes at the next call to the Metadata Repository.

Figure 1-3 shows the flow of activity from a Jolt client to and from the Oracle Tuxedo system. The call-out numbers correspond to descriptions of the activity in the table “Using the Jolt Class Library” on page 1-8.

Figure 1-3 Using the Jolt Class Library to Access Oracle Tuxedo Services



Run-Time

Table 1-1 briefly describes the flow of activity involved in using the Jolt Class Library to access Oracle Tuxedo services, as shown in the previous figure “Using the Jolt Class Library to Access Oracle Tuxedo Services.”

Table 1-1 Using the Jolt Class Library

Process	Step	Action
Connection	1	A Java-enabled Web browser uses HTTP protocol to download an HTML page.
	2	A Jolt applet is downloaded and executed in the Java Virtual Machine on the client.
Request	3	The first Java applet task is to open a separate connection to the Jolt Server.
	4	The Jolt client now knows the signature of the service (such as name, parameters, types); can build a service request object based on Jolt class definitions, and make a method call.

**Table 1-1 Using the Jolt Class Library (Continued)**

...	5	The request is sent to the Jolt Server, which translates the Java-based request into an Oracle Tuxedo request and forwards the request to the Oracle Tuxedo environment.
<b>Reply</b>	6	The Oracle Tuxedo system processes the request and returns the information to the Jolt Server, which translates it back to the Java applet.

## JoltBeans

Oracle Jolt now includes JoltBeans, Java beans components that you use in a Java-enabled integrated development environment (IDE) to construct Oracle Jolt clients. Using JoltBeans, and popular JavaBeans-enabled development tools such as Symantec Visual Café, you can graphically create client applications.

Oracle JoltBeans provide a JavaBeans-compliant interface to Oracle Jolt that enables you to develop a fully functional Oracle Jolt client without writing any code. You can drag and drop JoltBeans from the component palette of a development tool and position them on the Java form (or forms) of the Jolt client application you are creating. You can populate the properties of the beans and graphically establish event source-listener relationships between various beans of the application or applet. Typically, the development tool is used to generate the event hook-up code, or you can code the hook-up manually. Client development with JoltBeans is integrated with the Oracle Tuxedo Service Metadata Repository, which provides easy access to available Oracle Tuxedo functions.

## Jolt Server and Jolt Client Communication

The Jolt system handles all communication between the Jolt Server and the Jolt client using the Oracle Jolt Protocol. The communication process between the Jolt Server and the Jolt client applet or applications functions as follows:

1. Oracle Tuxedo service requests and associated parameters are packaged into a message buffer and delivered over the network to the Jolt Server.
2. The Jolt Server unpacks the data from the message and performs necessary data conversions, such as numeric format conversions or character set conversions.
3. The Jolt Server makes the appropriate service request to the application service requested by the Jolt client.

4. Once a service request enters the Oracle Tuxedo system, it is executed in exactly the same manner as requests issued by any other Oracle Tuxedo client.
5. The results are then returned to the Oracle Jolt Server, which packages the results and any error information into a message that is sent to the Jolt client.
6. The Jolt client then maps the contents of the message into the various Jolt client interface objects, completing the request.

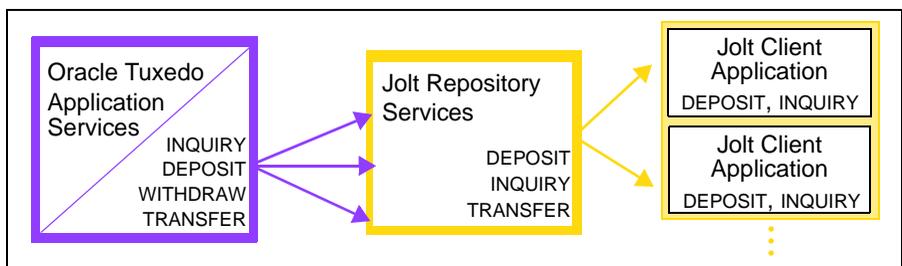
## Oracle Tuxedo Service Metadata Repository

Jolt uses the Oracle Tuxedo Service Metadata Repository as the database where Oracle Tuxedo services are defined, such as name, number, type, parameter size, and permissions. The repository functions as a central database of definitions for Oracle Tuxedo services and permits new and existing Oracle Tuxedo services to be made available to Jolt client applications. An Oracle Tuxedo application can have many services or service definitions, such as `ADD_CUSTOMER`, `GET_ACCOUNTBALANCE`, `CHANGE_LOCATION`, and `GET_STATUS`. All or only a few of these definitions can be exported to the Metadata Repository. Within the Metadata Repository, the developer or system administrator uses the Metadata Editor to export these services to the Jolt client application. The original Jolt Repository is deprecated now and all service definitions that it stores can be loaded to Oracle Tuxedo Service Metadata Repository. For more information, see [Migrating from Jolt Repository to Oracle Tuxedo Service Metadata Repository](#).

All Repository services that are exported to one client are exported to all clients. Oracle Tuxedo handles the cases where subsets of services may be needed for one client and not others.

[Figure 1-4](#) illustrates how the Metadata Repository brokers Oracle Tuxedo services to multiple Jolt client applications. (Four Oracle Tuxedo services are shown; however, the `WITHDRAW` service is not defined in the repository and the `TRANSFER` service is defined but not exported.)

**Figure 1-4 Distributing Oracle Tuxedo Services Through Jolt**



## Jolt Internet Relay

The Jolt Internet Relay is a component that routes messages from a Jolt client to the Jolt Server. The Jolt Internet Relay consists of the **Jolt Relay (JRLY)** and the **Jolt Relay Adapter (JRAD)**. JRLY is a stand-alone software component that routes Jolt messages to the Jolt Relay Adapter. Requiring only minimal configuration to work with Jolt clients, the Jolt Relay eliminates the need for the Oracle Tuxedo system to run on the same machine as the Web server.

The JRAD is an Oracle Tuxedo system server, but does not include any Oracle Tuxedo services. It requires command-line arguments to allow it to work with the JSH and the Oracle Tuxedo system. JRAD receives client requests from JRLY, and forwards the request to the appropriate JSH. Replies from the JSH are forwarded back to the JRAD, which sends the response back to the JRLY. A single Jolt Internet Relay (JRLY/JRAD pair) handles multiple clients concurrently.

## Creating a Jolt Client to Access Oracle Tuxedo Applications

The main steps for creating and deploying a Jolt client, are described in the following procedure and in the figure “[Creating a Jolt Application](#)” on page 1-13.

1. Make sure you have created an Oracle Tuxedo system application.

For information about installing Oracle Tuxedo and creating an Oracle Tuxedo application, refer to *Installing the Oracle Tuxedo System* and *Setting Up an Oracle Tuxedo Application*.

2. Install the Jolt system.

Refer to *Installing the Oracle Tuxedo System*.

3. Use the Bulk Loader utility to load Tuxedo services into the Jolt Repository Database.

For information on using this utility, see “Bulk Loading Tuxedo Services.”

[Table 1-2](#) shows the mapping relationship between the Jolt repository data types and Tuxedo services data types.

**Table 1-2 Mapping Relationship Between Jolt Repository Database and Tuxedo Services Data Types**

Tuxedo Data Type	Jolt Repository Data Type
char (byte)	byte
short	short

**Table 1-2 Mapping Relationship Between Jolt Repository Database and Tuxedo Services Data Types**

<b>Tuxedo Data Type</b>	<b>Jolt Repository Data Type</b>
int	integer
float	float
double	double
bool	boolean
long long	long
struct nestedstructname	view32
string	string
string	carray

**Note:** The long type size of Tuxedo service can be 32 bits or 64 bits. If the Tuxedo service uses 32-bit long type, map it to `integer` in Jolt Client. Otherwise, if the Tuxedo service uses 64-bit long type, map it to `long` in Jolt Client.

Since Java does not support unsigned data type, to use the unsigned data types in Tuxedo services, you need to map them to Jolt data types, which have the same storage space. For example, "unsigned int" should be mapped to "integer" in Jolt client. Besides, if you want to show the exact value of unsigned data type in Jolt client, you need to do additional casting works in Java.

4. Create a client application by using the Jolt Class Library.

The following documentation shows you how to program your client application using the Jolt Class Library:

- [Using the Jolt Class Library](#)
- Oracle Jolt API Reference

5. Run the Jolt-based client applet or application.

Figure 1-5 Creating a Jolt Application





# Bulk Loading Oracle Tuxedo Services

As a systems administrator, you may have an existing Oracle Tuxedo application with multiple Oracle Tuxedo services. Manually creating these definitions in the repository database may take hours to complete. The Bulk Loader is a command utility that allows you to load multiple, previously defined Oracle Tuxedo services to the Oracle Tuxedo Service Metadata Repository database in a single step. Using the `jblld` program, the Bulk Loader utility reads the Oracle Tuxedo service definitions from the specified text file and bulk loads them into the Metadata Repository. The services are loaded to the repository database in one “bulk load.” After the services populate the Metadata Repository, you can create, edit, and group services with the Metadata Editor.

This topic includes the following sections:

- [Using the Bulk Loader](#)
- [Syntax of the Bulk Loader Data Files](#)
- [Troubleshooting](#)
- [Sample Bulk Load Data](#)

## Using the Bulk Loader

The `jblld` program is a Java application. Before running the `jblld` command, set the `CLASSPATH` environment variable (or its equivalent) to point to the directory where the Jolt class directory (that is, `jolt.jar` and `joltadmin.jar`) is located. If the `CLASSPATH` variable is not set, the Java Virtual Machine (JVM) cannot locate any Jolt classes.

For security reasons, `jbld` does not use command-line arguments to specify user authentication information (user password or application password). Depending on the server's security level, `jbld` automatically prompts the user for passwords.

The Bulk Loader utility gets its input from command-line arguments and from the input file.

## Activating the Bulk Loader

1. Type the following at the prompt (with the correct options):

```
java bea.jolt.admin.jbld [-n] [-p package] [-u usrname] [-r usrrole]  
//host:port filename
```

2. Use [Table 2-1](#) to correctly specify the command-line options.

## Command-line Options

**Table 2-1 Bulk Loader Command-line Options**

Option	Description
<code>-u <i>usrname</i></code>	Specifies the username (default is your account name). (Mandatory if required by security.)
<code>-r <i>usrrole</i></code>	Specifies the user role (default is <code>admin</code> ). (Mandatory if required by security.)
<code>-n</code>	Validates input file against the current repository; no updates are made to the repository. (Optional)
<code>//<i>host:port</i></code>	Specifies the JRLY or JSL address (host name and IP port number). (Mandatory)
<code><i>filename</i></code>	Specifies the file containing the service definitions. (Mandatory)

## The Bulk Load File

The bulk load file is a text file that defines services and their associated parameters. The Bulk Loader loads the services defined in the bulk loader file into the Metadata Repository using the package name “BULKPKG” by default.

If a service exists in a package other than the package you name that uses the `-p` option, the Bulk Loader reports the conflict and does not load a service from the bulk loader file into the

repository. Use the Repository Editor to remove duplicate services and load the bulk loader file again.

## Syntax of the Bulk Loader Data Files

Each service definition consists of service properties and parameters that have a set number of parameter properties. Each property is represented by a keyword and a value.

Keywords are divided into two levels:

- Service-level
- Parameter-level

## Guidelines for Using Keywords

The `jbld` program reads the service definitions from a text file. To use the keywords, observe the guidelines in [Table 2-2](#).

**Table 2-2 Guidelines for Using Keywords**

Guideline	Example
Each keyword must be followed by an equal sign (=) and the value.	<b>Correct:</b> <code>type=string</code> <b>Incorrect:</b> <code>type</code>
Only one keyword is allowed on each line.	<b>Correct:</b> <code>type=string</code> <b>Incorrect:</b> <code>type=string access=out</code>
Any lines not having an equal sign (=) are ignored.	<b>Correct:</b> <code>type=string</code> <b>Incorrect:</b> <code>type string</code>
Certain keywords only accept a well-defined set of values.	The keyword <b>access</b> accepts only these values: <b>in, out, inout, noaccess</b>

**Table 2-2 Guidelines for Using Keywords (Continued)**

<b>Guideline</b>	<b>Example</b>
The input file can contain multiple service definitions.	<pre>service=INQUIRY &lt;service keywords and values&gt; service=DEPOSIT &lt;service keywords and values&gt; service=WITHDRAWAL &lt;service keywords and values&gt; service=TRANSFER &lt;service keywords and values&gt;</pre>
Each service definition consists of multiple keywords and values.	<pre>service=DEPOSIT export=true inbuf=VIEW32 outbuf=VIEW32 inview=INVIEW outview=OUTVIEW</pre>

## Keyword Order in the Bulk Loader Data File

Keyword order must be maintained within the data files to ensure an error-free transfer during the bulk load.

The first keyword definition in the bulk loader data text file must be the initial `service=<NAME>` keyword definition (shown in the listing “Keyword Hierarchical Order in a Data File”).

Following the `service=<NAME>` keyword, all remaining service keywords that apply to the named service must be specified before the first `param=<NAME>` definition. These remaining service keywords can be in any order.

All parameters associated with the service must be specified. Following each `param=<NAME>` keywords are all the parameter keywords that apply to the named parameter until the next occurrence of a parameter definition. These remaining parameter keywords can be in any order. When all the parameters associated with the first service are defined, specify a new `service=<NAME>` keyword definition.

[Listing 2-1](#) lists the keyword hierarchical order in a data file.

**Listing 2-1 Keyword Hierarchical Order in a Data File**

---

```

service=<NAME>
<service keyword>=<value>
<service keyword>=<value>
<service keyword>=<value>
param=<NAME>
<parameter keyword>=<value>
<parameter keyword>=<value>
param=<NAME>
<parameter keyword>=<value>
<parameter keyword>=<value>

```

---

## Using Service-Level Keywords and Values

A service definition must begin with the `service=<NAME>` keyword. Services using `CARRAY`, `STRING`, or `XML` buffer types should only have one parameter in the service. The recommended parameter name for a service that uses a `CARRAY` buffer type is `CARRAY` with `carray` as the data type. For a service that uses a `STRING` buffer type, the recommended parameter name is `STRING` with `string` as the data type. For a service that uses a `XML` buffer type, the recommended parameter name is `XML` with `xml` as the data type.

[Table 2-3](#) contains the guidelines for use of the service-level keywords and acceptable values for each.

**Table 2-3 Service-Level Keywords and Values**

Keyword	Value
<code>service</code>	Any Oracle Tuxedo service name
<code>export</code>	True or false (default is false)

**Table 2-3 Service-Level Keywords and Values (Continued)**

<b>Keyword</b>	<b>Value</b>
<code>inbuf/outbuf</code>	Select one of these buffer types: FML FML32 VIEW VIEW32 STRING CARRAY XML X_OCTET X_COMMON X_C_TYPE
<code>inview</code>	Any view name for input parameters  (This keyword is optional <i>only</i> if one of the following buffer types is used: VIEW, VIEW32, X_COMMON, X_C_TYPE.)
<code>outview</code>	Any view name for output parameters (Optional)

## Using Parameter-Level Keywords and Values

A parameter begins with the `param=<NAME>` keyword followed by a number of parameter keywords. It ends when another `param` or `service` keyword, or end-of-file is encountered. The parameters can be in any order after the `param=<NAME>` keyword.

[Table 2-4](#) contains the guidelines for use of the parameter-level keywords and acceptable values for each.

**Table 2-4 Parameter-Level Keywords and Values**

<b>Keyword</b>	<b>Values</b>
param	Any parameter name
type	byte short integer float double string carray xml boolean long view32
access	in out inout noaccess
count	Maximum number of occurrences (default is 1). The value for unlimited occurrences is 0. Used only by the Repository Editor to format test screens.
Subtype " (" and ") "	If the parameter is of <code>view32</code> type, this field specifies the view structure name. Otherwise, this field is ignored. All the sub-parameters of nested <code>view32</code> are enclosed in parentheses.

# Troubleshooting

If you encounter problems using the Bulk Loader utility, refer to [Table 2-5](#). For a complete list of Bulk Loader utility error messages and solutions, see “System Messages.”

**Table 2-5 Bulk Loader Troubleshooting Table**

<b>If . . .</b>	<b>Then . . .</b>
The data file is not found	Check to ensure that the path is correct.
The keyword is invalid	Check to ensure that the keyword is valid for the package, service, or parameter.
The value of the keyword is null	Type a value for the keyword.
The value is invalid	Check to ensure that the value of a parameter is within the allocated range for that parameter.
The data type is invalid	Check to ensure that the parameter is using a valid data type.

# Sample Bulk Load Data

**Listing 2-2** contains a sample data file in the correct format using the UNIX command `cat servicefile`. This sample loads `TRANSFER`, `LOGIN`, and `PAYROLL` service definitions to the `BULKPKG`.

## Listing 2-2 Sample Bulk Load Data

---

```
service=TRANSFER
export=true
inbuf=FML
outbuf=FML
param=ACCOUNT_ID
type=integer
access=in
count=2
param=SAMOUNT
type=string
access=in
param=SBALANCE
type=string
access=out
count=2
param=STATLIN
type=string
access=out

service=LOGIN
inbuf=VIEW
inview=LOGINS
outview=LOGINR
export=true
param=user
type=string
access=in
param=passwd
type=string
access=in
```

```

param=token
type=integer
access=out

service=PAYROLL
inbuf=FML
outbuf=FML
param=EMPLOYEE_NUM
type=integer
access=in
param=SALARY
type=float
access=inout
param=HIRE_DATE
type=string
access=inout

service=QUERY
export=true
inbuf=VIEW32
outbuf=VIEW32
inview=MYVIEW2
outview=MYVIEW2
    param=Long1
    type=long
    access=inout
    count=1

    param=Myview1
    type=view32
    subtype=MYVIEW1
    access=inout
    count=1
(
    param=Float1
    type=float
    access=inout
    count=1

```

```
param=Double1  
type=double  
access=inout  
count=1
```

```
param=Long1  
type=long  
access=inout  
count=3
```

```
param=String1  
type=string  
access=inout  
count=2
```

```
)
```

---



# Configuring the Oracle Jolt System

This chapter describes how to configure Oracle Jolt. “[Quick Configuration](#)” is for users who are familiar with Jolt. The other sections provide more detailed information. It is presumed that readers are system administrators or application developers who have experience with the operating systems and workstation platforms on which they are configuring Oracle Jolt.

This topic includes the following sections:

- [Quick Configuration](#)
- [Jolt Background Information](#)
- [Jolt Relay](#)
- [Jolt Relay Adapter](#)
- [Oracle Tuxedo Service Metadata Repository](#)
- [Event Subscription](#)
- [Oracle Tuxedo Background Information](#)
- [Sample Applications in Oracle Jolt Online Resources](#)

## Quick Configuration

If you are already familiar with Oracle Jolt and Oracle Tuxedo, “[Quick Configuration](#)” provides efficient guidelines for the configuration procedure. If you have not used Jolt, refer to “[Jolt Background Information](#)” on [page 3-5](#) before you begin any configuration procedures.

[Quick Configuration](#) contains the information you need to configure the Jolt Server Listener (JSL) on Oracle Tuxedo and covers the following procedures:

- [Editing the UBBCONFIG File](#)
- [Configuring the Tuxedo Service Metadata Repository](#)
- [Initializing Services That Use Oracle Tuxedo and the Repository Editor](#)
- [Configuring the Oracle Tuxedo TMUSREVT Server for Event Subscription](#)
- [Configuring Jolt Relay](#)

## Editing the UBBCONFIG File

1. In the `MACHINES` section, specify `MAXWSCLIENTS=number` (Required).

**Note:** If `MAXWSCLIENTS` is not set, JSL does not boot.

2. In the `GROUPS` section, set `GROUPNAME` *required parameters* [*optional parameters*].

3. Set the `SERVERS` section (Required).

Lines within this section have the form:

```
JSL required parameters [optional parameters]
```

where `JSL` specifies the file (*string\_value*) to be executed by `tmboot(1)`.

4. Set the required parameters for `JSL`.

Required parameters are:

```
SVRGRP=string_value
```

```
SRVID=number
```

```
CLOPT="-A...-n...//host port"
```

5. Set other parameters for `JSL`.

You can use the following parameters with the `JSL`, but you need to understand how doing so affects your application. Refer to [“Parameters Usable with JSL”](#) on page 3-30 for additional information.

```
MAX # of JSHs
```

```
MIN # of JSHs
```

## Configuring the Tuxedo Service Metadata Repository

See [Managing the Oracle Tuxedo Service Metadata Repository](#) for more information about configuring the Tuxedo Service Metadata Repository.

## Initializing Services That Use Oracle Tuxedo and the Repository Editor

Define the Oracle Tuxedo services that use Oracle Tuxedo and Oracle Jolt in order to make the Jolt services available to the client.

1. Build the Oracle Tuxedo server that contains the service.
2. Access the Oracle Tuxedo Service Metadata Repository Editor.

## Configuring the Oracle Tuxedo TMUSREVT Server for Event Subscription

Jolt Event Subscription receives event notifications from either Oracle Tuxedo services or other Oracle Tuxedo clients. Configure the Oracle Tuxedo TMUSREVT server and modify the application UBBCONFIG file. The following listing, “[TMUSREVT Parameters in the UBBCONFIG File](#),” shows the relevant TMUSREVT parameters in the UBBCONFIG file:

---

### Listing 3-1 TMUSREVT Parameters in the UBBCONFIG File

---

```
TMUSREVT          SRVGRP=EVBGRP1  SRVID=40          GRACE=3600
                  ENVFILE="/usr/tuxedo/bankapp/TMUSREVT.ENV"
                  CLOPT="-e tmusrevt.out -o tmusrevt.out -A --
                  -f /usr/tuxedo/bankapp/tmusrevt.dat"
                  SEQUENCE=11
```

---

In the **SERVERS** sections of the UBBCONFIG file, **specify the SRVGRP and SRVID.**

# Configuring Jolt Relay

## On UNIX

Start the JRLY process on UNIX by typing the following command at the system prompt:

```
jrly -f <config_file_path>
```

If the configuration file does not exist or cannot be opened, the JRLY writes a message to standard error, attempts to log the startup failure in the error log, then exits.

## On UNIX and Windows 2003

The format of the configuration file is a TAG=VALUE format. Blank lines or lines starting with a “#” are ignored. The following listing, “[Formal Configuration File Specifications](#),” is an example of the formal specifications of the configuration file.

### Listing 3-2 Formal Configuration File Specifications

---

```
LOGDIR=<LOG_DIRECTORY_PATH>
ACCESS_LOG=<ACCESS_FILE_NAME in LOGDIR>
ERROR_LOG=<ERROR_FILE_NAME in LOGDIR>
LISTEN=<IP:Port combination where JRLY will accept comma-separated
connections>
CONNECT=<IP:Port1, IP:Port2...IP:PortN:Port (List of IP:Port combinations
associated with JRADs: can be 1...N)>
```

---

## On Windows 2003 Only (Optional)

SOCKETTIMEOUT is the time in seconds for which JRLY Windows 2003 service blocks for network activity (new connections, data to be read, closed connections). SOCKETTIMEOUT also affects the Service Control Manager (SCM). When the SCM requests the Windows 2003 service to stop, the SCM must wait for at least SOCKETTIMEOUT seconds before quitting.

**Note:** The format for directory and filenames is determined by the operating system. UNIX systems use the forward slash (/). Windows 2003 systems use the backslash (\). If any files specified in LOGDIR, ACCESS\_LOG, or ERROR\_LOG cannot be opened for writing, JRLY prints an error message on stderr and exits.

The formats for the host names and the port numbers are shown in [Table 3-1](#).

## Start the Jolt Relay Adapter (JRAD)

1. Type `tmloadcf -y <UBBFILE>`.
2. Type `tmboot`.

## Configure the JRAD

A single JRAD process can only be connected to a single JRLY. A JRAD can be configured to communicate with only one JSL and its associated JSH. However, multiple JRADs can be configured to communicate with one JSL. The `CLOPT` parameter for Oracle Tuxedo services must be included in the `UBBCONFIG` file.

1. Type `-l hexadecimal format` (The JSL port to which the JRLY connects on behalf of the client.)
2. Type `-c hexadecimal format` (The address of the corresponding JSL to which JRAD connects.)

**Note:** The format is `0x0002PPPNNN`, or, in dot notation, `100.100.10.100`.

3. Configure networked components.

**Jolt is now configured.**

# Jolt Background Information

This section contains additional information on Jolt components.

## Jolt Server

The Jolt Server is a listener that supports one or more handlers.

**Jolt Server Listener (JSL)**—the JSL is configured to support clients on an IP/port combination. The JSL works with the Jolt Server Handler (JSH) to provide client connectivity to the back-end of the Oracle Jolt system. The JSL runs as an Oracle Tuxedo server.

**Jolt Server Handler (JSH)**—the JSH is a program that runs on an Oracle Tuxedo server machine to provide a network connection point for remote clients. The JSH works with the JSL to provide client connectivity residing on the back-end of the Oracle Jolt system. More than one JSH can be available to the JSL, up to 32,767. (Refer to the description of the `-M` command-line option in “[JSL Command-line Options](#)” on page 3-8 for additional information.)

**System Administrator Responsibilities**—the system administrator’s responsibilities for the server components of Oracle Jolt include:

- Determining the JSL network address.
- Determining the number of Jolt clients to be serviced. (The number of clients to be serviced is limited by `MAXWSCLIENTS` in `UBB`.)
- Determining the minimum and maximum number of JSHs.

## Starting the JSL

To start all administrative and server processes **in the** `UBBCONFIG` **file**:

1. Type `tmloadcf`.

This command parses the configuration file and loads the binary version of the configuration file.

2. Type `tmboot -y`.

This command activates the application specified in the configuration file.

If you do not enter any options, a prompt asks you if you really want to overwrite your `TUXCONFIG` file.

See *Administering an Oracle Tuxedo Application at Run Time* or the *Oracle Tuxedo Command Reference* for information about `tmloadcf` and `tmboot`.

## Shutting Down the JSL

All shutdown requests to the Jolt servers are initiated by the Oracle Tuxedo command:

```
tmshutdown -y
```

During shutdown:

- No new client connections are accepted.
- All current client connections are terminated. Oracle Tuxedo rolls back in-flight transactions. Each client receives an error message indicating that the service is unavailable.

## Restarting the JSL

Oracle Tuxedo monitors the JSL and restarts it in the event of a failure. When Oracle Tuxedo restarts the listener process, the following events occur:

- Clients attempting a listener connection must try to reconnect. Clients attempting a handler connection receive a timeout or a time delay.
- Clients currently connected to a handler are disconnected (JSH exits when its corresponding JSL exits normally).

## Configuring the JSL

The Jolt Server Listener (JSL) is an Oracle Tuxedo server responsible for distributing connection requests from Jolt to the Jolt Server Handler (JSH). Oracle Tuxedo must be running on the host machine where the JSL and JREPSVR are located.

**Note:** The way the JSL selects ports for the JSH is different than the process for the Oracle Tuxedo Workstation Server Listener (WSL). For detailed information regarding on properly configuring JSL ports, refer to the “SERVERS Section” of “[Creating the UBBCONFIG File](#)” on page 3-27.

## JSL Command-line Options

The server may need to obtain information from the command line. The CLOPT parameter allows you to specify command-line options that can change some defaults in the server. The JSL command-line options are described in [Table 3-1](#).

**Table 3-1 JSL Command-line Options**

Option	Description
[-a]	<p>Enables or disables the security context for a Jolt connection pool. This option should be enabled if you want to implement authentication propagation between WebLogic Server and Jolt. If identity propagation is desired, then the Jolt Service Handler (JSH) must be started with this option. If the -a option is not set, but SecurityContext is enabled, the JSH will not accept this request. If the SecurityContext attribute is enabled, then the Jolt client will pass the username of the caller to the JSH.</p> <p>If the JSH, gets a message with the caller's identity, it calls <code>impersonate_user()</code> to get the appkey for the user. JSH caches the appkey, so the next time the caller makes a request, the appkey is retrieved from the cache and the request is forwarded to the service. A cache is maintained by each JSH, which means that there will be a cache maintained for all the session pools connected to the same JSH.</p>
[-A]	<p>Specifies that certificate-based authentication should be required when accepting an SSL connection from a remote application.</p> <p><b>Note:</b> The JSL -A option is equivalent to the ISL(5) and WSL(5) -a option. For more information see, <a href="#">Section 5 - File Formats, Data Descriptions, MIBs, and System Processes Reference</a>.</p>
[-c <i>compression_threshold</i> ]	<p>Enables application data sent between a Jolt client and a Jolt server (JSH) to be compressed during transmission over the network.</p> <p><i>compression_threshold</i> is a number that you specify between 0 and 2,147,483,647 bytes. Any messages that are larger than the specified compression threshold are compressed before transmission.</p> <p>The default is no compression; that is, if no compression threshold is specified, Oracle Jolt does not compress messages on client or server.</p>
[-d <i>device_name</i> ]	<p>The device for platforms using the Transport Layer Interface. There is no default. Required. (Optional for sockets)</p>

**Table 3-1 JSL Command-line Options**

Option	Description
[-H <i>external_netaddr</i> ]	<p>Specifies the network address mask Jolt clients use to connect to the application when there is network address translation. The JSL process uses this address to listen for clients attempting to connect at this address. If the external address mask is 0x0002MMMMddddddd and the JSH network address is 0x00021111ffffffff, the known (or external) network address is 0x00021111ddddddd. If the address starts with "/" network address, the type is IP based and the TCP/IP port number of the JSH network address is copied into the address to form the combined network address.</p> <p>The external IP address mask must be specified in the following form: -H //external ip address:MMMM</p> <p>(Optional for JSL in Oracle Tuxedo 6.4 and 6.5)</p> <p><b>Note:</b> The option does not support IPv6.</p>
[-I <i>init-timeout</i> ]	<p>The time (in seconds) that a Jolt client is allowed to complete initialization through the JSH before it is timed out by the JSL. Default is 60 seconds. (Optional)</p>
[-j <i>connection_mode</i> ]	<p>The following connection modes from clients are allowed:</p> <p>RETAINED—the network connection is retained for the full duration of a session.</p> <p>RECONNECT—the client establishes and brings down a connection when an idle timeout is reached, reconnecting for multiple requests within a session.</p> <p>ANY—the server allows a client to request either a RETAINED or RECONNECT type of connection for a session.</p> <p>The default is ANY. That is, if no option is specified, the server allows a client to request either a RETAINED or RECONNECT type of connection. (Optional)</p>
[-K { <i>client</i>   <i>handler</i>   <i>both</i>   <i>none</i> }]	<p>The -K option turns on the network keep-alive feature for the client, the handler, or both. You can turn off this option for both the client and handler by specifying none.</p>
[-m <i>minh</i> ]	<p>The minimum number of JSHs that are available in conjunction with the JSL at one time. The range of this parameter is from 0 through 255. Default is 0. (Optional)</p>

**Table 3-1 JSL Command-line Options**

Option	Description
<code>[-M <i>maxh</i>]</code>	<p>The maximum number of JSHs that are available in conjunction with the JSL at one time. If this option is not specified, the parameter defaults to the <code>MAXWSClients</code> divided by the <code>-x</code> multiplexing factor (MPX), with the result rounded up. If specified, the <code>-M</code> option takes a value from 1 to 32,767. (Optional)</p>
<code>[-n <i>netaddr</i>]</code>	<p>Network address used by the Oracle Jolt listener with Oracle Tuxedo 6.4 and 6.5, and WebLogic Enterprise 4.2.</p> <p>TCP/IP addresses may be specified in the following formats:</p> <ul style="list-style-type: none"><li>• IPv4<ul style="list-style-type: none"><li><code>//IP:port</code></li><li><code>//hostname:port_number</code></li><li><code>//#.##.##.##:port_number</code></li></ul><p>The domain finds an address for <i>hostname</i> by using the local name resolution facilities (usually DNS). <i>hostname</i> must be the local machine, and the local name resolution facilities must unambiguously resolve <i>hostname</i> to the address of the local machine.</p><p>The “#.##.##.##” is in dotted decimal format. In dotted decimal format, each # should be a number from 0 to 255. This dotted decimal number represents the IP address of the local machine. In both of the above formats, <i>port_number</i> is the TCP port number at which the domain process listens for incoming requests. <i>port_number</i> can either be a number between 0 and 65535 or a name.</p></li><li>• IPv6<ul style="list-style-type: none"><li><code>//[IPv6 address]:port</code></li><li><code>//hostname:port_number</code></li></ul><p><b>Note:</b> IPv6 does not support hexadecimal format</p></li><li>• SDP<ul style="list-style-type: none"><li><code>sdp://IB_IP:port</code></li></ul></li></ul>

**Table 3-1 JSL Command-line Options**

Option	Description
[-R renegotiation-interval]	<p>Specifies the renegotiation interval in minutes. After the specified number of minutes have elapsed without renegotiation of the SSL encryption parameters for a particular SSL session, the SSL encryption parameters will be renegotiated on the next exchange of data, as described in the SSL and TLS standards. The default is 0 minutes which results in no periodic session renegotiation.</p> <p><b>Note:</b> If the -R parameter is specified and the -S parameter is not specified or set to 0, the JSL sends a warning message to the userlog.</p>
[-S <i>Client-timeout</i> ]	<p>The idle time (in minutes) when the client does not have any outstanding requests. In other words, when the client is “snoozing.”</p> <p>This option can be used together with the -T option. When either timeout reached, JSH will close the session.</p> <p>If a parameter is not specified, the default is no timeout. (Optional)</p>
[-s secure-port]	<p>Specifies the port number that the JSL should use to listen for secure connections using the SSL protocol. You can configure the JSL to allow only secure connections by setting the port numbers specified by the -s and -n options to the same value.</p> <p>This option cannot be used if the JRLY and JRAD processes are used.</p> <p>The JSL -s option is equivalent to the ISL(5) and WSL(5) -S option. For more information see, <a href="#">Section 5 - File Formats, Data Descriptions, MIBs, and System Processes Reference</a>.</p>
[-T <i>Client-timeout</i> ]	<p>The time (in minutes) allowed for a client to stay idle. If a client does not make any requests during this time, the JSH disconnects the client and the session is terminated. If an argument is not supplied, the session does not timeout.</p> <p>When the -j ANY or -j RECONNECT option is used, always specify -T with an idle timeout value. If -T is not specified and the connection is suspended, JSH does not automatically terminate the session. The session never terminates if a client abnormally ends the session.</p> <p>If a parameter is not specified, the default is no timeout. (Optional)</p>
[-w JSH]	<p>This command-line option indicates the Jolt Server Handler. Default is JSH. (Optional)</p>

**Table 3-1 JSL Command-line Options**

Option	Description
<code>[-x <i>mpx-factor</i>]</code>	This is the number of clients that one JSH can service. Use this parameter to control the degree of multiplexing within each JSH process. If specified, this parameter takes a value from 1 to 32767 for UNIX and Windows 2003. Default value is 10. (Optional)
<code>[-z 0 56 128 256]</code>	Specifies the minimum level of encryption when establishing a network connection between a Jolt client and the JSH. 0 means no encryption while 56, 128, and 256 specify the length (in bits) of the encryption key. If this minimum level of encryption cannot be met, a connection will not be established.
<code>[-Z 0 56 128 256]</code>	When a network link between a Jolt client and the JSH is being established, this option allows encryption up to the specified level. The initial 0 means no DH nodes, no RC4. The numbers 56, 128, and 256 specify the length (in bits) of the encryption key. Either SSL or the DH key exchange is needed to generate keys. Session keys are not transmitted over the network. The default value is 0.  <b>Note:</b> A 0-bit maximum encryption level is not compatible with the -s SSL connection option.

## Security and Encryption

When LLE is used for Jolt security and encryption, authentication and key exchange data are transmitted between Jolt clients and the JSL/JSH using the Diffie-Hellman key exchange. All subsequent exchanges are encrypted using RC4 encryption. International packages use a DES key exchange and a 128 bit key, with 40 bits encrypted and 88 bits exposed.

When SSL is used for Jolt security and encryption, the SSL protocol is used for authentication, key exchange, and data exchange.

## Jolt Relay

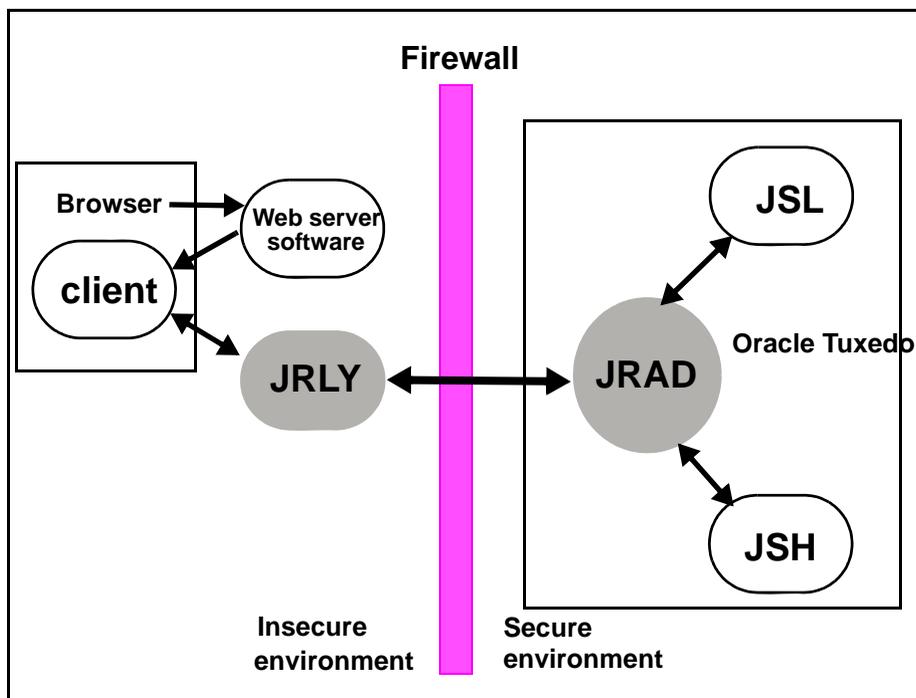
The combination of the Jolt Relay (JRLY) and its associated Jolt Relay Adapter (JRAD) is typically referred to as the Internet Relay. Jolt Relay routes messages from a Jolt client to a JSL or JSH. This eliminates the need for the JSH and Oracle Tuxedo to run on the same machine as the Web server (which is generally considered insecure). The Jolt Relay consists of the two components illustrated in the figure [“Jolt Internet Relay Path” on page 3-13](#).

- **Jolt Relay (JRLY)**—the JRLY is the Jolt Relay front-end. It is not an Oracle Tuxedo client or server and is not dependent on the Oracle Tuxedo version. It is a stand-alone software component. It requires only minimal configuration to allow it to work with Jolt clients.
- **Jolt Relay Adapter (JRAD)**—the JRAD is the Jolt Relay back-end. It is an Oracle Tuxedo system server, but does not include any Oracle Tuxedo services. It requires command-line arguments to allow it to work with the JSL and the Oracle Tuxedo system.

**Notes:** The Jolt Relay is transparent to Jolt clients and Jolt servers. A Jolt server can simultaneously connect to intranet clients directly, or through the Jolt Relay to Internet clients.

Tuxedo 10 supports SSL for Jolt clients and the JSL/JSH; however, SSL support has not been implemented for the JRAD and JRLY. Therefore, Tuxedo 10 Jolt configurations using SSL cannot make use of the JRAD and JRLY processes.

**Figure 3-1 Jolt Internet Relay Path**



This figure illustrates how a browser connects to the Web server software and downloads the Oracle Jolt applets. The Jolt applet or client connects to the JRLY on the Web server machine. The JRLY forwards the Jolt messages across the firewall to the JRAD. The JRAD selectively forwards messages to the JSL or appropriate JSH.

## Jolt Relay Failover

There are two points of failover associated with JRLY:

- Jolt Client to JRLY connection failover
- JRLY to JRAD connection failover

### Jolt Client to JRLY Connection Failover

If one server address does not result in a successful session, the failover function allows the Jolt Client API to connect to the next free (unconnected) JRLY specified in the argument list of the API. To enable this failover in a Windows 2003 environment, multiple Windows 2003 JRLY services can be executed. In a non-Windows 2003 environment, multiple JRLY processes are executed. Each JRLY (service or process) has its own configuration file. This type of failover is handled by the client API features in Oracle Jolt, which allow you to specify a list of Jolt server addresses (JSL or JRLY).

### JRLY to JRAD Adapter Connection Failover

Each JRLY configuration file has a list of JRAD addresses. When a JRAD is unavailable, JRLY tries to connect to the next free (unconnected) JRAD, in a round-robin fashion. Two JRLYs cannot connect to the same JRAD. Given these facts, you can make the connection efficient by giving different JRAD address orders. That is, if you make one extra JRAD available on standby, the first JRLY that loses its JRAD connects to the extra JRAD. This type of failover is handled by JRLY alone.

If any of the listed JRADs are not executing when JRLY is started, the initial connection fails. When a Jolt client tries to connect to JRLY, the JRLY again tries to connect to the JRAD.

To accommodate the failover functionality, you have to boot multiple JRADs by configuring them in the `UBBCONFIG` file.

## Jolt Relay Process

The JRLY (front-end relay) process can be started before or after the JRAD is started. If the JRAD is not available when the JRLY is started, the JRLY attempts to connect to the JRAD when

it receives a client request. If JRLY is still unable to connect to the JRAD, the client is denied access and a warning is written to the JRLY error log file.

## Starting the JRLY on UNIX

Start the JRLY process by typing the command name at a system prompt.

```
jrly -f config_file_path
```

If the configuration file does not exist or cannot be opened, the JRLY prints an error message.

If the JRLY is unable to start, it writes a message to standard error and attempts to log the startup failure in the error log, then exits.

## JRLY Command-line Options for Windows 2003

This section describes command-line options that are available from the Windows 2003 version of `JRLY.exe`. Note the following:

- JRLY as a Windows service is available only for Windows 2003.
- When the display suffix is optional (when `[display_suffix]` is shown), all operations are performed on the default JRLY Windows 2003 service instance.
- For manually installed, additional JRLY services, a suffix (any string) is required. Also, you can install the default service manually by omitting the optional string suffix.
- Each instance of JRLY Windows 2003 service uses the same binary executable file.
- A new process is started for each instance of JRLY Windows 2003 service.
- The syntax for these options is: `jrly -command`.
- Text specified within brackets ( `[ ]` ) is optional.
- All commands in the following list of command options except `-start` and `-stop` require that you have write access to Windows 2003 Registry.
- The `-start` and `-stop` commands require that you have Windows 2003 Service control access. These requirements are based on Windows 2003 user restrictions.

The JRLY command-line options are detailed in [Table 3-2](#):

**Table 3-2 JRLY Command-line Options for Windows 2003**

Option	Description
<pre>jrly -install [display_suffix]</pre>	<p><b>Install jrly as a Windows 2003 service.</b></p> <p><b>Example 1:</b></p> <pre>jrly -install</pre> <p>In this example, the default JRLY is installed as a Windows 2003 Service and is displayed in the Service Control Manager (SCM) as <b>Jolt Relay</b>.</p> <p><b>Example 2:</b></p> <pre>jrly -install MASTER</pre> <p>In this case, an instance of JRLY is installed as a Windows 2003 Service and is displayed in the SCM as <b>Jolt Relay_MASTER</b>. The suffix, MASTER, does not have any significance; it is only used to uniquely identify various instances of JRLYs.</p> <p>At this point, this instance of JRLY is not ready to start. It must be assigned the configuration file (see the <code>set</code> command discussion) that specifies the listening TCP/IP port, JSH connection TCP/IP port, log files, and <code>sockettimeout</code>. This file should not be shared between various instances of JRLY.</p>
<pre>jrly -remove [display_suffix]   -all</pre>	<p>Remove one or all instances of JRLY from Windows 2003 service.</p> <p>If <code>[display_suffix]</code> is specified, this command removes the specified JRLY service.</p> <p>If <code>[display_suffix]</code> is not specified, this command removes the default JRLY from being a Windows 2003 Service.</p> <p>If the <code>-all</code> option is specified, all JRLY Windows 2003 Services are removed. Related Windows 2003 registry entries under <code>HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Oracle JoltRelay</code> and <code>HKEY_LOCAL_MACHINE\Software\Oracle Systems\Jolt\x.x</code> are removed.</p>

**Table 3-2 JRLY Command-line Options for Windows 2003 (Continued)**

<pre>jrlly -set [-d <i>display_suffix</i>] -f <i>config_file</i></pre>	Update the registry with the full path of a new configuration file.
<p><b>Example 1:</b></p> <pre>jrlly -set -f c:\tux71\udataobj\jolt\jrlly.con</pre> <p>In this example, the default JRLY Windows 2003 Service (Jolt Relay) is assigned a configuration file called <code>jrlly.con</code> that is located in: <code>c:\tuxdir\udataobj\jolt</code> directory.</p>	
<p><b>Example 2:</b></p> <pre>jrlly -set -d MASTER -f c:\tuxdir\udataobj\jolt\master.con</pre>	
<p>Here, the JRLY Windows 2003 Service instance, called <b>Jolt Relay_MASTER</b> is assigned a configuration file called <code>jrlly_master.con</code> that is located in <code>c:\tuxdir\udataobj\jolt</code> directory.</p>	
<pre>jrlly -manual [<i>display_suffix</i>]</pre>	<p>Set the start/stop to manual.</p> <p>This command sets the specified JRLY instance to be manually controlled, using either the command-line options or the SCM.</p>
<pre>jrlly -auto [<i>display_suffix</i>]</pre>	<p>Set the start/stop to automatic.</p> <p>This command sets all the operations for a specified Windows 2003 Service to be automatically started when the OS boots and stopped when the OS shuts down.</p>
<pre>jrlly -start [<i>display_suffix</i>]</pre>	Start the specified JRLY.
<pre>jrlly -stop [<i>display_suffix</i>]</pre>	Stop the specified JRLY.
<pre>jryl -version</pre>	Print the current version of JRLY binary.
<pre>jrlly -help</pre>	Print command-line options with brief descriptions.

## JRLY Command-line Option for UNIX

There is only one JRLY command-line option for UNIX as shown in [Table 3-3](#):

**Table 3-3 JRLY Command-line Option for UNIX**

Option	Description
<code>jrlly -f config_file_path</code>	Start the JRLY process.  This option starts the JRLY process. If the configuration file does not exist or cannot be opened, the JRLY prints an error message. If the JRLY cannot start, it writes a message to standard error, attempts to log the startup failure in the error log, then exits.

## JRLY Configuration File

The format of the configuration file is a TAG=VALUE format. Blank lines or lines starting with a “#” are ignored. [Listing 3-3](#) contains an example of the formal specifications of the configuration file.

**Listing 3-3 Specification of Configuration File**

```
LOGDIR=<LOG_DIRECTORY_PATH>
ACCESS_LOG=<ACCESS_FILE_NAME in LOGDIR>
ERROR_LOG=<ERROR_FILE_NAME in LOGDIR>
LISTEN=<IP:Port combination where JRLY will accept connections>
CONNECT=<IP:Port combination associated with JRAD>
SOCKETTIMEOUT=<Seconds for socket accept() function>
```

**Note:** `SOCKETTIMEOUT` is the duration (in seconds) of which the relay Windows 2003 service blocks the establishment of new socket connections to allow network activity (new connections, data to be read, closed connections). It is valid only on Windows 2003 machines. `SOCKETTIMEOUT` also affects the SCM. When the SCM requests that the service stop, the SCM needs to wait at least `SOCKETTIMEOUT` seconds before doing so.

[Listing 3-4](#) shows an example of the JRLY configuration file. The `CONNECT` line specifies the IP address and port number of JRAD machine.

---

#### Listing 3-4 Example of JRLY Configuration File

---

```
LOGDIR=/usr/log/relay
ACCESS_LOG=access_log
ERROR_LOG=errorlog
# jrly will listen on port 4444
LISTEN=200.100.10.100:4444
CONNECT=machine1:port1
CONNECT=machine2:port2

SOCKETTIMEOUT=30           //See text under listing
```

---

The format for directory and filenames is determined by the operating system. UNIX systems use the forward slash (/). Windows 2003 systems use the backslash (\). If any file specified in `LOGDIR`, `ACCESS_LOG` or `ERROR_LOG` cannot be opened for writing, the JRLY prints an error message on `stderr` and exits.

The formats for host names and port numbers are shown in [Table 3-4](#).

**Note:** JRLY supports IPv6.

**Table 3-4 Host Name and Port Number Formats**

IPv4	IPv6
//IP:port	//[IPv6 address]:port
IP is a dotted notation IP address, port is a decimal number	

---

**Table 3-4 Host Name and Port Number Formats**

IPv4	IPv6
<code>//hostname:port_number</code> IP is a dotted notation IP address, port is a decimal number	<code>//hostname:port_number</code>
<code>//#. #. #. #:port_number</code>	Hex format is not supported

## Jolt Relay Adapter

The Jolt Relay Adapter (back-end relay) is an Oracle Tuxedo system server. The Jolt Relay Adapter (JRAD) server may or may not be located on the same Oracle Tuxedo host machine in single host mode (SHM) and server group to which the JSL server is connected.

The JRAD can be started independently of its associated JRLY. JRAD tracks its startup and shutdown activity in the Oracle Tuxedo log file.

## JRAD Configuration

A single JRAD process can only be connected to a single JRLY. A JRAD can be configured to communicate with only one JSL and its associated JSHs. However, multiple JRADs can be configured to communicate with one JSL. The `CLOPT` parameter for the Oracle Tuxedo servers must be included in the `UBBCONFIG` file. A sample of the file is shown in the listing “[Sample JRAD Entry in UBBCONFIG File](#)” on page 3-22.

[Table 3-5](#) contains additional information about the `CLOPT` parameters.

**Table 3-5 JRAD CLOPT Parameter Descriptions**

CLOPT Parameter	Description
-l <i>netaddr</i>	<p data-bbox="619 435 1139 487">Port to listen for the JRJLY to connect on behalf of the client.</p> <ul data-bbox="619 505 706 527" style="list-style-type: none"> <li data-bbox="619 505 706 527">• IPv4           <ul data-bbox="659 539 964 635" style="list-style-type: none"> <li data-bbox="659 539 784 562">//IP:port</li> <li data-bbox="659 574 964 597">//hostname:port_number</li> <li data-bbox="659 609 951 631">//#. #. #. #:port_number</li> </ul> <p data-bbox="659 649 1170 791">The domain finds an address for <i>hostname</i> by using the local name resolution facilities (usually DNS). <i>hostname</i> must be the local machine, and the local name resolution facilities must unambiguously resolve <i>hostname</i> to the address of the local machine.</p> <p data-bbox="659 808 1170 951">In the second example, the “#. #. #. #” is in dotted decimal format. In dotted decimal format, each # should be a number from 0 to 255. This dotted decimal number represents the IP address of the local machine. In both of the above formats, <i>port_number</i> is the TCP port number at which the domain process listens for incoming requests. <i>port_number</i> can either be a number between 0 and 65535 or a name.</p> </li> <li data-bbox="619 1078 706 1100">• IPv6           <ul data-bbox="659 1112 897 1177" style="list-style-type: none"> <li data-bbox="659 1112 857 1135">//[IPv6 address]:port</li> <li data-bbox="659 1147 897 1170">//hostname:port_number</li> </ul> <p data-bbox="659 1187 1139 1209"><b>Note:</b> IPv6 does not support hexadecimal format.</p> </li> </ul>

**Table 3-5 JRAD CLOPT Parameter Descriptions (Continued)**

CLOPT Parameter	Description
<code>-c netaddr</code>	The address of the corresponding JSL to which JRAD connects.  Ipv4 and IPv6 address format same as <code>-l netaddr</code> .
<code>-H netaddr</code>	The listening address for an external proxy. An external proxy is one that runs on a client host. This proxy handles HTTP and other protocols. The other end of the proxy connects to JRLY, which connects to JSL/JSH.  In order for the proxy to work for Jolt clients (specifically applets that connect to JRLY), the JRAD passes the <code>-H</code> argument to an applet, instructing it to connect to the proxy address instead of the JRLY address.  <b>Note:</b> Unlike the JSL <code>-H</code> option, the JRAD <code>-H</code> option is not used as a network address translator, nor is it used as an address mask. IPv6 does not support the JRAD <code>-H</code> option.

The address for the JRAD CLOPT parameters can be specified in either of the following formats:

`//hostname:port`

`0x0002pppphhhhhhhh`

(where `pppp` is the port number and `hhhhhhhh` is the hexadecimal IP address)

Listing 3-5 shows the sample JRAD entry in UBBCONFIG file.

### Listing 3-5 Sample JRAD Entry in UBBCONFIG File

```
# JRAD host 200.100.100.10 listens at port 2000, connects to JSL port 8000
on the same host

JRAD    SRVGRP=JSLGRP    SRVID=60
        CLOPT="-A -- -l 0x000207D0C864640A -c 0x00021f40C864640A"
```

## Network Address Configurations

A Jolt Internet Relay configuration requires that several networked components work together. Prior to configuration, review the criteria in [Table 3-6](#) and record the information to minimize the possibility of misconfiguration.

**Table 3-6 Jolt Internet Relay Network Address Configuration Criteria**

JRLY	JRAD	JSL
LISTEN: <i>Location where the clients connect.</i>	-l: <i>Location where the listener connects to the JRLY.</i>	-n: <i>Location of JSL. Must match -c parameter of JRAD.</i>
CONNECT: <i>Location of your JRAD. Must match the -l parameter of JRAD.</i>	-c: <i>Location of JSL. Must match -n parameter of JSL.</i>	

## Oracle Tuxedo Service Metadata Repository

The Oracle Tuxedo Service Metadata Repository contains Oracle Tuxedo service definitions that allow Jolt clients to access Oracle Tuxedo services. See [Managing The Oracle Tuxedo Service Metadata Repository](#) for the instructions.

### Initializing Services By Using Oracle Tuxedo and the Repository Editor

Define the Oracle Tuxedo services by using Oracle Tuxedo and Oracle Tuxedo Service Metadata Repository Editor in order to make the Jolt services available to the client.

1. Build the Oracle Tuxedo server containing the service. See [Administering an Oracle Tuxedo Application at Run Time](#) or [Programming Oracle Tuxedo ATMI Applications Using C](#) for additional information on the following:
  - Building the Oracle Tuxedo application server
  - Editing the UBBCONFIG file
  - Updating the TUXCONFIG file
  - Administering the `tmboot` command

2. Access the Metadata Repository Editor.

## Event Subscription

Jolt Event Subscription receives event notifications from either Oracle Tuxedo services or other Oracle Tuxedo clients:

- **Unsolicited Event Notifications**—a Jolt client receives these notifications as a result of a Oracle Tuxedo client or service subscribing to unsolicited events, and an Oracle Tuxedo client issuing a broadcast (using either a `tpbroadcast()` or a directly targeted message via a `tpnotify()` ATMI call). Unsolicited event notifications do not need the `TMUSREVT` server.
- **Brokered Event Notifications**—a Jolt client receives these notifications through the Oracle Tuxedo Event Broker. The notifications are only received when both Jolt clients subscribe to an event and any Oracle Tuxedo client or server posts an event using `tppost()`. Brokered event notifications require the `TMUSREVT` server.

## Configuring for Event Subscription

Configure the Oracle Tuxedo `TMUSREVT` server and modify the application `UBBCONFIG` file. [Listing 3-6](#) shows the relevant sections of `TMUSREVT` parameters in the `UBBCONFIG` file. See *Programming Oracle Tuxedo ATMI Applications Using C* for information about the syntax of the entries for the file.

### Listing 3-6 UBBCONFIG File

---

```
TMUSREVT      SRVGRP=EVBGRP1  SRVID=40          GRACE=3600
              ENVFILE="/usr/tuxedo/bankapp/TMUSREVT.ENV"
              CLOPT="-e tmusrevt.out -o tmusrevt.out -A --
              -f /usr/tuxedo/bankapp/tmusrevt.dat"
              SEQUENCE=11
```

---

In the `SERVERS` section of the `UBBCONFIG` file, modify the `SRVGRP` and `SRVID` parameters as needed.

## Filtering Oracle Tuxedo FML or VIEW Buffers

*Filtering* is a process that allows you to customize a subscription. If you require additional information about the Oracle Tuxedo Event Broker, subscribing to events, or filtering, refer to *Programming Oracle Tuxedo ATMI Applications Using C*.

In order to filter Oracle Tuxedo FML or VIEW buffers, the field definition file must be available to Oracle Tuxedo at run time.

**Note:** There are no special requirements for filtering STRING buffers.

### Buffer Types

Table 3-7 shows the Oracle Tuxedo types.

**Table 3-7 Oracle Tuxedo Buffer Types**

Buffer Type	Description
FML	Attribute, value pair. Explicit.
VIEW	C structure. Very precise offsetting. Implicit.
STRING	Length and offset are different values. All readable.
CARRAY	Character array. BLOB of binary data. Only client and server know - JSL doesn't.
X_C_TYPE	Equivalent to VIEW.
X_COMMON	Equivalent to VIEW, but used for both COBOL and C.
X_OCTET	Equivalent to CARRAY.
XML	Well-formed XML documents. Similar to CARRAY.

### FML Buffer Example

The listing “[FIELDTBLS Variable in the TMUSREVT.ENV File](#)” on page 3-26 shows an example that uses the FML buffer. The FML field definition table is made available to Oracle Tuxedo by setting the `FIELDTBLS` and `FLDTBLDIR` variables.

To filter a field found in the `my.fields` file:

1. Copy the `my.flds` file to `/usr/me/bankapp` directory.
2. Add `my.flds` to the `FIELDTBLS` variable in the `TMUSREVT.ENV` file as shown in the following listing:

---

### Listing 3-7 FIELDTBLS Variable in the TMUSREVT.ENV File

---

```
FIELDTBLS=Usysflds,bank.flds,credit.flds,event.flds,my.flds
FLDTBLDIR=/usr/tuxedo/me/T6.2/udataobj:/usr/me/bankapp
```

---

If `ENVFILE="/usr/me/bankapp/TMUSREVT.ENV"` is included in the definition of the `UBBCONFIG` file (shown in the listing “[UBBCONFIG File](#)” on page 3-24), the `FIELDTBLS` and `FLDTBLDIR` definitions are taken from the `TMUSREVT.ENV` file and not from your environment variable settings.

If you remove the `ENVFILE="/usr/me/bankapp/TMUSREVT.ENV"` definition, the `FIELDTBLS` and `FLDTBLDIR` definitions are taken from your environment variable settings. The `FIELDTBLS` and `FLDTBLDIR` definitions must be set to the appropriate value prior to booting the Oracle Tuxedo system.

For additional information on event subscriptions and the Oracle Jolt Class Library, refer to [Chapter 4, “Using the Jolt Class Library.”](#)

## Oracle Tuxedo Background Information

The following sections provide detailed configuration information. Even if you are familiar with Oracle Tuxedo, you should refer to this section for information concerning Jolt Service Handler (JSL) configuration.

### Configuration File

The Oracle Tuxedo configuration file for your application exists in two forms, the ASCII file, `UBBCONFIG`, and a compiled version called `TUXCONFIG`. Once you create a `TUXCONFIG`, consider your `UBBCONFIG` as a backup.

You can make changes to the `UBBCONFIG` file with your preferred text editor. Then, at a time when your application is not running, and when you are logged in to your `MASTER` machine, you can recompile your `TUXCONFIG` by running `tmloadcf(1)`. `System/T` prompts you to make

sure you really want to overwrite your existing `TUXCONFIG` file. (If you enter the command with the `-y` option, the prompt is suppressed.)

## Creating the UBBCONFIG File

A binary configuration file called the `TUXCONFIG` file contains information used by `tmboot(1)` to start the servers and initialize the bulletin board of an Oracle Tuxedo system in an orderly sequence. The binary `TUXCONFIG` file cannot be created directly. Initially, you must create a `UBBCONFIG` file. That file is parsed and loaded into the `TUXCONFIG` using `tmloadcf(1)`. Then `tmadmin(1)` uses the configuration file or a copy of it in its monitoring activity. `tmshutdown(1)` references the configuration file for information needed to shut down the application.

## Configuration File Format

The `UBBCONFIG` file can consist of up to nine specification sections. Lines beginning with an asterisk (\*) indicate the beginning of a specification section. Each such line contains the name of the section immediately following the \*. Allowable section names are: `RESOURCES`, `MACHINES`, `GROUPS`, `NETGROUPS`, `NETWORK`, `SERVERS`, `SERVICES`, `INTERFACES`, and `ROUTING`.

**Note:** The `RESOURCES` (if used) and `MACHINES` sections *must* be the first two sections, in that order; the `GROUPS` section must be ahead of `SERVERS`, `SERVICES`, and `ROUTING`.

To configure the JSL, you must modify the `UBBCONFIG` file. For further information about Oracle Tuxedo configuration, refer to *Administering an Oracle Tuxedo Application at Run Time*.

[Listing 3-8](#) shows relevant portions of the `UBBCONFIG` file.

### Listing 3-8 UBBCONFIG File

---

```
*MACHINES
MACH1  LMID=SITE1
        MAXWSCLIENTS=40

*GROUPS
JSLGRP          GRPNO=95  LMID=SITE1

*SERVERS
JSL SRVGRP=JSLGRP SRVID=30 CLOPT= " -- -n 0x0002PPPPNNNNNNNN -d
/dev/tcp -m2 -M4 -x10"
```

---

The parameters shown in the following table are the only parameters that must be designated for the Jolt Server groups and Jolt Servers. You are not required to specify any other parameters.

Change the sections of the `UBBCONFIG` file as shown in [Table 3-8](#).

**Table 3-8 UBBCONFIG File Sections**

Section	Parameters to be specified
MACHINES	MAXWSCLIENTS
GROUPS	GRPNO, LMID
SERVERS	SRVGRP, SRVID, CLOPT

## MACHINES Section

The `MACHINES` section specifies the logical names for physical machines for the configuration. It also specifies parameters specific to a given machine. The `MACHINES` section must contain an entry for each physical processor used by the application. Entries have the form:

*ADDRESS or NAME required parameters [optional parameters]*

where *ADDRESS* is the physical name of the processor, for example, the value produced by the UNIX system `uname -n` command.

*LMID=string\_value*

This parameter specifies that the *string\_value* is to be used in other sections as the symbolic name for *ADDRESS*. This name cannot contain a comma, and must be 30 characters or less. This parameter is required. There must be an `LMID` line for every machine used in a configuration.

*MAXWSCLIENTS=number*

The `MAXWSCLIENTS` parameter is required in the `MACHINES` section of the configuration file. It specifies the number of accesser entries on this processor to be reserved for Jolt and Workstation clients only. The value of this parameter must be between 0 and 32,768, inclusive.

The Jolt Server and Workstation use `MAXWSCLIENTS` in the same way. For example, if 200 slots are configured for `MAXWSCLIENTS`, this number configures Oracle Tuxedo for the total number of remote clients used by Jolt and Workstation.

Be sure to specify `MAXWSCLIENTS` in the configuration file. If it is not specified, the default is 0.

**Note:** If `MAXWSCLIENTS` is not set, the JSL does not boot.

## GROUPS Section

This section provides information about server groups, and must have at least one server group defined in it. A server group entry provides a logical name for a collection of servers and/or services on a machine. The logical name is used as the value of the `SRVGRP` parameter in the `SERVERS` section to identify a server as part of this group. `SRVGRP` is also used in the `SERVICES` section to identify a particular instance of a service with its occurrences in the group. Other `GROUPS` parameters associate this group with a specific resource manager instance (for example, the employee database). Lines within the `GROUPS` section have the form:

```
GROUPNAME required parameters [optional parameters]
```

where `GROUPNAME` specifies the logical name (string\_value) of the group. The group name must be unique within all group names in the `GROUPS` section and `LMID` values in the `MACHINES` section. The group name cannot contain an asterisk(\*), comma, or colon, and must be 30 characters or less.

A `GROUPS` entry is required for the group that includes the Jolt Server Listener (JSL). Make the `GROUPS` entry as follows:

1. The group name is selected by the application, for example: `JSLGRP` and `JREPGRP`.
2. Specify the same identifiers given as the value of the `LMID` parameter in the `MACHINES` section.
3. Specify the value of the `GRPNO` between 1 and 30,000 in the `*GROUPS` section.

**Note:** Make sure that Resource Managers are *not* assigned as a default value for all groups in the `GROUPS` section of your `UBBCONFIG` file. Making Resource Managers the default value assigns a Resource Manager to the JSL and you receive an error during `tmboot`. In the `SERVERS` section, default values for `RESTART`, `MAXGEN`, etc., are acceptable defaults for the JSL.

## SERVERS Section

This section provides information on the initial conditions for servers started in the system. The notion of a server as a process that continually runs and waits for a server group's service requests to process may or may not apply to a particular remote environment. For many environments, the operating system, or perhaps a remote gateway, is the sole dispatcher of services. When either of these is the case, you need only specify `SERVICE` entry points for remote program entry points, and not `SERVER` table entries. Oracle Tuxedo system gateway servers would advertise and queue remote domain service requests. Host-specific reference pages must indicate whether or not

UBBCONFIG server table entries apply in their particular environments, and if so, the corresponding semantics. Lines within the `SERVERS` section have the form:

```
AOUT required parameters [optional parameters]
```

where `AOUT` specifies the file (*string\_value*) to be executed by `tmboot(1)`. `tmboot` executes `AOUT` on the machine specified for the server group to which the server belongs. `tmboot` searches for the `AOUT` file on its target machine, thus, `AOUT` must exist in a file system on that machine. (Of course, the path to `AOUT` can include RFS connections to file systems on other machines.) If a relative pathname for a server is given, the search for `AOUT` is done sequentially in `APPDIR`, `TUXDIR/bin`, `/bin`, and then in *path*, where *<path>* is the value of the last `PATH=` line appearing in the machine environment file, if one exists. The values for `APPDIR` and `TUXDIR` are taken from the appropriate machine entry in the `TUXCONFIG` file.

Clients connect to Oracle Jolt applications through the Jolt Server Listener (JSL). Services are accessed through the Jolt Server Handler (JSH). The JSL supports multiple clients and acts as a single point of contact for all the clients to connect to the application at the network address that is specified on the JSL command line. The JSL schedules work for handler processes. A handler process acts as a substitute for clients on remote workstations within the administrative domain of the application. The handler uses a multiplexing scheme to support multiple clients on one port concurrently.

The network address specified for the JSL designates a TCP/IP address for both the JSL and any JSH processes associated with that JSL. The port number identified by the network address specifies the port number on which the JSL accepts new client connections. Each JSH associated with the JSL uses consecutive port numbers at the same TCP/IP address. For example, if the initial JSL port number is 8000 and there are a maximum of three JSH processes, the JSH processes use ports 8001, 8002, and 8003.

**Note:** Misconfiguration of the subsequent JSL results in a port number collision.

## Parameters Usable with JSL

In addition to the parameters specified in the previous sections, the following parameters can be used with the JSL, although you need to understand how doing so would affect your application.

```
SVRGRP=string_value
```

This parameter specifies the group name for the group in which the server is to run. *string\_value* must be the logical name associated with a server group in the `*GROUPS` section, and must be 30 characters or less. This association with an entry in the `*GROUPS` section means that `AOUT` is executed on the machine with the `LMID` specified for the server group. This association also specifies the `GRPNO` for the server group and parameters to pass when the

associated resource manager is opened. All server entries must have a server group parameter specified.

`SRVID=number`

This parameter specifies an *identifier*, an integer between 1 and 30,000, inclusive, that identifies this server within its group. This parameter is required on every server entry, even if the group has only one server. If multiple occurrences of servers are desired, do not use consecutive numbers for `SRVIDS`; leave enough room for the system to assign additional `SRVIDS` up to `MAX`.

## Optional Parameters

The optional parameters of the `SERVERS` section are divided into boot parameters and run-time parameters.

### Boot Parameters

Boot parameters are used by `tmboot` when it executes a server. Once running, a server reads its entry from the configuration file to determine its run-time options. The unique server identification number is used to find the right entry. The following are boot parameters.

`CLOPT=string_value`

The `CLOPT` parameter specifies a string of command-line options to be passed to `AOUT` when booted. The `servopts(5)` page in the *File Formats, Data Descriptions, MIBs, and System Processes Reference* lists the valid parameters.

Some of the available options apply primarily to servers under development. For example, the `-r` option directs the server to write a record to its standard error file each time a service request begins or ends.

Other command-line options can be used to direct the server's standard out (`stdout`) and standard error (`stderr`) to specific files, or to start the server so that it initially advertises a limited set of its available services.

The default value for the `CLOPT` parameter is `-A`, which means that the server is started with all available services advertised.

The maximum length of the `CLOPT` parameter value is 256 characters; it must be enclosed in double quotes.

`SEQUENCE=number`

This parameter specifies when to shut down or boot relative to other servers. If `SEQUENCE` is not specified, servers are booted in the order found in the `SERVERS` section (and shut down in the

reverse order). If some servers have sequence numbers specified and others do not, all servers with sequence numbers are booted first from low to high sequence number, then all servers without sequence numbers are booted in the order in which they appear in the configuration file. Sequence numbers range between 1 and 9999. If the same sequence number is assigned to more than one server, `tmboot` may boot those servers in parallel.

`MIN=number`

The `MIN` parameter specifies the minimum number of occurrences of the server to boot by `tmboot`. If an `RQADDR` is specified, and `MIN` is greater than 1, the servers form a multiple servers single queue (MSSQ) set. The identifiers for the servers are `SRVID` up to  $(SRVID + (MAX - 1))$ . All occurrences of the server have the same sequence numbers as well as any other server parameters. The value range for `MIN` is 0 to 1000. If `MIN` is not specified, the default value is 1.

`MAX=number`

The `MAX` parameter sets the maximum number of occurrences of the server to be booted. Initially, `tmboot` boots `MIN` servers, and additional servers can be booted up to `MAX` occurrences using the `-i` option of `tmboot` to specify the associated server identifier. The value range for `MAX` is 0 to 1000. If no value is specified for `MAX`, the default is the same as for `MIN`, or 1.

- `tmboot` starts `MIN` occurrences unless you explicitly call for more with the `-i SRVID` option of `tmboot`.
- If `RQADDR` is specified and `MIN` is greater than one, an MSSQ set is formed
- If `MIN` is not specified, the default is 1.
- If `MAX` is not specified, the default is `MIN`.
- `MAX` is especially important for conversational servers because they are spawned automatically as needed.

## Run-time Parameters

The server uses run-time parameters after it is started by `tmboot`. As indicated previously, `tmboot` uses the values found in the `TUXDIR`, `APPDIR` and `ENVFILE` parameters for the `MACHINES` section when booting the server. It also sets the `PATH` for the server to:

```
“APPDIR:TUXDIR/bin:/bin:path”
```

where `path` is the value of the last `PATH=` line appearing in the `ENVFILE` file. The following parameters are run-time parameters.

`ENVFILE=string_value`

You can use the `ENVFILE` parameter for a server to add values to the environment established by `tmboot` during initialization of the server. You can optionally set variables specified in the file named in the `SERVERS ENVFILE` parameter after you set those in the `MACHINES ENVFILE` used by `tmboot`. These files cannot be used to override `TUXDIR`, `APDIR`, `TUXCONFIG`, or `TUSOFFSET`. The best policy is to include in the server's `ENVFILE` only those variable assignments known to be needed to ensure proper running of the application.

On the server, the `ENVFILE` file is processed *after* the server starts. Therefore, it cannot be used to set the pathnames used to find executable or dynamically loaded files needed to execute the server. If you need to perform these tasks, use the machine `ENVFILE` instead.

Within `ENVFILE` only lines of the form

```
VARIABLE =string
```

are allowed. `VARIABLE` must start with an underscore or alphabetic character and can contain only underscore or alphanumeric characters. If the server is associated with a server group that can be migrated to a second machine, the `ENVFILE` must be in the same location on both machines.

```
CONV={Y | N}
```

`CONV` specifies whether the server is a conversational server. `CONV` takes a `Y` value if a conversational server is being defined. Connections can only be made to conversational servers. For a request/response server, you can either set `CONV=N`, which is the default, or omit the parameter.

```
RQADDR=string_value
```

`RQADDR` assigns a symbolic name to the request queue of this server. `MSSQ` sets are established by using the same symbolic name for more than one server (or by specifying `MIN` greater than 1). All members of an `MSSQ` set must offer an identical set of services and must be in the same server group.

If `RQADDR` is not specified, the system assigns a unique key to serve as the queue address for this server. However, `tmadmin` commands that take a queue address as an argument are easier to use if queues are given symbolic names.

```
RQPERM=number
```

Use the `RQPERM` parameter to assign UNIX-style permissions to the request queue for this server. The value of `number` can be between 0001 and 0777, inclusive. If no parameter is specified, the permissions value of the bulletin board, as specified by `PERM` in the `RESOURCES` section, is used. If no value is specified there, the default of 0666 is used (the default exposes your application to possible use by any login on the system, so consider this carefully).

```
REPLYQ={ Y | N }
```

The `REPLYQ` parameter specifies whether a reply queue, separate from the request queue, should be established for `AOUT`. If `N` is specified, the reply queue is created on the same `LMID` as the `AOUT`. If only one server is using the request queue, replies can be retrieved from the request queue without causing problems. However, if the server is a member of an `MSSQ` set and contains services programmed to receive reply messages, `REPLYQ` should be set to `Y` so that an individual reply queue is created for this server. If set to `N`, the reply is sent to the request queue shared by all servers for the `MSSQ` set, and you cannot ensure that the reply will be picked up by the server that is waiting for it.

It should be standard practice for all member servers of an `MSSQ` set to specify `REPLYQ=Y` if replies are anticipated. Servers in an `MSSQ` set are required to have identical offerings of services, so it is reasonable to expect that if one server in the set expects replies, any server in the set can also expect replies.

`RPPERM=number`

Use the `RPPERM` parameter to assign permissions to the reply queue. *number* is specified in the usual UNIX fashion (for example, 0600); the value can be between 0001 and 0777, inclusive. If `RPPERM` is not specified, the default value 0666 is used. This parameter is useful only when `REPLYQ=Y`. If requests and replies are read from the same queue, only `RQPERM` is needed; `RPPERM` is ignored.

`RESTART={ Y | N }`

The `RESTART` parameter takes a `Y` or `N` to indicate whether `AOUT` is restartable. The default is `N`. If the server is in a group that can be migrated, `RESTART` must be `Y`. A server started with a `SIGTERM` signal cannot be restarted; it must be rebooted.

An application's policy on restarting servers might vary according to whether the server is in production or not. During the test phase of application development it is reasonable to expect that a server might fail repeatedly, but server failures should be rare events once the application has been put into production. You might want to set more stringent parameters for restarting servers once the application is in production.

## Parameters Associated with `RESTART`

`RCMD=string_value`

If `AOUT` is restartable, this parameter specifies the command that should be executed when `AOUT` abnormally terminates. The string, up to the first space or tab, must be the name of an executable UNIX file, either a full pathname or relative to `APPDIR`. (Do not attempt to set a shell variable at the beginning of the command.) Optionally, the command name can be followed by command-line arguments. Two additional arguments are appended to the command line: the

GRPNO and SRVID associated with the restarting server. *string\_value* is executed in parallel with restarting the server.

You can use the RCMD parameter to specify a command to be executed in parallel with the restarting of the server. The command must be an executable UNIX system file residing in a directory on the server's PATH. An example is a command that sends a customized message to the userlog to mark the restarting of the server.

MAXGEN=*number*

If AOUT is restartable, this parameter specifies that it can be restarted at most (*number* - 1) times within the period specified by GRACE. The value must be greater than 0 and less than 256. If not specified, the default is 1 (which means that the server can be started once, but not restarted). If the server is to be restartable, MAXGEN must be equal to or greater than 2. RESTART must be Y or MAXGEN is ignored.

GRACE=*number*

If RESTART is Y, the GRACE parameter specifies the time period (in seconds) during which this server can be restarted, (MAXGEN - 1) times. The number assigned must be equal to or greater than 0, and less than 2,147,483,648 seconds (or a little more than 68 years). If GRACE is not specified the default is 86,400 seconds (24 hours). Setting GRACE to 0 removes all limitations; the server can be restarted an unlimited number of times.

## Entering Parameters

You can use Oracle Tuxedo parameters, including RESTART, RQADDR, and REPLYQ, with the JSL. (See *Administering an Oracle Tuxedo Application at Run Time* for additional information regarding run-time parameters.) Enter the following parameters:

1. To identify the SRVGRP parameter, type the previously defined group name value from the GROUPS section.
2. To indicate the SRVID, type a number between 1 and 30,000 that identifies the server within its group.
3. Verify that the syntax for the CLOPT parameter is as follows:

```
CLOPT= "-- -n 0x0002PPPPNNNNNNNN -d /dev/tcp -m2 -M4 -x10"
```

**Note:** The CLOPT parameters may vary. Refer to the table [“JSL Command-line Options” on page 3-8](#) for pertinent command-line information.

4. If necessary, type the optional parameters:

- Type the `SEQUENCE` parameter to determine the order that the servers are booted.
- Specify `Y` to permit release of the `RESTART` parameter.
- Type `0` to permit an infinite number of server restarts using the `GRACE` parameter.

## Sample Applications in Oracle Jolt Online Resources

You can access sample code that can be modified for use with Oracle Jolt through the Oracle Jolt product Web page at:

<http://www.bea.com/products/jolt/index.htm>

These samples demonstrate and utilize Oracle Jolt features and functionality.

Other Web sites with Java-related information include:

- Javasoft Home page (<http://www.java.sun.com/>)
- Newsgroups in the `comp.lang.java` hierarchy. These groups contain lists of past articles and communications regarding Java, and are a valuable source of archival material.

# Using the Jolt Class Library

The Oracle Jolt Class Library provides developers with a set of object-oriented Java language classes for accessing Oracle Tuxedo services. The class library contains the class files that implement the Jolt API. Using these classes, you can extend applications for Internet and intranet transaction processing. You can use the Jolt Class Library to customize access to Oracle Tuxedo services from Java applets.

This topic includes the following sections:

- [Class Library Functionality Overview](#)
- [Jolt Object Relationships](#)
- [Jolt Class Library Walkthrough](#)
- [Using Oracle Tuxedo Buffer Types with Jolt](#)
- [Multithreaded Applications](#)
- [Event Subscription and Notifications](#)
- [Clearing Parameter Values](#)
- [Reusing Objects](#)
- [Deploying and Localizing Jolt Applets](#)
- [Using SSL](#)

To use the information in the following sections, you need to be generally familiar with the Java programming language and object-oriented programming concepts. All the programming examples are in Java code.

**Note:** All program examples are only fragments used to illustrate Jolt capabilities. They are not intended to be compiled and run as provided. These program examples require additional code to be fully executable.

## Class Library Functionality Overview

The Jolt Class Library gives the Oracle Tuxedo application developer the tools to develop client-side applications or applets that run as independent Java applications or in a Java-enabled Web browser. The `bea.jolt` package contains the Jolt Class Library. To use the Jolt Class Library, the client program or applet must import this package. For an example of how to import the `bea.jolt` package, refer to the listing “[Jolt Transfer of Funds Example \(SimXfer.java\)](#)” on [page 4-11](#).

## Java Applications Versus Java Applets

Java programs that run in a browser are called applets. Applets are small, easily downloaded parts of an overall application that perform specific functions. Many popular browsers impose limitations on the capabilities of Java applets in order to provide a high degree of security for the users of the browser. Applets have the following restrictions:

- An applet ordinarily cannot read or write files on any host system.
- An applet cannot start any program on the host (client) that is executing the applet.
- An applet can make a network connection only to the host from which the applet originated; it cannot make other network connections, not even to the client machine.

Programming workarounds exist for most restrictions on Java applets. Check your browser’s Web site (for example, [www.netscape.com](http://www.netscape.com) or [www.microsoft.com](http://www.microsoft.com)) or developer documentation for specific information about the applet capabilities that the browser supports or restricts. You can also use Jolt Relay to work around some of the network connection restrictions.

A Java application, however, is not run in the context of a browser and is not restricted in the same ways. For example, a Java application can start another application on the host machine where it is executing. While an applet relies on the windowing environment of a browser or appletviewer for much of its user interface, a Java application requires that you create your own user interface. An applet is designed to be small and highly portable. A Java application, on the other hand, can

operate much like any other non-Java program. The security restrictions for applets imposed by various browsers and the scope of the two program types are the most important differences between a Java application and a Java applet.

## Jolt Class Library Features

The Jolt Class Library has the following characteristics:

- Features fully thread-safe classes.
- Encapsulates typical transaction functions such as logon, synchronous calling, transaction begin, commit, rollback, and logoffs as Java objects.
- Contains methods that allow you to set idle timeouts for continuous and intermittent client network connections.
- Features methods that allow a Jolt client to subscribe to and receive event-based notifications.

## Error and Exception Handling

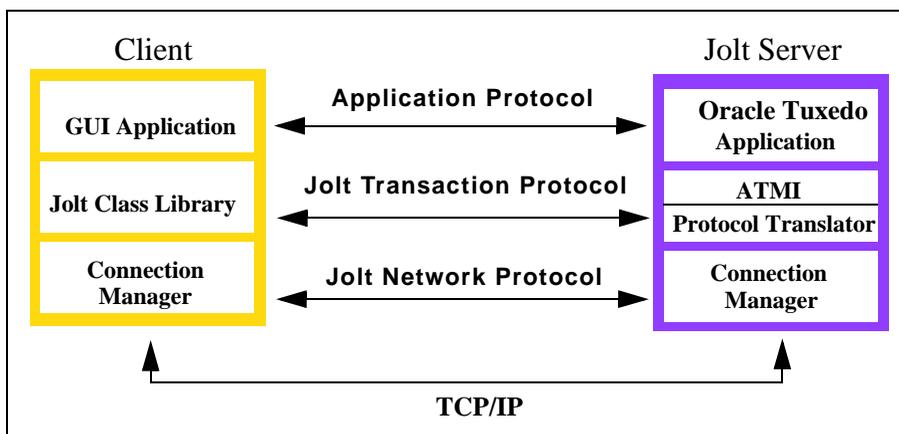
The Jolt Class Library returns both Jolt interpreter and Oracle Tuxedo errors as exceptions. The Jolt Class Library Reference contains the Jolt classes and lists the errors or exceptions thrown for each class. The *Oracle Jolt API Reference* contains the Error and Exception Class Reference.

## Jolt Client/Server Relationship

Oracle Jolt works in a distributed client/server environment and connects Java clients to Oracle Tuxedo-based applications.

Figure 4-1 illustrates the client/server relationship between a Jolt program and the Jolt Server.

Figure 4-1 Jolt Client/Server Relationship



As illustrated in the figure, the Jolt Server acts as a proxy for a native Oracle Tuxedo client, implementing functionality available through the native Oracle Tuxedo client. The Oracle Jolt Server accepts requests from Oracle Jolt clients and maps those requests into Oracle Tuxedo service requests through the Oracle Tuxedo ATMI interface. Requests and associated parameters are packaged into a message buffer and delivered over the network to the Oracle Jolt Server. The Oracle Jolt Connection Manager handles all communication between the Oracle Jolt Server and the Oracle Jolt applet using the Oracle Jolt Transaction Protocol. The Oracle Jolt Server unpacks the data from the message, performs any necessary data conversions, such as numeric format conversions or character set conversions, and makes the appropriate service request to Oracle Tuxedo as specified by the message.

Once a service request enters the Oracle Tuxedo system, it is executed in exactly the same manner as any other Oracle Tuxedo request. The results are returned through the ATMI interface to the Oracle Jolt Server, which packages the results and any error information into a message that is sent to the Oracle Jolt client applet. The Oracle Jolt client then maps the contents of the message into the various Oracle Jolt client interface objects, completing the request.

On the client side, the user program contains the client application code. The Jolt Class Library packages a `JoltSession` and `JoltTransaction`, which in turn handle service requests.

[Table 4-1](#) describes the client-side requests and Jolt Server-side actions in a simple example program.

**Table 4-1 Jolt Client/Server Interaction**

Jolt Client	Jolt Server
1 <code>attr=new JoltSessionAttributes();</code> <code>attr.setString(attr.APPADDRESS,</code> <code>"/myhost:8000");</code>	Binds the client to the Oracle Tuxedo environment
2 <code>session=new JoltSession(attr, username,</code> <code>userRole, userPassword, appPassword);</code>	Logs the client onto Oracle Tuxedo
3 <code>withdrawal=new JoltRemoteService(</code> <code>servname, session );</code>	Looks up the service attributes in the Repository
4 <code>withdrawal.addString("accountnumber",</code> <code>"123");</code> <code>withdrawal.addFloat("amount", (float)</code> <code>100.00);</code>	Populates variables in the client (no Jolt Server activity)
5 <code>trans=new JoltTransaction( time-out,</code> <code>session);</code>	Begins a new Tuxedo transaction
6 <code>withdrawal.call(trans);</code>	Executes the Oracle Tuxedo service
7 <code>trans.commit() or trans.rollback();</code>	Completes or rolls back transaction
8 <code>balance=withdrawal.getFloatDef("balance,"</code> <code>(float) 0.0);</code>	Retrieves the results (no Jolt Server activity)
9 <code>session.endSession();</code>	Logs the client off of Oracle Tuxedo

The following tasks summarize the interaction shown in the previous table, [“Jolt Client/Server Interaction.”](#)

1. Bind the client to the Oracle Tuxedo environment using the `JoltSessionAttributes` class.

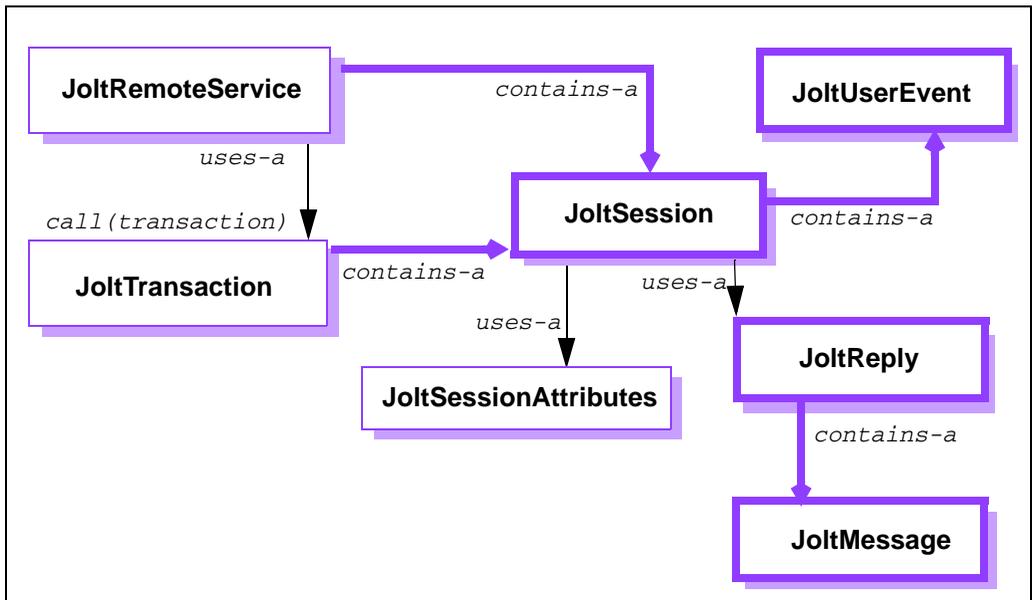
2. Establish a session.
3. Set variables.
4. Perform the necessary transaction processing.
5. Log the client off of the Oracle Tuxedo system.

Each of these activities is handled through the use of the Jolt Class Library classes. These classes include methods for setting and clearing data and for handling remote service actions. [“Jolt Object Relationships” on page 4-7](#) describes the Jolt Class Library classes in more detail.

# Jolt Object Relationships

Figure 4-2 illustrates the relationship between the instantiated objects of the Jolt Class Library classes.

Figure 4-2 Jolt Object Relationships



As objects, the Jolt classes interact in various relationships with each other. In the previous figure, the relationships are divided into three basic categories:

- Contains-a relationship—at the class level an object can contain other objects. For example, a **JoltTransaction** stores (or contains) a **JoltSession** object.
- Is-a relationship—the is-a relationship usually occurs at the class instance or sub-object level and denotes that the object is an instance of a particular object.
- Uses-a relationship—an object can use another object without containing it. For example, a **JoltSession** can use the **JoltSessionAttributes** object to obtain the host and port information.

# Jolt Class Library Walkthrough

Use Jolt classes to perform the basic functions of transaction processing: logon/logoff; synchronous service calling; transaction begin, commit, and rollback. The following sections describe how Jolt classes are used to perform these functions.

- [Logon and Logoff](#)
- [Synchronous Service Calling](#)
- [Transaction Begin, Commit, and Rollback](#)

You can also use the Jolt class library to develop multithreaded applications, define Tuxedo buffer types, and subscribe to events and unsolicited messages. These functions are discussed in later sections.

## Logon and Logoff

The client application must log on to the Oracle Tuxedo environment prior to initiating any transaction activity. The Jolt Class Library provides the `JoltSessionAttributes` class and `JoltSession` class to establish a connection to an Oracle Tuxedo system.

The `JoltSessionAttributes` class will contain the connection properties of Jolt and Oracle Tuxedo systems as well as various other properties of the two systems. To establish a connection, the client application must create an instance of the `JoltSession` class. This instance is the `JoltSession` object. After the developer instantiates a Jolt Session and Oracle Tuxedo object, the Jolt and Oracle Tuxedo logon capability is enabled. Calling the `endSession` method ends the session and allows the user to log off.

## Synchronous Service Calling

Transaction activities such as requests and replies are handled through a `JoltRemoteService` object (an instance of the `JoltRemoteService` class). Each `JoltRemoteService` object refers to an exported Oracle Tuxedo request/reply service. You must provide a service name and a `JoltSession` object to instantiate a `JoltRemoteService` object before it can be used.

To use a `JoltRemoteService` object:

1. Set the input parameters.
2. Invoke the service.
3. Examine the output parameters.

For efficiency, Jolt does not make a copy of any input parameter object; only the references to the object (for example, string and byte array) are saved. Because `JoltRemoteService` object is a stateful object, its input parameters and the request attributes are retained throughout the life of the object. You can use the `clear()` method to reset the attributes and input parameters before reusing the `JoltRemoteService` object.

Because Jolt is designed for a multithreaded environment, you can invoke multiple `JoltRemoteService` objects simultaneously by using the Java multithreading capability. Refer to [“Multithreaded Applications” on page 4-35](#) for additional information.

## Transaction Begin, Commit, and Rollback

In Jolt, a transaction is represented as an object of the class `JoltTransaction`. The transaction begins when the transaction object is instantiated. The transaction object is created with a timeout and `JoltSession` object parameter:

```
trans = new JoltTransaction(timeout, session)
```

Jolt uses an explicit transaction model for any services involved in a transaction. The transaction service invocation requires a `JoltTransaction` object as a parameter. Jolt also requires that the service and the transaction belong to the same session. Jolt does *not* allow you to use services and transactions that are not bound to the same session.

The sample code in the listing [“Jolt Transfer of Funds Example \(SimXfer.java\)” on page 4-11](#) describes how to use the Jolt Class Library and includes the use of the `JoltSessionAttributes`, `JoltSession`, `JoltRemoteService`, and `JoltTransaction` classes.

The same sample combines two user-defined Oracle Tuxedo services (`WITHDRAWAL` and `DEPOSIT`) to perform a simulated `TRANSFER` transaction. If the `WITHDRAWAL` operation fails, a rollback is performed. Otherwise, a `DEPOSIT` is performed and a commit completes the transaction.

The following programming steps describe the transaction process shown in the sample code listing [“Jolt Transfer of Funds Example \(SimXfer.java\)” on page 4-11](#):

1. Set the connection attributes like `hostname` and `portnumber` in the `JoltSessionAttribute` object.

Refer to this line in the following code listing:

```
sattr = new JoltSessionAttributes();
```

2. The `sattr.checkAuthenticationLevel()` allows the application to determine the level of security required to log on to the server.

Refer to this line in the following code listing:

```
switch (sattr.checkAuthenticationLevel())
```

3. The logon is accomplished by instantiating a `JoltSession` object.

Refer to these lines in the following code listing:

```
session = new JoltSession (sattr, userName, userRole,  
userPassword, appPassword);
```

This example does not explicitly catch `SessionException` errors.

4. All `JoltRemoteService` calls require a service to be specified and the session key returned from `JoltSession()`.

Refer to these lines in the following code listing:

```
withdrawal = new JoltRemoteService("WITHDRAWAL", session);  
deposit = new JoltRemoteService("DEPOSIT", session);
```

These calls bind the service definition of both the `WITHDRAWAL` and `DEPOSIT` services, which are stored in the Oracle Tuxedo Service Metadata Repository, to the `withdrawal` and `deposit` objects, respectively. The services `WITHDRAWAL` and `DEPOSIT` must be defined in the Metadata Repository; otherwise a `ServiceException` is thrown. This example does not explicitly catch `ServiceException` errors.

5. Once the service definitions are returned, the application-specific fields such as account number `ACCOUNT_ID` and withdrawal amount `SAMOUNT` are automatically populated.

Refer to these lines in the following code listing:

```
withdrawal.addInt("ACCOUNT_ID", 100000);  
withdrawal.addString("SAMOUNT", "100.00");
```

The `add*()` methods can throw `IllegalAccessError` or `NoSuchFieldError` exceptions.

6. The `JoltTransaction` call allows a timeout to be specified if the transaction does not complete within the specified time.

Refer to this line in the following code listing:

```
trans = new JoltTransaction(5,session);
```

7. Once the withdrawal service definition is automatically populated, the withdrawal service is invoked by calling the `withdrawal.call(trans)` method.

Refer to this line in the following code listing:

```
withdrawal.call(trans);
```

8. A failed WITHDRAWAL can be rolled back.

Refer to this line in the following code listing:

```
trans.rollback();
```

9. Otherwise, once the DEPOSIT is performed, all the transactions are committed. Refer to these lines in the following code listing:

```
deposit.call(trans);
```

```
trans.commit();
```

[Listing 4-1](#) shows an example of a simple application for the transfer of funds using the Jolt classes.

#### **Listing 4-1 Jolt Transfer of Funds Example (SimXfer.java)**

---

```
/* Copyright 1999 Oracle Systems, Inc. All Rights Reserved */
import bea.jolt.*;
public class SimXfer
{
    public static void main (String[] args)
    {
        JoltSession session;
        JoltSessionAttributes sattr;
        JoltRemoteService withdrawal;
        JoltRemoteService deposit;
        JoltTransaction trans;
        String userName=null;
        String userPassword=null;
        String appPassword=null;
        String userRole="myapp";

        sattr = new JoltSessionAttributes();
        sattr.setString(sattr.APPADDRESS, "//bluefish:8501");

        switch (sattr.checkAuthenticationLevel())
        {
            case JoltSessionAttributes.NOAUTH:
                System.out.println("NOAUTH\n");
                break;
```

```

case JoltSessionAttributes.APPASSWORD:
    appPassword = "appPassword";
    break;
case JoltSessionAttributes.USRPASSWORD:
    userName = "myname";
    userPassword = "mysecret";
    appPassword = "appPassword";
    break;
}
sattr.setInt(sattr.IDLETIMEOUT, 300);
session = new JoltSession(sattr, userName, userRole,
userPassword, appPassword);
// Simulate a transfer
withdrawal = new JoltRemoteService("WITHDRAWAL", session);
deposit = new JoltRemoteService("DEPOSIT", session);

withdrawal.addInt("ACCOUNT_ID", 100000);
withdrawal.addString("SAMOUNT", "100.00");

// Begin the transaction w/ a 5 sec timeout
trans = new JoltTransaction(5, session);
try
{
    withdrawal.call(trans);
}
catch (ApplicationException e)
{
    e.printStackTrace();
    // This service uses the STATLIN field to report errors
    // back to the client application.
    System.err.println(withdrawal.getStringDef("STATLIN", "NO
STATLIN"));
    System.exit(1);
}

String wbal = withdrawal.getStringDef("SBALANCE", "$-1.0");

// remove leading "$" before converting string to float
float w = Float.valueOf(wbal.substring(1)).floatValue();
if (w < 0.0)

```

```
{
    System.err.println("Insufficient funds");
    trans.rollback();
    System.exit(1);
}
else // now attempt to deposit/transfer the funds
{
    deposit.addInt("ACCOUNT_ID", 100001);
    deposit.addString("SAMOUNT", "100.00");

    deposit.call(trans);
    String dbal = deposit.getStringDef("SBALANCE", "-1.0");
    trans.commit();

    System.out.println("Successful withdrawal");
    System.out.println("New balance is: " + wbal);

    System.out.println("Successful deposit");
    System.out.println("New balance is: " + dbal);
}

session.endSession();
System.exit(0);
} // end main
} // end SimXfer
```

---

# Using Oracle Tuxedo Buffer Types with Jolt

Jolt supports the following built-in Oracle Tuxedo buffer types:

- FML, FML32
- VIEW, VIEW32
- X\_COMMON
- X\_C\_TYPE
- CARRAY
- X\_OCTET
- STRING
- XML
- MBSTRING

**Note:** X\_OCTET is used identically to CARRAY.

X\_COMMON and X\_C\_TYPE are used identically to VIEW.

Of the Oracle Tuxedo built-in buffer types, the Jolt programmer should be particularly aware of how Jolt handles the CARRAY (character array) and STRING buffer types:

- The CARRAY type is used to handle data opaquely (that is, the characters of a CARRAY data type are not interpreted in any way). Therefore, no data conversion is performed between a Jolt client and Oracle Tuxedo service.
- The STRING data type is character and, unlike CARRAY, you can determine its transmission length by counting the number of characters in the buffer until reaching the null character. Therefore, data is automatically converted when data is exchanged by machines with different character sets.

For more information about all the Oracle Tuxedo typed buffers, data types, and buffer types, refer to the following documents:

- *Programming Oracle Tuxedo ATMI Applications Using C*
- *Oracle Tuxedo ATMI C Function Reference*
- *Oracle Tuxedo ATMI FML Function Reference*
- *File Formats, Data Descriptions, MIBs, and System Processes Reference*

## Using the STRING Buffer Type

The `STRING` buffer type is an array of non-null characters that terminates with a null character. Unlike `CARRAY`, you can determine its transmission length by counting the number of characters in the buffer until reaching the null character. Since the `STRING` buffer is self-describing, the Oracle Tuxedo System can convert data automatically when data is exchanged by machines with different character sets.

**Note:** During the data conversion from Jolt to `STRING`, the null terminator is automatically appended to the end of the `STRING` buffers because a Java string is not null-terminated.

Using the `STRING` buffer type requires two main steps:

1. Define the Tuxedo service that you will be using with the buffer type.
2. Write the code that uses the `STRING` buffer type.

The next two sections provide examples that demonstrate these steps.

The `ToUpper` code fragment shown in the listing [“Use of the STRING Buffer Type \(ToUpper.java\)”](#) on page 4-18 illustrates how Jolt works with a service whose buffer type is `STRING`. The `ToUpper` Oracle Tuxedo Service is available in the Oracle Tuxedo `simpapp` example.

### Define TOUPPER in the Repository Editor

Before running the `ToUpper.java` example, you need to define the `TOUPPER` service through the Metadata Repository Editor.

### Tuxedo Services Metadata

#### Services

Service name	SOAP	REST
GETX_CTYPE		
GETX_COMMON		
CAR_X_COMMON		
FML32_CAR		
VIEW_STR		
X_COMMON_STR		
CAR_FML32		
CARRAY		
X_OCTET_MBSTRING		
MBSTRING_VIEW		

1 2 3 4 5 ... 7 >

Delete Selected Service

#### Service Details

Save Create New Service

\* Service TOUPPER

\* Tuxedo Service TOUPPER Service Description

Export Y

Service Type request-response

Input Buffer STRING

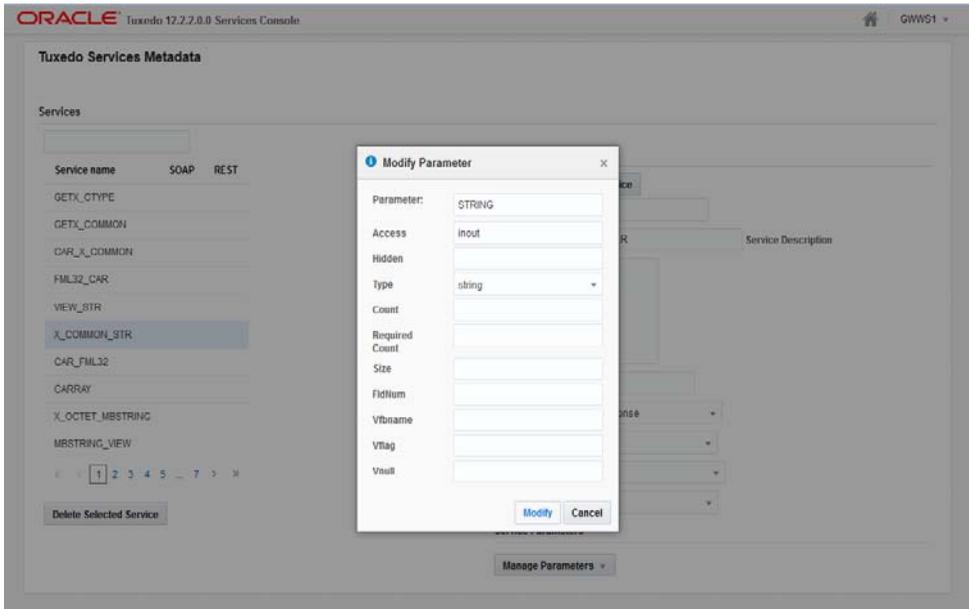
Output Buffer STRING

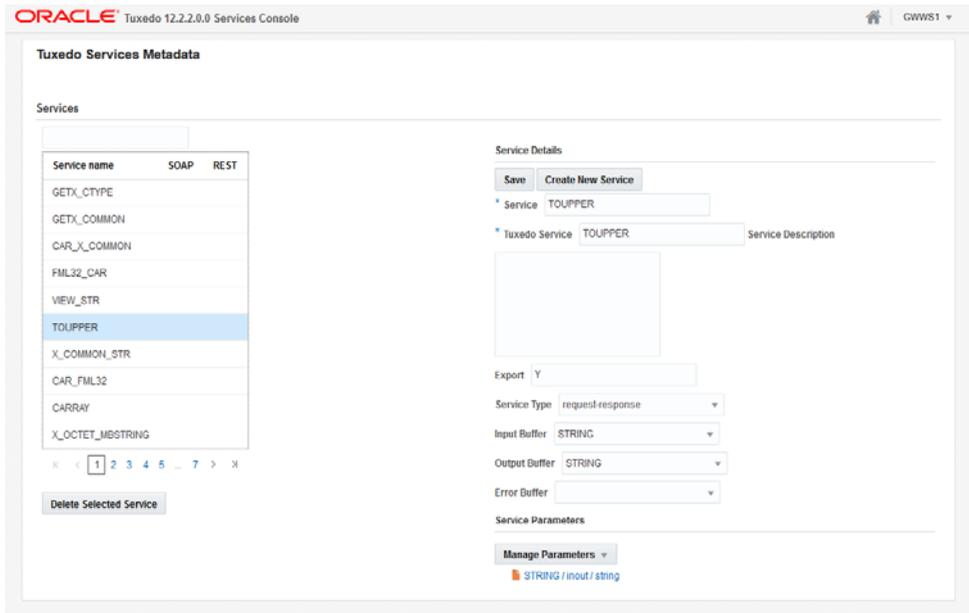
Error Buffer

#### Service Parameters

Manage Parameters

- Modify After
- Remove Before
- Add Child





## ToUpper.java Client Code

The `ToUpper.java` Java code fragment in [Listing 4-2](#) illustrates how Jolt works with a service with a buffer type of `STRING`. The example shows a Jolt client using a `STRING` buffer to pass data to a server. The Oracle Tuxedo server would take the buffer, convert the string to all uppercase letters, and pass the string back to the client. The following example assumes that a session object was already instantiated.

### Listing 4-2 Use of the `STRING` Buffer Type (`ToUpper.java`)

```

/* Copyright 1996 Oracle Systems, Inc. All Rights Reserved */
import bea.jolt.*;
public class ToUpper
{
    public static void main (String[] args)
    {
        JoltSession          session;
        JoltSessionAttributes sattr;
    }
}

```

```

JoltRemoteService      toupper;
JoltTransaction        trans;
String userName=null;
String userPassword=null;
String appPassword=null;
String userRole="myapp";
String outstr;

sattr = new JoltSessionAttributes();
sattr.setString(sattr.APPADDRESS, "//myhost:8501");

switch (sattr.checkAuthenticationLevel())
{
case JoltSessionAttributes.NOAUTH:
    break;
case JoltSessionAttributes.APPASSWORD:
    appPassword = "appPassword";
    break;
case JoltSessionAttributes.USRPASSWORD:
    userName = "myname";
    userPassword = "mysecret";
    appPassword = "appPassword";
    break;
}
sattr.setInt(sattr.IDLETIMEOUT, 300);
session = new JoltSession(sattr, userName, userRole,
userPassword, appPassword);
toupper = new JoltRemoteService ("TOUPPER", session);
toupper.setString("STRING", "hello world");
toupper.call(null);
outstr = toupper.getStringDef("STRING", null);
if (outstr != null)
    System.out.println(outstr);

session.endSession();
    System.exit(0);
} // end main
} // end ToUpper

```

## Using the CARRAY Buffer Type

The CARRAY buffer type is a simple character array buffer type that is built into the Oracle Tuxedo system. Because the system does not interpret the data (although the data type is known) when you use the CARRAY buffer type, you must specify a data length in the Jolt client application. The Jolt client must specify a data length when passing this buffer type.

For example, if an Oracle Tuxedo service uses a CARRAY buffer type and the user sets a 32-bit integer (in Java the integer is in big-endian byte order), then the data is sent unmodified to the Oracle Tuxedo service.

To use the CARRAY buffer type, you first define the Tuxedo service that you will be using with the buffer type. Then, write the code that uses the buffer type. The next two sections demonstrate these steps.

**Note:** X\_OCTET is used identically to CARRAY.

## Define the Tuxedo Service in the Repository Editor

Before running the ECHO example, you must write and boot a Tuxedo ECHO service. The ECHO service takes a buffer and passes it back to the Jolt client. You need to define the ECHO service in the Metadata Repository Editor (see [Define TOUPPER in the Repository Editor](#) for an example).

## tryOnCARRAY.java Client Code

The code in the following listing illustrates how Jolt works with a service with a buffer type of CARRAY. Because Jolt does not look into the CARRAY data stream, it is the programmer's responsibility to ensure that the data formats between the Jolt client and the CARRAY service match. The example in [Listing 4-3](#) assumes that a session object was already instantiated.

### Listing 4-3 CARRAY Buffer Type Example

---

```
/* Copyright 1996 Oracle Systems, Inc. All Rights Reserved */

/* This code fragment illustrates how Jolt works with a service
 * whose buffer type is CARRAY.
 */

import java.io.*;
import bea.jolt.*;
class ...
```

```

{
    ...
public void tryOnCARRAY()
{
    byte data[];
    JoltRemoteService csvc;
    DataInputStream din;
    DataOutputStream dout;
    ByteArrayInputStream bin;
    ByteArrayOutputStream bout;
    /*
     * Use java.io.DataOutputStream to put data into a byte array
     */
    bout = new ByteArrayOutputStream(512);
    dout = new DataOutputStream(bout);
    dout.writeInt(100);
    dout.writeFloat((float) 300.00);
    dout.writeUTF("Hello World");
    dout.writeShort((short) 88);
    /*
     * Copy the byte array into a new byte array "data". Then
     * issue the Jolt remote service call.
     */
    data = bout.toByteArray();
    csvc = new JoltRemoteService("ECHO", session);
    csvc.setBytes("CARRAY", data, data.length);
    csvc.call(null);
    /*
     * Get the result from JoltRemoteService object and use
     * java.io.DataInputStream to extract each individual value
     * from the byte array.
     */
    data = csvc.getBytesDef("CARRAY", null);
    if (data != null)
    {
        bin = new ByteArrayInputStream(data);
        din = new DataInputStream(bin);
        System.out.println(din.readInt());
    }
}

```

```
        System.out.println(din.readFloat());
        System.out.println(din.readUTF());
        System.out.println(din.readShort());
    }
}
}
```

---

## Using the FML Buffer Type

FML (Field Manipulation Language) is a flexible data structure that can be used as a typed buffer. The FML data structure stores tagged values that are typed, variable in length, and may have multiple occurrences. The typed buffer is treated as an abstract data type in FML.

FML gives you the ability to access and update data values without having to know how the data is structured and stored. In your application program, you simply access or update a field in the fielded buffer by referencing its identifier. To perform the operation, the FML run time determines the field location and data type.

FML is especially suited for use with Jolt clients because the client and server code can be in two languages (for example, Java and C); the client/server platforms can have different data type specifications; or the interface between the client and the server can change frequently.

The following `tryOnFml` examples illustrate the use of the FML buffer type. The examples show a Jolt client using FML buffers to pass data to a server. The server takes the buffer, creates a new FML buffer to store the data, and passes that buffer back to the Jolt client. The examples consist of the following components.

- The [“tryOnFml.java Code Example” on page 4-23](#) is a Jolt client that contains a PASSFML service.
- The [“tryOnFml.f16 Field Definitions” on page 4-24](#) is an Oracle Tuxedo FML field definitions table used by the PASSFML service.
- The [“tryOnFml.c Code Example” on page 4-24](#) is a server code fragment that contains the server side C code for handling the data sent by the Jolt client.

## tryOnFml.java Client Code

The `tryOnFml.java` Java code fragment in [Listing 4-4](#) illustrates how Jolt works with a service whose buffer type is FML. In this example, it is assumed that a session object was already instantiated.

### Listing 4-4 tryOnFml.java Code Example

---

```

/* Copyright 1997 Oracle Systems, Inc. All Rights Reserved */

import bea.jolt.*;
class ...
{
    ...
    public void tryOnFml ()
    {
        JoltRemoteService passFml;
        String outputString;
        int outputInt;
        float outputFloat;
        ...
        passFml = new JoltRemoteService("PASSFML", session);
        passFml.setString("INPUTSTRING", "John");
        passFml.setInt("INPUTINT", 67);
        passFml.setFloat("INPUTFLOAT", (float)12.0);
        passFml.call(null);
        outputString = passFml.getStringDef("OUTPUTSTRING", null);
        outputInt = passFml.getIntDef("OUTPUTINT", -1);
        outputFloat = passFml.getFloatDef("OUTPUTFLOAT", (float)-1.0);
        System.out.print("String =" + outputString);
        System.out.print(" Int =" + outputInt);
        System.out.println(" Float =" + outputFloat);
    }
}

```

---

## FML Field Definitions

The entries in the following listing, [“tryOnFml.f16 Field Definitions,”](#) show the FML field definitions for the previous listing, [“tryOnFml.java Code Example.”](#)

#### Listing 4-5 tryOnFml.f16 Field Definitions

---

```
#
# FML field definition table
#
*base      4100
INPUTSTRING 1    string
INPUTINT    2    long
INPUTFLOAT  3    float
OUTPUTSTRING 4    string
OUTPUTINT   5    long
OUTPUTFLOAT 6    float
```

---

## Define PASSFML in the Repository Editor

The BULKPKG package contains the PASSFML service, which is used with the `tryOnFml.java` and `tryOnFml.c` code. Before running the `tryOnFml.java` example, you need to modify the PASSFML service through the Metadata Repository Editor (see [Define TOUPPER in the Repository Editor](#) for an example).

## tryOnFml.c Server Code

[Listing 4-6](#) illustrates the server side code for using the FML buffer type. The PASSFML service reads in an input FML buffer and outputs a FML buffer.

#### Listing 4-6 tryOnFml.c Code Example

---

```
/*
 * tryOnFml.c
 *
 * Copyright (c) 1997 Oracle Systems, Inc. All rights reserved
 *
 * Contains the PASSFML Oracle Tuxedo server.
 */
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
```

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <malloc.h>
#include <math.h>
#include <string.h>
#include <fml.h>
#include <fml32.h>
#include <Usysflds.h>
#include <atmi.h>
#include <userlog.h>
#include "tryOnFml.f16.h"
/*
 * PASSFML service reads in a input fml buffer and outputs a fml buffer.
 */
void
PASSFML( TPSVCINFO *rqst )
{
    FLDDLEN len;
    FBFR *svcinfol = (FBFR *) rqst->data;
    char inputString[256];
    long inputInt;
    float inputFloat;
    FBFR *fml_ptr;
    int rt;
    if (Fget(svcinfo, INPUTSTRING, 0, inputString, &len) < 0) {
        (void)userlog("Fget of INPUTSTRING failed %s",
            Fstrerror(Error));
        tpreturn(TPFAIL, 0, rqst->data, 0L, 0);
    }
    if (Fget(svcinfo, INPUTINT, 0, (char *) &inputInt, &len) < 0) {
        (void)userlog("Fget of INPUTINT failed %s", Fstrerror(Error));
        tpreturn(TPFAIL, 0, rqst->data, 0L, 0);
    }
    if (Fget(svcinfo, INPUTFLOAT, 0, (char *) &inputFloat, &len) < 0) {
        (void)userlog("Fget of INPUTFLOAT failed %s",
            Fstrerror(Error));
        tpreturn(TPFAIL, 0, rqst->data, 0L, 0);
    }
    /* We could just pass the FML buffer back as is, put lets*/
    /* store it into another FML buffer and pass it back.*/
    if ((fml_ptr = (FBFR *) tmalloc("FML", NULL, rqst->len)) == (FBFR *) NULL) {
        (void)userlog("tpalloc failed in PASSFML %s",
            tpstrerror(tperrno));
        tpreturn(TPFAIL, 0, rqst->data, 0L, 0);
    }
    if (Fadd(fml_ptr, OUTPUTSTRING, inputString, (FLDDLEN)0) == -1) {
        userlog("Fadd failed with error: %s", Fstrerror(Error));
    }
}

```

```

        tpfree((char *)fml_ptr);
        tpreturn(TPFAIL, 0, NULL, 0L, 0);
    }
    if(Fadd(fml_ptr, OUTPUTINT, (char *)&inputInt, (FLDLEN)0) == -1) {
        userlog("Fadd failed with error: %s", Fstrerror(Ferror));
        tpfree((char *)fml_ptr);
        tpreturn(TPFAIL, 0, NULL, 0L, 0);
    }
    if(Fadd(fml_ptr, OUTPUTFLOAT, (char *)&inputFloat, (FLDLEN)0) == -1) {
        userlog("Fadd failed with error: %d\n", Fstrerror(Ferror));
        tpfree((char *)fml_ptr);
        tpreturn(TPFAIL, 0, NULL, 0L, 0);
    }
    tpreturn(TPSUCCESS, 0, (char *)fml_ptr, 0L, 0);
}

```

---

## Using the VIEW Buffer Type

VIEW is a built-in Oracle Tuxedo typed buffer. The VIEW buffer provides a way to use C structures and COBOL records with the Oracle Tuxedo system. The VIEW typed buffer enables the Oracle Tuxedo run-time system to understand the format of C structures and COBOL records based on the view description that is read at run time.

When allocating a VIEW, 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 parameter name must match the field name in that view. Because the Oracle Tuxedo run-time system can determine the space needed based on the structure size, your application need not provide a buffer length. The run-time system can also automatically handle such things as computing how much data to send in a request or response, and handle encoding and decoding when the message transfers between different machine types.

The following examples show the use of the VIEW buffer type with a Jolt client and its server-side application.

- The [“simpview.java Code Example” on page 4-27](#) is the Jolt client that contains the code used to connect to Oracle Tuxedo and uses the VIEW buffer type.
- The listing [“simpview.v16 Field Definitions” on page 4-28](#) contains the Oracle Tuxedo VIEW field definitions.
- The [“simpview.c Code Example” on page 4-29](#) contains the server side C code for handling the input from the Jolt client.

The Jolt client treats a null character in a VIEW buffer string format as an end-of-line character and truncates any part of the string that follows the null.

## Define VIEW in the Repository Editor

Before running the `simpview.java` and `simpview.c` examples, you need to define the SIMPVIEW service through the Metadata Repository Editor (see [Define TOUPPER in the Repository Editor](#) for an example).

## simpview.java Client Code

The listing “[simpview.java Code Example](#)” on page 4-27 illustrates how Jolt works with a service whose buffer type is VIEW. The client code is identical to the code used for accessing an FML service.

**Note:** The code in the following listing does not catch any exceptions. Because all Jolt exceptions are derived from `java.lang.RuntimeException`, the Java Virtual Machine (JVM) catches these exceptions if the application does not. (A well-written application will catch these exceptions and take appropriate actions.)

Before running the example in the following listing, you need to add the VIEW service to the SIMPAPP package using the Metadata Repository Editor and write the `simpview.c` Oracle Tuxedo application. This service takes the data from the client VIEW buffer, creates a new buffer and passes it back to the client as a new VIEW buffer. The following example assumes that a session object has already been instantiated.

### Listing 4-7 simpview.java Code Example

---

```

/* Copyright 1997 Oracle Systems, Inc. All Rights Reserved */
/*
 * This code fragment illustrates how Jolt works with a service whose buffer
 * type is VIEW.
 */
import bea.jolt.*;
class ...
{
    ...
    public void simpview ()
    {
        JoltRemoteService ViewSvc;
        String outString;
        int outInt;
        float outFloat;
    }
}

```

```

    // Create a Jolt Service for the Oracle Tuxedo service "SIMPVIEW"
    ViewSvc = new JoltRemoteService("SIMPVIEW",session);
    // Set the input parameters required for SIMPVIEW
    ViewSvc.setString("inString", "John");
    ViewSvc.setInt("inInt", 10);
    ViewSvc.setFloat("inFloat", (float)10.0);
    // Call the service. No transaction required, so pass
    // a "null" parameter
    ViewSvc.call(null);
    // Process the results
    outString = ViewSvc.getStringDef("outString", null);
    outInt = ViewSvc.getIntDef("outInt", -1);
    outFloat = ViewSvc.getFloatDef("outFloat", (float)-1.0);
    // And display them...
    System.out.print("outString=" + outString + ",");
    System.out.print("outInt=" + outInt + ",");
    System.out.println("outFloat=" + outFloat);
}
}

```

---

## VIEW Field Definitions

The “[simpview.v16 Field Definitions](#)” listing shows the Oracle Tuxedo VIEW field definitions for the `simpview.java` example that were shown in the previous listing.

### Listing 4-8 simpview.v16 Field Definitions

---

```

#
# VIEW for SIMPVIEW. This view is used for both input and output. The
# service could also have used separate input and output views.
# The first 3 params are input params, the second 3 are outputs.
#
VIEW SimpView
$
#type  cname          ffname count  flag   size  null
string inString      -      1      -     32    -
long   inInt         -      1      -     -     -
float  inFloat       -      1      -     -     -
string outString     -      1      -     32    -
long   outInt        -      1      -     -     -

```

```
float outFloat      -      1      -      -      -
END
```

---

## simpview.c Server Code

In [Listing 4-9](#), the input and output buffers are VIEW. The code accepts the VIEW buffer data as input and outputs the same data as VIEW.

### Listing 4-9 simpview.c Code Example

---

```
/*
 * SIMPVIEW.c
 *
 * Copyright (c) 1997 Oracle Systems, Inc. All rights reserved
 *
 * Contains the SIMPVIEW Oracle Tuxedo server.
 *
 */
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <malloc.h>
#include <math.h>
#include <string.h>
#include <fml.h>
#include <fml32.h>
#include <Usysflds.h>
#include <atmi.h>
#include <userlog.h>
#include "simpview.h"
/*
 * Contents of simpview.h.
 *
 *struct SimpView {
 *
 *    char    inString[32];
 *    long    inInt;
 *    float   inFloat;
 *    char    outString[32];
```

```

*     long   outInt;
*     float  outFloat;
*};
*/
/*
*  service reads in a input view buffer and outputs a view buffer.
*/
void
SIMPVIEW( TPSVCINFO *rqst )
{
    /*
    *  get the structure (VIEWSSVC) from the TPSVCINFO structure
    */
    struct SimpView*svcinfn = (struct SimpView *) rqst->data;
    /*
    *  print the input params to the UserLog. Note there is
    *  no error checking here. Normally a SERVER would perform
    *  some validation of input and return TPFALL if the input
    *  is not correct.
    */
    (void)userlog("SIMPVIEW: InString=%s, InInt=%d, InFloat=%f",
        svcinfn->inString, svcinfn->inInt, svcinfn->inFloat);
    /*
    *  Populate the output fields and send them back to the caller
    */

    strcpy (svcinfn->outString, "Return from SIMPVIEW");
    svcinfn->outInt = 100;
    svcinfn->outFloat = (float) 100.00;
    /*
    *  If there was an error, return TPFALL
    *  tpreturn(TPFALL, ErrorCode, (char *)svcinfn, sizeof (*svcinfn), 0);
    */
    tpreturn(TPSUCCESS, 0, (char *)svcinfn, sizeof (*svcinfn), 0);
}

```

## Using the XML Buffer Type

The XML buffer type enables Oracle Tuxedo applications to use XML documents for exchanging data within and between applications. Oracle Tuxedo applications can send and receive XML buffers, and route those buffers to the appropriate servers. All logic for dealing with XML documents, including parsing, resides in the application.

A well-formed XML document consists of:

- Text in the form of a sequence of encoded characters, including proper headings, opening and closing tags, etc.

- A description of the logical structure of the document and information about that structure.

To use the XML buffer type, you first define the Tuxedo service that you will be using with the buffer type, and then write the code that uses the buffer type. The next two sections demonstrate these steps.

**Note:** Similar to CARRAY, the XML buffer type is treated as a byte array, not a STRING. Therefore, no data conversion takes place between a Jolt client and an Oracle Tuxedo service.

## Define the Tuxedo Service in the Repository Editor

Before running the XML example, you must write and boot a Tuxedo XML service. The XML service takes a buffer and passes it back to the Jolt client. You need to define the XML service in the Metadata Repository Editor (see [Define TOUPPER in the Repository Editor](#) for an example).

## simpxml.java Client Code

The code in the following listing illustrates how Jolt works with a service with an XML type buffer. Because Jolt does not look into the XML data stream, it is the programmer's responsibility to ensure that the data formats between the Jolt client and the XML service match. The example in [Listing 4-10](#) assumes that a session object was already instantiated.

### Listing 4-10 XML Buffer Type Example

---

```

/* Copyright 2001 Oracle Systems, Inc. All Rights Reserved */
/*
 * This code fragment illustrates how Jolt works with a service whose buffer
 * type is XML.
 */

import java.io.*;
import java.lang.*;
import bea.jolt.*;

public class xmldoc {

    public static void main (String[] args) {
        JoltSessionAttributes  sattr;
        JoltSession            session;
        JoltRemoteService      echo_xml;

        String inString = "<?xml version=\"1.0\"
encoding=\"UTF-8\"?><ORDER><HEADER DATE=\"05/13/1999\"
ORDERNO=\"22345\" /><COMPANY>ACME</COMPANY><LINE><ITEM MODEL=\"Pabc\"

```

```

QUANTITY=\ "5\ ">LAPTOP</ITEM></LINE><LINE><ITEM MODEL=\ "P500\ "
QUANTITY=\ "15\ ">LAPTOP</ITEM></LINE></ORDER>" ;

```

```

    byte data[];
    DataInputStream din;
    DataOutputStream dout;
    ByteArrayInputStream bin;
    ByteArrayOutputStream bout;

    byte odata[];
    String outString = null;
    String appAddress = null;

    //...Create Jolt Session

    try {
        /*
         * Use java.io.DataOutputStream to put data
         * into a byte array
         */
        bout = new ByteArrayOutputStream(inString.length());
        dout = new DataOutputStream(bout);
        dout.writeBytes(inString);

        /*
         * Copy the byte array into a new byte array "data".
         * Then issue the Jolt remote service call.
        */
        data = bout.toByteArray();
    } catch (Exception e) {
        System.out.println("toByteArray error");
        return;
    }

    try {
        echo_xml = new JoltRemoteService("ECHO_XML", session);
        System.out.println("JoltRemoteService Created");
        echo_xml.setBytes("XML", data, data.length);
    } catch (Exception e) {
        System.out.println("RemoteService call error" + e);
        return;
    }

    echo_xml.call(null);
    System.out.println("Service Call Returned");
    odata = echo_xml.getBytesDef("XML", null);

    try {
        System.out.println("Return String is:" + new
String(odata));

```

```

        } catch (Exception e) {
            System.err.println("getByteDef Error");
        }
    }
}
// end of class

```

---

## Using the MBSTRING Buffer Type

Starting with Tuxedo 9.0, Jolt supports the MBSTRING buffer type which is already supported by Tuxedo ATMI as of Tuxedo 8.1.

Since Java uses Unicode as the standard for multi byte character encoding and provides String class for handling Unicode string data, Jolt MBSTRING support will use the String class as the MBSTRING container on the Java client side. Jolt automatically converts the Unicode MBSTRING data in a String object between byte array MBSTRING data, which is the ATMI's MBSTRING representation, when the data is transferred between a Jolt client and a Tuxedo server.

The following methods are added to `bea.jolt.Message` interface and to `bea.jolt.JoltMessage` and `bea.jolt.JoltRemoteService` classes.

```

addMBString
setMBString
setMBStringItem
getMBStringDef
getMBStringItemDef

```

The usage of the MBSTRING buffer type is very similar to the STRING buffer type except that the buffer type specified in the Metadata Repository Editor is “MBSTRING” and the Java methods used for setting and getting the MBSTRING data are listed above.

In addition, the following Java system properties are used to specify the character encoding name for MBSTRING data sent to Tuxedo servers.

```
bea.jolt.mbencoding
```

The Tuxedo encoding name used for converting Unicode MBSTRING data to the corresponding byte array MBSTRING data while sending MBSTRING data to a Tuxedo server. If this property is not specified, the Java default character encoding name is used and mapped to the corresponding Tuxedo encoding name. For example, the default Japanese Windows encoding name “MS932” should be mapped to the corresponding Tuxedo encoding name “CP932” and specified in this property.

`bea.jolt.mbencodingmap`

The full path name for the file which specifies character encoding name mapping between Jolt clients and Tuxedo servers. This mapping is necessary because the character encoding name for the same character encoding is sometimes different between Java and Tuxedo. For example, the default Japanese Windows encoding name is MS932 in Java, but in Tuxedo it is CP932. If this property is not specified, mapping is not done.

This means that the Java character encoding name is directly set in the MBSTRING data sent to the Tuxedo server, and the encoding name which is set in the received MBSTRING data from the Tuxedo server is used as the Java encoding name. This may cause a conversion error if the encoding name is not supported by Java or Tuxedo.

To specify the `bea.jolt.mbencoding` or `bea.jolt.mbencodingmap`, `jolti18n.jar` must be included in the `CLASSPATH`. If `jolti18n.jar` is not included in the `CLASSPATH`, the encoding name is set to “ISO-8859-1” and no encoding name is done between Java and Tuxedo even if these properties are specified in the Java command line.

## Multithreaded Applications

As a Java-based set of classes, Jolt supports multithreaded applications; however, various implementations of the Java language differ with respect to certain language and environment features. Jolt programmers need to be aware of the following:

- The use of preemptive and non-preemptive threads when creating applications or applets with the Jolt Class Library.
- The use of threads to get asynchronous behavior similar to the `tpacall()` function in Oracle Tuxedo.

“**Threads of Control**” describes the issues arising from using threads with different Java implementations and is followed by an example of the use of threads in a Jolt program.

**Note:** Most Java implementations provide preemptive rather than non-preemptive threads. The difference between these two models can lead to very different performance and programming requirements.

## Threads of Control

Each concurrently operating task in the Java virtual machine is a thread. Threads exist in various states, the important ones being **RUNNING**, **RUNNABLE**, or **BLOCKED**.

- A **RUNNING** thread is a currently executing thread.
- A **RUNNABLE** thread can be run once the current thread has relinquished control of the CPU. There can be many threads in the **RUNNABLE** state, but only one can be in the **RUNNING** state. Running a thread means changing the state of a thread from **RUNNABLE** to **RUNNING**, and causing the thread to have control of the Java Virtual Machine (VM).
- A **BLOCKED** thread is a thread that is waiting on the availability of some event or resource.

**Note:** The Java VM schedules threads of the same priority to run in a round-robin mode.

## Preemptive Threading

The main performance difference between the two threading models arises in telling a running thread to relinquish control of the Java VM. In a preemptive threading environment, the usual procedure is to set a hardware timer that goes off periodically. When the timer goes off, the

current thread is moved from the `RUNNING` to the `RUNNABLE` state, and another thread is chosen to run.

## Non-Preemptive Threading

In a non-preemptive threading environment, a thread must volunteer to give up control of the CPU and move to the `RUNNABLE` state. Many methods in the Java language classes contain code that volunteers to give up control, and are typically associated with actions that might take a long time. For example, reading from the network generally causes a thread to wait for a packet to arrive. A thread that is waiting on the availability of some event or resource is in the `BLOCKED` state. When the event occurs or the resource becomes available, the thread becomes `RUNNABLE`.

## Using Jolt with Non-Preemptive Threading

If your Jolt-based Java program is running on a non-preemptive threading Virtual Machine (such as Sun Solaris), the program must either:

- Occasionally call a method that blocks the thread, or
- Explicitly give up control of the CPU using the `Thread.yield()` method

The typical usage is to make the following call in all long-running code segments or potentially time-consuming loops:

```
Thread.currentThread.yield();
```

Without sending this message, the threads used by the Jolt Library may never get scheduled and, as such, the Jolt operation is impaired.

The only virtual machine known to use non-preemptive threading is the Java Developer's Kit (JDK) machine running on a Sun platform. If you want your applet to work on JDK 1.3, you must make sure to send the yield messages. As mentioned earlier, some methods contain yields. An important exception is the `System.in.read` method. This method does not cause a thread switch. Rather than rely on these messages, we suggest using yields explicitly.

## Using Threads for Asynchronous Behavior

You can use threads in Jolt to get asynchronous behavior that is analogous to the `tpacall()` function in Oracle Tuxedo. With this capability, you do not need an asynchronous service request function. You can get this functionality because Jolt is thread-safe. For example, the Jolt client application can start one thread that sends a request to an Oracle Tuxedo service function and then

immediately start another thread that sends another request to an Oracle Tuxedo service function. So even though the Jolt `tpacall()` is synchronous, the application is asynchronous because the two threads are running at the same time.

## Using Threads with Jolt

A Jolt client-side program or applet is fully thread-safe. Jolt support of multithreaded applications includes the following client characteristics:

- Multiple sessions per client
- Multithreaded within a session
- Client application manages threads, not asynchronous calls
- Performs synchronous calls

[Listing 4-11](#) illustrates the use of two threads in a Jolt application.

### Listing 4-11 Using Multiple Threads with Jolt (ThreadBank.java)

---

```

/* Copyright 1996 Oracle Systems, Inc. All Rights Reserved */
import bea.jolt.*;
public class ThreadBank
{
    public static void main (String [] args)
    {
        JoltSession session;
        try
        {
            JoltSessionAttributes dattr;
            String userName = null;
            String userPasswd = null;
            String appPasswd = null;
            String userRole = null;

            // fill in attributes required
            dattr = new JoltSessionAttributes();
            dattr.setString(dattr.APPADDRESS, "//bluefish:8501");

```

```

// instantiate domain
// check authentication level
switch (datr.checkAuthenticationLevel())
{
case JoltSessionAttributes.NOAUTH:
    System.out.println("NOAUTH\n");
    break;
case JoltSessionAttributes.APPASSWORD:
    appPasswd = "myAppPasswd";
    break;
case JoltSessionAttributes.USRPASSWORD:
    userName = "myName";
    userPasswd = "mySecret";
    appPasswd = "myAppPasswd";
    break;
}

datr.setInt(dattr.IDLETIMEOUT, 60);
session = new JoltSession (datr, userName, userRole,
                           userPasswd, appPasswd);

T1 t1 = new T1 (session);
T2 t2 = new T2 (session);

t1.start();
t2.start();

Thread.currentThread().yield();
try
{
    while (t1.isAlive() && t2.isAlive())
    {
        Thread.currentThread().sleep(1000);
    }
}
catch (InterruptedException e)
{
    System.err.println(e);
    if (t2.isAlive())
    {

```

```

        System.out.println("job 2 is still alive");
        try
        {
            Thread.currentThread().sleep(1000);
        }
        catch (InterruptedException e1)
        {
            System.err.println(e1);
        }
    }
    else if (t1.isAlive())
    {
        System.out.println("job1 is still alive");
        try
        {
            Thread.currentThread().sleep(1000);
        }
        catch (InterruptedException e1)
        {
            System.err.println(e1);
        }
    }
}
session.endSession();
}
catch (SessionException e)
{
    System.err.println(e);
}
finally
{
    System.out.println("normal ThreadBank term");
}
}
}

class T1 extends Thread
{

```

```

JoltSession j_session;
JoltRemoteService j_withdrawal;

public T1 (JoltSession session)
{
    j_session=session;
    j_withdrawal= new JoltRemoteService("WITHDRAWAL",j_session);
}

public void run()
{
    j_withdrawal.addInt("ACCOUNT_ID",10001);
    j_withdrawal.addString("SAMOUNT","100.00");
    try
    {
        System.out.println("Initiating Withdrawal from account 10001");
        j_withdrawal.call(null);
        String W = j_withdrawal.getStringDef("SBALANCE","-1.0");
        System.out.println("-->Withdrawal Balance: " + W);
    }
    catch (ApplicationException e)
    {
        e.printStackTrace();
        System.err.println(e);
    }
}
}

class T2 extends Thread
{
    JoltSession j_session;
    JoltRemoteService j_deposit;

    public T2 (JoltSession session)
    {
        j_session=session;
        j_deposit= new JoltRemoteService("DEPOSIT",j_session);
    }

    public void run()
    {
        j_deposit.addInt("ACCOUNT_ID",10000);

```

```

j_deposit.addString("SAMOUNT", "100.00");
try
{
    System.out.println("Initiating Deposit from account 10000");
    j_deposit.call(null);
    String D = j_deposit.getStringDef("SBALANCE", "-1.0");
    System.out.println("-->Deposit Balance: " + D);
}
catch (ApplicationException e)
{
    e.printStackTrace();
    System.err.println(e);
}
}
}

```

---

## Event Subscription and Notifications

Programmers developing client applications with Jolt can receive event notifications from either Oracle Tuxedo Services or other Oracle Tuxedo clients. The Jolt Class Library contains classes that support the following types of Oracle Tuxedo notifications for handling event-based communication:

- **Unsolicited Event Notifications**—these are notifications that a Jolt client receives as a result of an Oracle Tuxedo client or service issuing a broadcast using either a `tpbroadcast()` or a directly targeted message via a `tpnotify()` ATMI call.
- **Brokered Event Notifications**—these notifications are received by a Jolt client through the Oracle Tuxedo Event Broker. The notifications are only received when the Jolt client subscribes to an event *and* any Oracle Tuxedo client or server issues a system-posted event or `tppost()` call.

## Event Subscription Classes

The Jolt Class Library provides four classes that implement the asynchronous notification mechanism for Jolt client applications:

- **JoltSession**—the `JoltSession` class includes an `onReply()` method for receiving notifications and notification messages.
- **JoltReply**—the `JoltReply` class gives the client application access to any messages received with an event or notification.
- **JoltMessage**—the `JoltMessage` class provides `get()` methods for obtaining information about the notification or event.
- **JoltUserEvent**—the `JoltUserEvent` class supports subscription to both unsolicited and event notification types.

For additional information about these classes refer to the Oracle Jolt API Reference.

## Notification Event Handler

For both unsolicited notifications and a brokered event notification, the Jolt client application requires an event handler routine that is invoked upon receipt of a notification. Jolt only supports a single handler per session. In Oracle Tuxedo versions, you cannot determine which event generated a notification. Therefore, you cannot invoke an event-specific handler based on a particular event.

The client application must provide a single handler (by overriding the `onReply()` method) per session that will be invoked for all notifications received by that client for that session. The single handler call-back function is used for both unsolicited and event notification types. It is up to the (user-supplied) handler routine to determine what event caused the handler invocation and to take appropriate action. If the user does not override the session handler, then notification messages are silently discarded by the default handler.

The Jolt client provides the call back function by subclassing the `JoltSession` class and overriding the `onReply()` method with a user-defined `onReply()` method.

In Oracle Tuxedo/ATMI clients, processing in the handler call-back function is limited to a subset of ATMI calls. This restriction does not apply to Jolt clients. Separate threads are used to monitor notifications and run the event handler method. A Jolt client can perform all Jolt-supported functionality from within the handler. All the rules that apply to a normal Jolt client program apply to the handler, such as a single transaction per session at any time.

Each invocation of the handler method takes place in a separate thread. The application developer should ensure that the `onReply()` method is either synchronized or written thread-safe, because separate threads could be executing the method simultaneously.

Jolt uses an implicit model for enabling the handler routine. When a client subscribes to an event, Jolt internally enables the handler for that client, thus enabling unsolicited notifications as well. A Jolt client cannot subscribe to event notifications without also receiving unsolicited notifications. In addition, a single `onReply()` method is invoked for both types of notifications.

## Connection Modes

Jolt supports notification receipts for clients working in either connection-retained or connection-less modes of operation. Connection-retained clients receive all notifications. Jolt clients working in connection-less mode receive notifications while they have an active network connection to the Jolt Session Handler (JSH). When the network connection is closed, the JSH logs and drops notifications destined for the client. Jolt clients operating in a connection-less mode do not receive unsolicited messages or notifications while they do not have an active network connection. All messages received during this time are logged and discarded by the JSH.

Connection mode notification handling includes acknowledged notifications for Jolt clients in the Oracle Tuxedo environment. If a JSH receives an acknowledged notification for a client and the client does not have an active network connection, the JSH logs an error and returns a failure acknowledgment to the notification.

## Notification Data Buffers

When a client receives notification, it is accompanied by a data buffer. The data buffer can be of any Oracle Tuxedo data buffer type. Jolt clients (for example, the handler) receive these buffers as a `JoltMessage` object and should use the appropriate `JoltMessage` class `get*()` methods to retrieve the data from this object.

The Oracle Tuxedo Service Metadata Repository does not need to have the definition of the buffers used for notification. However, the Jolt client application programmer needs to know field names.

The Jolt system does not provide functionality equivalent to `tptypes()` in Oracle Tuxedo. For FML and VIEW buffers, the data is accessed using the `get*()` methods with the appropriate field name, for example:

```
getIntDef ("ACCOUNT_ID", -1);
```

For STRING and CARRAY buffers, the data is accessed by the same name as the buffer type:

```
getStringDef ("STRING", null);
getBytesDef ("CARRAY", null);
```

STRING and CARRAY buffers contain only a single data element. This complete element is returned by the preceding `get*()` methods.

## Oracle Tuxedo Event Subscription

Oracle Tuxedo brokered event notification allows Oracle Tuxedo programs to post events without knowing what other programs are supposed to receive notification of an event's occurrence. The Jolt event notification allows Jolt client applications to subscribe to Oracle Tuxedo events that are broadcast or posted using the Oracle Tuxedo `tpnotify()` or `tpbroadcast()` calls.

Jolt clients can only subscribe to events and notifications that are generated by other components in Oracle Tuxedo (such as an Oracle Tuxedo service or client). Jolt clients can not send events or notifications.

### Supported Subscription Types

Jolt only supports notification types of subscriptions. The Jolt `onReply()` method is called when a subscription is fulfilled. The Jolt API does not support dispatching a service routine or enqueueing a message to an application queue when a notification is received.

### Subscribing to Notifications

If a Jolt client subscribes to a single event notification, the client receives both unsolicited messages and event notification. Subscribing to an event implicitly enables unsolicited notification. This means that if the application creates a `JoltUserEvent` object for Event "X", the client automatically receives notifications directed to it as a result of `tpnotify()` or `tpbroadcast()`.

**Note:** Subscribing to single event notification is *not* the recommended method for enabling unsolicited notification. If you want unsolicited notification, the application should explicitly subscribe to unsolicited notifications (as described in the `JoltUserEvent` class). The next section is about *unsubscribing* from notifications.

### Unsubscribing from Notifications

To stop subscribing to event notifications and/or unsolicited messages, you need to use the `JoltUserEvent` `unsubscribe` method. In Jolt, disabling unsolicited notifications with an `unsubscribe` method does not turn off all subscription notifications. This differs from Oracle Tuxedo. In Oracle Tuxedo the use of `tpsetunsol()` with a NULL handler turns off all subscription notifications.

When unsubscribing, the following considerations apply:

- If a client is subscribed to a single event, unsubscribing from notification disables both event notification and unsolicited messages.
- If a client has multiple subscriptions, then unsubscribing from any single subscription disables only that single subscription. Unsolicited notifications continue. Only the last subscription to be unsubscribed causes unsolicited notification to stop.
- If a client subscribes to both unsolicited and event notifications, then unsubscribing to only the unsolicited notification will not stop either type of notification from continuing. In addition, this unsubscribe does not throw an exception. However, the Jolt API notes that an unsubscribe has taken place, and a subsequent unsubscribe to the remaining event disables both event notification and unsolicited messages.

If you want to stop unsolicited messages in your client application, you need to make sure that you have unsubscribed to all events.

## Using the Jolt API to Receive Oracle Tuxedo Notifications

The “[Asynchronous Notification](#)” listing shows how to use the Jolt Class Library for receiving notifications and includes the use of the `JoltSession`, `JoltReply`, `JoltMessage` and `JoltUserEvent` classes.

### Listing 4-12 Asynchronous Notification

---

```
class EventSession extends JoltSession
{
    public EventSession( JoltSessionAttributes attr, String user,
                        String role, String upass, String apass )
    {
        super(attr, user, role, upass, apass);
    }
    /**
     * Override the default unsolicited message handler.
     * @param reply a place holder for the unsolicited message
     * @see bea.jolt.JoltReply
     */
    public void onReply( JoltReply reply )
    {
```

```

// Print out the STRING buffer type message which contains
// only one field; the field name must be "STRING". If the
// message uses CARRAY buffer type, the field name must be
// "CARRAY". Otherwise, the field names must conform to the
// elements in FML or VIEW.

JoltMessage msg = (JoltMessage) reply.getMessage();
System.out.println(msg.getStringDef("STRING", "No Msg"));
}
public static void main( Strings args[] )
{
    JoltUserEvent  unsolEvent;
    JoltUserEvent  helloEvent;
    EventSession   session;
    ...

    // Instantiate my session object which can print out the
    // unsolicited messages. Then subscribe to HELLO event
    // and Unsolicited Notification which both use STRING
    // buffer type for the unsolicited messages.

    session = new EventSession(...);

    helloEvent = new JoltUserEvent("HELLO", null, session);
    unsolEvent = new JoltUserEvent(JoltUserEvent.UNSOLMSG, null,
                                   session);
    ...
    // Unsubscribe the HELLO event and unsolicited notification.
    helloEvent.unsubscribe();
    unsolEvent.unsubscribe();
}
}

```

---

## Clearing Parameter Values

The Jolt Class Library contains the `clear()` method, which allows you to remove existing attributes from an object and, in effect, provides for the reuse of the object. The “[Jolt Object Reuse \(reuseSample.java\)](#)” listing illustrates how to use the `clear()` method to clear parameter

values and how to reuse the `JoltRemoteService` parameter values; you do not have to destroy the service to reuse it. Instead, the `svc.clear()` statement is used to discard the existing input parameters before reusing the `addString()` method.

#### Listing 4-13 Jolt Object Reuse (reuseSample.java)

---

```

/* Copyright 1999 Oracle Systems, Inc. All Rights Reserved */
import java.net.*;
import java.io.*;
import bea.jolt.*;
/*
 * This is a Jolt sample program that illustrates how to reuse the
 * JoltRemoteService after each invocation.
 */
class reuseSample
{
    private static JoltSession s_session;
    static void init( String host, short port )
    {
        /* Prepare to connect to the Tuxedo domain. */
        JoltSessionAttributes attr = new JoltSessionAttributes();
        attr.setString(attr.APPADDRESS,"/"+ host+": " + port);

        String username = null;
        String userrole = "sw-developer";
        String applpasswd = null;
        String userpasswd = null;

        /* Check what authentication level has been set. */
        switch (attr.checkAuthenticationLevel())
        {
            case JoltSessionAttributes.NOAUTH:
                break;
            case JoltSessionAttributes.APPASSWORD:
                applpasswd = "secret8";
                break;
            case JoltSessionAttributes.USRPASSWORD:
                username = "myName";

```

```

        userpasswd = "BEA#1";
        applpasswd = "secret8";
        break;
    }

    /* Logon now without any idle timeout (0). */
    /* The network connection is retained until logoff. */
    attr.setInt(attr.IDLETIMEOUT, 0);
    s_session = new JoltSession(attr, username, userrole,
        userpasswd, applpasswd);
}

public static void main( String args[] )
{
    String host;
    short port;
    JoltRemoteService svc;

    if (args.length != 2)
    {
        System.err.println("Usage: reuseSample host port");
        System.exit(1);
    }

    /* Get the host name and port number for initialization. */
    host = args[0];
    port = (short)Integer.parseInt(args[1]);

    init(host, port);

    /* Get the object reference to the DELREC service. This
     * service has no output parameters, but has only one input
     * parameter.
     */
    svc = new JoltRemoteService("DELREC", s_session);
    try
    {
        /* Set input parameter REPNAME. */
        svc.addString("REPNAME", "Record1");
        svc.call(null);
        /* Change the input parameter before reusing it */
    }
}

```

```

        svc.setString("REPNAME", "Record2");
        svc.call(null);

        /* Simply discard all input parameters */
        svc.clear();
        svc.addString("REPNAME", "Record3");
        svc.call(null);
    }
    catch (ApplicationException e)
    {
        System.err.println("Service DELREC failed: "+
            e.getMessage()+" "+ svc.getStringDef("MESSAGE", null));
    }

    /* Logoff now and get rid of the object. */
    s_session.endSession();
}
}

```

---

## Reusing Objects

The following listing, “[Extending Jolt Remote Service \(extendSample.java\)](#),” illustrates one way to subclass the JoltRemoteService class. In this case, a TransferService class is created by subclassing the JoltRemoteService class. The TransferService class extends the JoltRemoteService class, adding a Transfer feature that makes use of the Oracle Tuxedo BANKAPP funds TRANSFER service.

[Listing 4-14](#) uses the `extends` keyword from the Java language. The `extends` keyword is used in Java to subclass a base (parent) class. The following code shows one of many ways to extend from JoltRemoteService.

### Listing 4-14 Extending Jolt Remote Service (extendSample.java)

---

```

/* Copyright 1999 Oracle Systems, Inc. All Rights Reserved */

import java.net.*;
import java.io.*;
import bea.jolt.*;

```

```

/*
 * This Jolt sample code fragment illustrates how to customize
 * JoltRemoteService. It uses the Java language "extends" mechanism
 */
class TransferService extends JoltRemoteService
{
    public String    fromBal;
    public String    toBal;

    public TransferService( JoltSession session )
    {
        super("TRANSFER", session);
    }

    public String doxfer( int fromAcctNum, int toAcctNum, String amount )
    {
        /* Clear any previous input parameters */
        this.clear();

        /* Set the input parameters */
        this.setIntItem("ACCOUNT_ID", 0, fromAcctNum);
        this.setIntItem("ACCOUNT_ID", 1, toAcctNum);
        this.setString("SAMOUNT", amount );

        try
        {
            /* Invoke the transfer service. */
            this.call(null);

            /* Get the output parameters */
            fromBal = this.getStringItemDef("SBALANCE", 0, null);
            if (fromBal == null)
                return "No balance from Account " +
                    fromAcctNum;
            toBal = this.getStringItemDef("SBALANCE", 1, null);
            if (toBal == null)
                return "No balance from Account " + toAcctNum;
            return null;
        }
        catch (ApplicationException e)
        {

```

```

        /* The transaction failed, return the reason */
        return this.getStringDef("STATLIN", "Unknown reason");
    }
}
}
class extendSample
{
    public static void main( String args[] )
    {
        JoltSession s_session;
        String host;
        short port;
        TransferService xfer;
        String failure;

        if (args.length != 2)
        {
            System.err.println("Usage: reuseSample host port");
            System.exit(1);
        }

        /* Get the host name and port number for initialization. */
        host = args[0];
        port = (short)Integer.parseInt(args[1]);

        /* Prepare to connect to the Tuxedo domain. */
        JoltSessionAttributes attr = new JoltSessionAttributes();
        attr.setString(attr.APPADDRESS,"/"+ host+": " + port);

        String username = null;
        String userrole = "sw-developer";
        String applpasswd = null;
        String userpasswd = null;

        /* Check what authentication level has been set. */
        switch (attr.checkAuthenticationLevel())
        {
            case JoltSessionAttributes.NOAUTH:
                break;
            case JoltSessionAttributes.APPASSWORD:

```

```

        applpasswd = "secret8";
        break;
    case JoltSessionAttributes.USRPASSWORD:
        username = "myName";
        userpasswd = "BEA#1";
        applpasswd = "secret8";
        break;
}

/* Logon now without any idle timeout (0). */
/* The network connection is retained until logoff. */
attr.setInt(attr.IDLETIMEOUT, 0);
s_session = new JoltSession(attr, username, userrole,
userpasswd, applpasswd);

/*
 * TransferService extends from JoltRemoteService and uses the
 * standard Oracle Tuxedo BankApp TRANSFER service. We invoke this
 * service twice with different parameters. Note, we assume
 * that "s_session" is initialized somewhere before.
 */

xfer = new TransferService(s_session);
if ((failure = xfer.doxfer(10000, 10001, "500.00")) != null)
    System.err.println("Tranasaction failed: " + failure);
else
{
    System.out.println("Transaction is done.");
    System.out.println("From Acct Balance: "+xfer.fromBal);
    System.out.println(" To Acct Balance: "+xfer.toBal);
}

if ((failure = xfer.doxfer(51334, 40343, "$123.25")) != null)
    System.err.println("Tranasaction failed: " + failure);
else
{
    System.out.println("Transaction is done.");
    System.out.println("From Acct Balance: "+xfer.fromBal);
    System.out.println(" To Acct Balance: "+xfer.toBal);
}
}

```

```
}  
}
```

---

# Deploying and Localizing Jolt Applets

Using the Jolt Class Library, you can build Java applications that execute from within a client Web browser. For these types of applications, perform the following application development tasks:

- Deploy your Jolt applet in an HTML page.
- Localize your Jolt applets for different languages and character sets.

The following sections describe these application development considerations.

## Deploying a Jolt Applet

When you deploy a Jolt applet, consider the following:

- Installation and configuration requirements for the Oracle Tuxedo server and Jolt Server
- Client-side execution of the applet
- Requirements for the Web server that downloads the Java applet

Information for configuring the Oracle Tuxedo server and Jolt server to work with Jolt is available in *Installing the Oracle Tuxedo System*. The following sections describe common client and Web server considerations for deploying Jolt applets.

## Client Considerations

When you write a Java applet that incorporates Jolt classes, the applet works just as any other Java applet in an HTML page. A Jolt applet can be embedded in an HTML page using the HTML applet tag:

```
<applet code="applet_name.class"> </applet>
```

If the Jolt applet is embedded in an HTML page, the applet is downloaded when the HTML page loads. You can code the applet to run immediately after it is downloaded, or you can include code that sets the applet to run based upon a user action, a timeout, or a set interval. You can also create an applet that downloads in the HTML page, but opens in another window or, for instance, simply plays a series of sounds or musical tunes at intervals. The programmer has a large degree of freedom in coding the applet initialization procedure.

**Note:** If the user loads a new HTML page into the browser, the applet execution is stopped.

## Web Server Considerations

When you use the Jolt classes in a Java applet, the Jolt Server must run on the same machine as the Web server that downloads the Java applet unless you install Jolt Relay on the Web server.

When a webmaster sets up a Web server, a directory is specified to store all the HTML files. Within that directory, a subdirectory named “classes” must be created to contain all Java class files and packages. For example:

```
<html-dir>/classes/bea/jolt
```

Or, you can set the CLASSPATH to include the `jolt.jar` file that contains all the Jolt classes.

**Note:** You can place the Jolt classes subdirectory anywhere. For convenient access, you may want to place it in the same directory as the HTML files. The only requirement for the Jolt classes subdirectory is that the classes must be made available to the Web server.

The HTML file for the Jolt applet should refer the codebase to the `jolt.jar` file or the `classes` directory. For example:

```
/export/html/
  |___ classes/
  |   |___ bea/
  |   |   |___ jolt/
  |   |       |___ JoltSessionAttributes.class
  |   |       |___ JoltRemoteServices.class
  |   |       |___ ...
  |   |___ mycompany/
  |       |___ app.class
  |___ ex1.html
  |___ ex2.html
```

The webmaster may specify the “app” applet in `ex1.html` as:

```
<applet codebase="classes" code=mycompany.app.class width=400 height=200>
```

## Localizing a Jolt Applet

If your Jolt application is intended for international use, you must address certain localization issues. Localization considerations apply to applications that execute from a client Web browser and applications that are designed to run outside a Web browser environment. Localization tasks can be divided into two categories:

- Adapting an application from its original language to a target language.

- Translating strings from one language to another. This sometimes requires specifying a different alphabet or a character set from the one used in the original language.

For localization, the Jolt Class Library package relies on the conventions of the Java language and the Oracle Tuxedo system. Jolt transfers Java 16-bit Unicode characters to the JSH. The JSH provides a mechanism to convert Unicode to the local character set.

For information about the Java implementation for Unicode and character escapes, refer to your Java Development Kit (JDK) documentation.

## Using SSL

Jolt can use SSL as the preferred secure transport mechanism instead of default Link Level Encryption. To enable Jolt to use SSL, the JSL must be configured with '-s secure\_port' in the TUXEDO UBBCONFIG file.

Jolt client library automatically chooses SSL if the JSL connection port is the SSL port. The SSL requires Jolt client to provide information about the location of the X.509 certificate, the private key, and passphrase that is used to encrypt the passphrase.

There are five attributes added to the `JoltSessionAttributes` class to handle these requirement:

- `KEYSTORE`—file path for client private key and X.509 certificate
- `KSPASSPHRASE`—key store passphrase
- `TRUSTSTORE`—trust store file path for trusted X.509 certificates
- `TSPASSPHRASE`—trust store passphrase
- `KEYPASSPHRASE`—private key passphrase

Jolt client library uses the third-party Java Secure Socket Extension (JSSE) implementation for SSL communication. The following JSSE implementations have been tested:

- Sun JSSE implementation bundled in Sun JRE 8.0
- Sun JSSE implementation bundled in HP JRE 8.0
- IBM JSSE implementation bundled in IBM JRE 8.0

**Note:** Starting with JDK release 8u31, the `SSLV3` protocol is deactivated and is not available by default. If `SSLV3` is required, the protocol can be reactivated by removing "SSLV3" from the `jdk.tls.disabledAlgorithms` property in the

<JRE\_HOME>/lib/security/java.security file, or by dynamically setting this Security property to "true" before JSSE is initialized.

[Listing 4-15](#) shows a Jolt client code example that makes it possible to use SSL when communicating with JSL/JSH.

#### **Listing 4-15 Using SSL in Jolt Client Code**

---

```
import java.util.*;
import bea.jolt.*;

public class simpcl extends Object {
    private String      userName      = null;
    private String      userRole      = null;
    private String      appPassword   = null;
    private String      userPassword  = null;
    private JoltSessionAttributes attr = null;
    private JoltSession session       = null;
    private JoltRemoteService toupper = null;
    private JoltTransaction trans     = null;

    // JSL is configured with '-s 5555'
    // the communication between jolt client and JSH will use SSL
    private String      address      = new String('//cerebrum:5555');

    public static void main(String args[]) {
        simpcl c = new simpcl();
        c.doTest();
    }
}
```

```

public void doTest() {
    attr = new JoltSessionAttributes();

    // adding these session attribute
    attr.setString(attr.APPADDRESS, address);
    attr.setString(attr.TRUSTSTORE, 'c:\\samples\\samplecacerts');
    attr.setString(attr.KEYSTORE, 'c:\\samples\\client\\testkeys');

    // Only key store and key will be protected by passphrase in this
sample.

    // But optionally the trust store can also be protected by a passphrase
    // although it is not in this sample.
    attr.setString(attr.KSPASSPHRASE, 'passphrase');
    attr.setString(attr.KEYPASSPHRASE, 'passphrase');
    attr.setInt(attr.IDLETIMEOUT, 300);

    userName = 'juser';
    userRole = 'JUSER';
    userPassword = 'abcd';
    appPassword = 'abcd';

    session = new JoltSession(attr, userName, userRole, userPassword,
                               appPassword);

    // access a Tuxedo TOUPPER service
    toupper = new JoltRemoteService('TOUPPER', session);
    toupper.addString('STRING', 'string');
    trans = new JoltTransaction(60, session);

```

```
try {
    toupper.call(trans);
} catch (ApplicationException ae) {
    ae.printStackTrace();
    System.exit(1);
}

String retString = toupper.getStringDef('STRING', null);
trans.commit();
System.out.println(' returned: ' + retString);
session.endSession();
return;
}
}
```

---



# Using JoltBeans

Formerly available as an add on, JoltBeans are included in Oracle Jolt and are as easy to use as JavaBeans. They are JavaBeans components you use in Java development environments to construct Jolt clients.

You can use popular Java-enabled development tools such as Symantec Visual Café to graphically construct client applications. JoltBeans provide a JavaBeans-compliant interface to Oracle Jolt. You can develop a fully functional Oracle Jolt client without writing any code.

This topic includes the following sections:

- [Overview of Jolt Beans](#)
- [Basic Steps for Using JoltBeans](#)
- [JavaBeans Events and Oracle Tuxedo Events](#)
- [How JoltBeans Use JavaBeans Events](#)
- [The JoltBeans Toolkit](#)
- [Jolt-Aware GUI Beans](#)
- [Using the Property List and the Property Editor to Modify the JoltBeans Properties](#)
- [JoltBeans Class Library Walkthrough](#)
- [Using the Oracle Tuxedo Service Metadata Repository and Setting the Property Values](#)
- [JoltBeans Programming Tasks](#)

# Overview of Jolt Beans

JoltBeans consists of two sets of Java Beans. The first set, the JoltBeans Toolkit, is a beans version of the Jolt API. The second set consists of GUI beans, which include Jolt-aware AWT beans and Jolt-aware Swing beans. These GUI components are a “Jolt-enabled” version of some of the standard Java AWT and Swing components, and help you build a Jolt client GUI with minimal or no coding.

You can drag and drop JoltBeans from the component palette of a development tool and position them on the Java form (or forms) of the Jolt client application you are creating. You can populate the properties of the beans and graphically establish event source-listener relationships between various beans of the application or applet. Typically, the development tool is used to generate the event hook-up code, or you can code the hook-up manually. Client development using JoltBeans is integrated with the Oracle Tuxedo Service Metadata Repository, providing easy access to available Oracle Tuxedo services.

**Note:** Currently, Symantec Visual Café 3.0 is the only IDE that is certified by Oracle for use with JoltBeans. However, JoltBeans are also compatible with other Java development environments such as Visual Age.

To use the JoltBeans Toolkit, it is recommended that you be familiar with JavaBeans-enabled, integrated development environments (IDEs). The walkthrough in this chapter is based on Symantec’s Visual Café 3.0 IDE and illustrates the basic steps of building a sample applet.

## JoltBeans Terms

You will encounter the following terms as you work with JoltBeans:

### **JavaBeans**

Portable, platform-independent, reusable software components that are graphically displayed in a development environment.

### **JoltBeans**

Two sets of Java Beans: JoltBeans toolkit and Jolt aware GUI beans.

### **Custom GUI element**

A Java GUI class that communicates with JoltBeans. The means of communication can be JavaBeans events, methods, or properties offered by JoltBeans.

### **Jolt-Aware Bean**

A bean that is the source of JoltInputEvents, listener of JoltOutputEvents, or both. Jolt-aware beans are a subset of Custom GUI elements that follow beans guidelines.

### **Jolt-Aware GUI Beans**

Two packages of GUI components Abstract Window Toolkit (AWT) and Swing, both containing the JoltList, JoltCheckBox, JoltTextField, JoltLabel, and JoltChoice components.

### **JoltBeans Toolkit**

A JavaBeans-compliant interface to Oracle Jolt, which includes the JoltServiceBean, JoltSessionBean, and JoltUserEventBean.

### **Wiring**

The process of connecting beans together so that one bean is registered as a listener of events from another bean.

# Adding JoltBeans to Your Java Development Environment

Before you can use JoltBeans, set up your Java development environment to include JoltBeans:

- Set the `CLASSPATH` in your development environment to include all Jolt classes.
- Add JoltBeans to the Component Library of your development environment.

The method of setting the `CLASSPATH` can vary, depending on the development environment you use.

JoltBeans includes a set of `.jar` files containing all of the JoltBeans. You can add these `.jar` files to your preferred Java development environment so that JoltBeans are available in the component library of your Java tool. For example, using Symantec Visual Café, you can set the `CLASSPATH` so that the `.jar` files are visible in the Component Library window of Visual Café. You only need to set the `CLASSPATH` of these `.jar` files in your development environment once. After you place these `.jar` files in the `CLASSPATH` of your development environment, you can then add JoltBeans to the Component Library. Then you can simply drag and drop any JoltBean directly onto the Java form on which you are developing your Jolt client application.

To set the `CLASSPATH` in your Java development environment, follow the instructions in the product documentation for your development environment. Navigate from the IDE of your development tool to the directory where the `jolt.jar` file resides. The `jolt.jar` file is typically found in the directory called `%TUXDIR%\udatadoj\jolt`. The `jolt.jar` file contains the main Jolt classes. Set the `CLASSPATH` to include these classes. The JoltBean `.jar` files do not need to be added to the `CLASSPATH`. To use them, you only need to add them as components in your IDE.

After you have set the `CLASSPATH` to include the Jolt classes, you can add JoltBeans to the Component Library of your development environment. See the documentation for your particular development environment for instructions on populating the Component Library.

When you are ready to add JoltBeans to the Component Library of your development environment, add only the development version of JoltBeans. Refer to [“Using Development and Run-time JoltBeans”](#) for complete details.

## Using Development and Run-time JoltBeans

The `.jar` files containing JoltBeans contain two versions of each JoltBean, a development version and a run-time version. The development version of each JoltBean name ends with the suffix `Dev`. The run-time version of each class name ends with the suffix `Rt`. For example, the development version of the class, JoltBean, is `JoltBeanDev`, while the run-time version of the same class is `JoltBeanRt`.

Use the development version of JoltBeans during the development process. The development JoltBeans have additional properties that enhance development in a graphic IDE. For example, the JoltBeans have graphic properties (“bean information”) that allow you to work with them as graphic icons in your development environment.

The run-time version of JoltBeans does not have these additional properties. You do not need the additional development properties of the beans at run time. The run-time beans are simply a pared down version of the development JoltBeans.

When you compile your application in your development environment, it is compiled using the development beans. However, if you want to run it from a command line outside of your development environment, it is recommended that you set the `CLASSPATH` so that the run-time beans are used when compiling your application.

## Basic Steps for Using JoltBeans

The basic steps for using JoltBeans are as follows:

1. Add the development version of JoltBeans to the Component Library of your Java development environment, as described in [“Adding JoltBeans to Your Java Development Environment.”](#)
2. Drag the beans from the JoltBeans component palette of your development environment to the Java form-designer for a Jolt client application or applet.
3. Populate the properties of the beans and set up the event-source listener relationships between the beans of the application or applet (“wire” the beans together). The development tool generates the event hook-up code.
4. Add the application logic to the event callbacks.

These steps are explained in more detail in later sections. The JoltBeans walkthrough demonstrates each of these steps with an example.

## JavaBeans Events and Oracle Tuxedo Events

JavaBeans communicate through events. An event in an Oracle Tuxedo system is different from an event in a JavaBeans environment. In an Oracle Tuxedo application, an event is raised from one part of an application to another part of the same application. JoltBeans events are communicated between beans.

## Using Oracle Tuxedo Event Subscription and Notification with JoltBeans

Oracle Tuxedo supports brokered and unsolicited event notification. Jolt provides a mechanism for Jolt clients to receive Oracle Tuxedo events. JoltBeans also include this capability.

**Note:** Oracle Tuxedo event subscription and notification is different from JavaBeans events.

The following procedure illustrates how the Oracle Tuxedo asynchronous notification mechanism is used in JoltBeans applications.

1. Use the `setEventName()` and `setFilter()` methods of the `JoltUserEventBean` to specify the Oracle Tuxedo event to which you want to subscribe.
2. The component that receives the event notifications registers itself as a `JoltOutputListener` to the `JoltSessionBean`.
3. The `subscribe()` method is called on `JoltUserEventBean`.
4. When the actual Oracle Tuxedo event notification arrives, `JoltSessionBean` sends a `JoltOutputEvent` to its listeners by calling `serviceReturned()` on them. The `JoltOutputEvent` object contains the data of the Oracle Tuxedo event.

When the client no longer needs to receive the event, it calls `unsubscribe()` on the `JoltUserEventBean`.

**Note:** If the client will only subscribe to unsolicited events, use `setEventName("\\.UNSOLMSG")`, which can be set using the property sheet. `EventName` and `Filter` are properties of the `JoltUserEventBean`.



# The JoltBeans Toolkit

The JoltBeans Toolkit includes the following beans:

- [JoltSessionBean](#)
- [JoltServiceBean](#)
- [JoltUserEventBean](#)

These components transform the complete Jolt Class Library into beans components, with all of the features of any typical JavaBean, including easy reuse and graphic development.

Refer to the online Oracle Jolt API Reference for specific descriptions of the JoltBeans classes, constructors, and methods.

The following sections provide information about the properties of each bean.

## JoltSessionBean

The `JoltSessionBean`, which represents the Oracle Tuxedo session, encapsulates the functionality of the `JoltSession`, `JoltSessionAttributes`, and `JoltTransaction` classes. The `JoltSessionBean` has properties that you use to set session and security attributes, such as sending a timeout or an Oracle Tuxedo username, as well as methods to open and close an Oracle Tuxedo session.

The `JoltSessionBean` sends a `PropertyChange` event when the Oracle Tuxedo session is established or closed. `PropertyChange` is a standard bean event defined in the `java.beans` package. The purpose of this event is to signal other beans about a change of the value of a property in the source bean. In this case, the source is the `JoltSessionBean`; the targets are `JoltServiceBeans` or `JoltUserEventBeans`; and the property changing is the `LoggedOn` property of the `JoltSessionBean`. When a logon is successful and a session is established, `LoggedOn` is set to `true`. After the logoff is successful and the session is closed, the `LoggedOn` property is set to `false`.

The `JoltSessionBean` provides methods to control transactions, including `beginTransaction()`, `commitTransaction()`, and `rollbackTransaction()`.

[Table 5-1](#) shows the `JoltSessionBean` properties and descriptions.

**Table 5-1 JoltSessionBean Properties and Descriptions**

Property	Description
AppAddress	Set the IP address (host name) and port number of the JSL or the Jolt Relay. The format is <code>//host:port number</code> (for example, <code>myhost:7000</code> ).
AppPassword	Set the Oracle Tuxedo application password used at logon, if required.
IdleTimeOut	Set the IDLETIMEOUT value.
inTransaction	Indicate <code>true</code> or <code>false</code> depending if a transaction has been started and not committed or aborted.
LoggedOn	Indicate <code>true</code> or <code>false</code> if an Oracle Tuxedo session does or does not exist.
ReceiveTimeOut	Set the RECVTIMEOUT value.
SendTimeOut	Set the SENDTIMEOUT value.
SessionTimeOut	Set the SESSIONTIMEOUT value.
UserName	Indicate the Oracle Tuxedo username, if required.
UserPassword	Indicate the Oracle Tuxedo user password, if required.
UserRole	Indicate the Oracle Tuxedo user role, if required.

## JoltServiceBean

The `JoltServiceBean` represents a remote Oracle Tuxedo service. The name of the service is set as a property of the `JoltServiceBean`. The `JoltServiceBean` listens to `JoltInputEvents` from other beans to populate its input buffer. `JoltServiceBean` offers the `callService()` method to invoke the service. `JoltServiceBean` is an event source for `JoltOutputEvents` that carry information about the output of the service. After a successful `callService()`, listener beans are notified via a `JoltOutputEvent` that carries the reply message.

Although the primary way of changing and querying the underlying message buffer of the `JoltServiceBean` is via events, the `JoltServiceBean` also provides methods to access the underlying message buffer directly (`setInputValue(...)`, `getOutputValue(...)`).

[Table 5-2](#) shows the `JoltServiceBean` properties and descriptions.

**Table 5-2 JoltServiceBean Properties and Descriptions**

Property	Description
<code>ServiceName</code>	The name of the Oracle Tuxedo service represented by this <code>JoltServiceBean</code> .
<code>Session</code>	The <code>JoltSessionBean</code> associated with the bean that allows access to the Oracle Tuxedo client session.
<code>Transactional</code>	Set to <code>true</code> if this <code>JoltServiceBean</code> is to be included in the transaction that was started by its <code>JoltSessionBean</code> .

## JoltUserEventBean

The `JoltUserEventBean` provides access to Oracle Tuxedo events. You define the Oracle Tuxedo event to which you subscribe or unsubscribe by setting the appropriate properties of this bean (event name and event filter). The actual event notification is delivered in the form of a `JoltOutputEvent` from the `JoltSessionBean`.

[Table 5-3](#) shows the `JoltUserEventBean` properties and descriptions.

**Table 5-3 JoltUserEventBean Properties and Descriptions**

Property	Description
<code>EventName</code>	Set the name of the user event represented by the bean.
<code>Filter</code>	Set the event filter.
<code>Session</code>	The <code>JoltSessionBean</code> associated with the bean that allows access to the Oracle Tuxedo client session.

## Jolt-Aware GUI Beans

The Jolt-aware GUI Beans consist of Java AWTbeans and Swing beans, and are inherited from the Java Abstract Windowing Toolkit. They include:

- [JoltTextField](#)
- [JoltLabel](#)

- [JoltList](#)
- [JoltCheckbox](#)
- [JoltChoice](#)

**Note:** To avoid errors when compiling, it is recommended that you use only the AWT beans together, or the Swing beans together, rather than mixing beans from these two packages.

## JoltTextField

This is a Jolt-aware extension of `java.awt.TextField` and `Swing JtextField`. `JoltTextField` contains parts of the input for a service. A `JoltServiceBean` can listen to events raised by a `JoltTextField`. `JoltTextField` sends `JoltInputEvents` to its listeners (typically `JoltServiceBeans`) when its contents changes.

`JoltTextField` displays output from a service. In this case, `JoltTextField` listens to `JoltOutputEvents` from `JoltServiceBeans` and updates its contents according to the occurrence of the field to which it is linked.

## JoltLabel

This is a Jolt-aware extension of `java.awt.Label` and `Swing JLabel` that is linked to a specific field in the Jolt output buffer by its `JoltFieldName` property. If the field occurs multiple times, the occurrence to which this textfield is linked is specified by the `occurrenceIndex` property of this bean. `JoltLabel` can be connected with `JoltServiceBeans` to display output from a service. A `JoltLabel` listens to `JoltOutputEvents` from `JoltServiceBeans` and updates its contents according to the occurrence of the field to which it is linked.

## JoltList

This is a Jolt-aware extension of `java.awt.List` and `Swing Jlist` that is linked to a specific Jolt field in the Jolt input or output buffer by its `JoltFieldName` property. If the field occurs multiple times in the Jolt input buffer, the occurrence this list is linked to is specified by the `occurrenceIndex` property of this bean. `JoltList` can be connected with `JoltServiceBeans` in two ways:

- `JoltList` contains parts of the input for a service. A `JoltServiceBean` listens to events raised by a `JoltList`. `JoltList` sends `JoltInputEvents` to its listeners when the selection in the listbox changes. The `JoltInputEvent`, in this case, is populated with the single value of the selected item.

- `JoltList` displays output from a service. When used to display the output of a service, `JoltList` listens to `JoltOutputEvents` from `JoltServiceBeans` and updates its contents accordingly with all occurrences of the field to which it is linked.

## JoltCheckbox

`JoltCheckbox` is a Jolt-aware extension of `java.awt.Checkbox` and Swing `JCheckBox` that is linked to a specific field in the Jolt input buffer by its `JoltFieldName` property. If the field occurs multiple times, the occurrence to which this checkbox is linked is specified by the `occurrenceIndex` property of this bean.

`JoltCheckbox` can be connected with `JoltServiceBeans` to contain parts of the input for a service. A `JoltServiceBean` listens to events raised by a `JoltCheckbox`. `JoltCheckbox` sends `JoltInputEvents` to its listeners (typically `JoltServiceBeans`) when the selection in the checkbox changes. The `JoltInputEvent` in this case is populated with the `TrueValue` property of data type `String` (if the box is selected) or `FalseValue` (if the box is unselected).

## JoltChoice

`JoltChoice` provides a Jolt-aware extension of `java.awt.Choice` and Swing `JChoice` that is linked to a specific field in the Jolt input buffer by its `JoltFieldName` property. If the field occurs multiple times, the occurrence to which this choice is linked is specified by the `occurrenceIndex` property of this bean.

`JoltChoice` can be connected to `JoltServiceBeans` to contain parts of the input for a service. A `JoltServiceBean` can listen to events raised by a `JoltChoice`. `JoltChoice` sends `JoltInputEvents` to its listeners (typically `JoltServiceBeans`) when the selection in the choicebox changes. The `JoltInputEvent` in this case is populated with the single value of the selected item.

**Note:** For a detailed description of these classes, see the Oracle Jolt API Reference.

# Using the Property List and the Property Editor to Modify the JoltBeans Properties

The values of most `JoltBeans` properties can be modified by editing the right column of the Property List in your integrated development environment (IDE), such as Visual Café, as shown in the following figure “[Property List: Ellipsis Button.](#)”

Custom property editors are provided for some properties of `JoltBeans`.

The custom property editors, accessed from the Property List, include dialog boxes that you use to modify the property values. You can invoke the custom property editors from the Property List by clicking the button with the ellipsis (“...”) that is next to the value of the corresponding property value.

**Figure 5-2 Property List: Ellipsis Button**



When you click the ellipsis button, the Property Editor shown in [Figure 5-3](#) is displayed.

**Figure 5-3 Custom Property Editor Dialog Box**



The Custom Property Editor of JoltBeans reads cached information. Initially, no cached information is available, so when the Property Editor is used for the first time, the dialog box is empty. Log on to the Oracle Tuxedo Service Metadata Repository and load the property editor cache from the repository.

For details about the logon and using the Property List and Property Editor, see [“Using the Oracle Tuxedo Service Metadata Repository and Setting the Property Values”](#) on page 5-41.

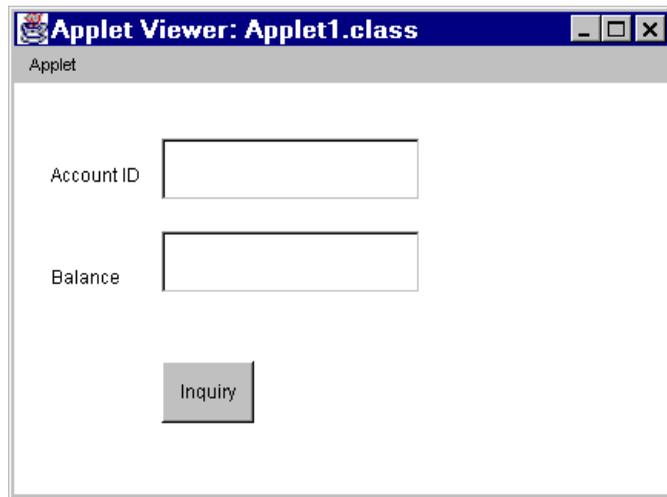
# JoltBeans Class Library Walkthrough

This walkthrough describes how to build an applet that you use to:

- Enter an account ID
- Click on the Inquiry button
- Display the balance of the account (shown in the following figure)

Figure 5-4 shows an example of a completed Java form containing JoltBeans. The applet implements the client functionality for the INQUIRY service of the BANKAPP sample that is included with Oracle Tuxedo. To run this sample, the Oracle Tuxedo server must be running.

**Figure 5-4 Sample Inquiry Applet**



Refer to the figure “[Visual Café 3.0 Form Designer](#)” on page 5-18 for an example of each item required by the Java form. Each item in that figure is described in the following table “[Required Form Elements](#)”.

**Table 5-4 Required Form Elements**

Element	Purpose
Applet (or JApplet, if JFC applet is chosen)	A form used to paint the beans in your development environment.
JoltSessionBean	Logs on to an Oracle Tuxedo session.
JoltTextField	Gets input from the user (in this case, ACCOUNT_ID).
JoltTextField	Displays the result (in this case, SBALANCE).
JoltServiceBean	Accesses an Oracle Tuxedo service. (In this case, INQUIRY from BANKAPP).
Button	Initiates an action.
Label	Describes the field on the applet.

## Building the Sample Form

The sample form is created using an integrated development environment (IDE), in this example, Visual Café 3.0. The example demonstrates how to build an applet that allows you to enter an account ID and use an Oracle Tuxedo service to get and show the account balance.

Follow the basic steps below to create this sample.

1. In Visual Café, choose File→New Project and select either JFC Applet or AWT application. This step provides you with the basic form designer on which you drop the JoltBeans.
2. Drag and drop all of the JoltBeans you want to use in your applet from the Component Library onto the form designer.
3. Modify or customize each bean using the property list or the custom property editor.
4. Wire the beans together using the Interaction Wizard.
5. Compile the applet.

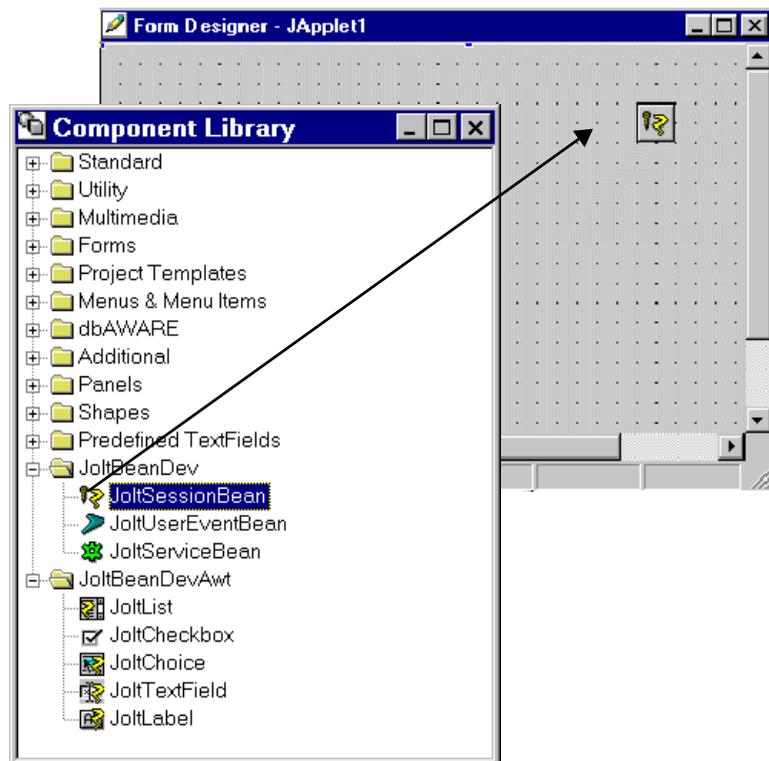
These steps are described in detail in the following sections.

**Note:** The graphic interface of previous versions of Visual Café differ from the look of Visual Café 3.0. You can complete this sample applet in a previous version of Visual Café; however, the steps executed in the Interaction Wizard differ slightly from this example.

## Placing JoltBeans onto the Form Designer

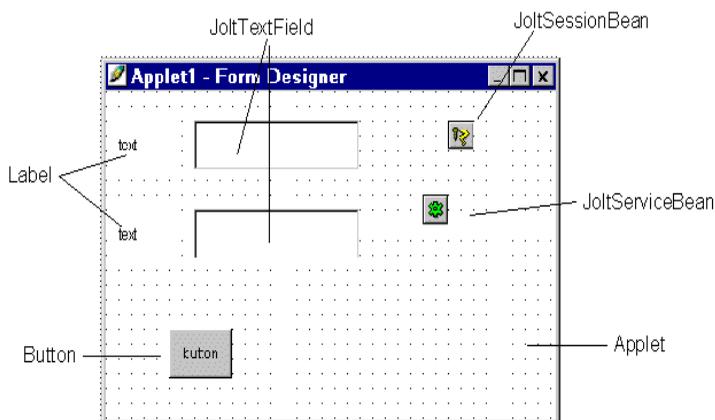
1. Choose File→New Project, and choose JFC Applet.
2. Drag and drop the beans from the Component Library (shown in [Figure 5-5](#)) onto the palette of the form designer.

**Figure 5-5** JoltBeans and the Form Designer in Visual Café



The following figure “[Visual Café 3.0 Form Designer](#)” illustrates how JoltBeans appear when they are placed on the palette of the Form Designer.

**Figure 5-6 Visual Café 3.0 Form Designer**



3. Set the properties of each bean. To modify or customize the buttons, labels or fields, use the property list. Some JoltBeans use a Custom Property Editor.

The following figure, “[Example of JoltTextField Property List and Custom Property Editor](#),” shows how selecting the JoltFieldName of the button property list displays the Custom Property Editor.

4. Set the properties of the beans (for example, set the JoltFieldName property of the JoltTextField to ACCOUNT\_ID).

**Note:** For complete information on setting and modifying the properties of the JoltBeans, refer to “[Using the Oracle Tuxedo Service Metadata Repository and Setting the Property Values](#)” on page 5-41.

[Table 5-5](#) specifies the property values that should be set. Values specified in **bold** and *italic* text are required, and those in plain text are recommended.

**Table 5-5 Required and Recommended Property Values**

Bean	Property	Value
label1	Text	Account ID
label2	Text	Balance
JoltTextField1	Name	accountId

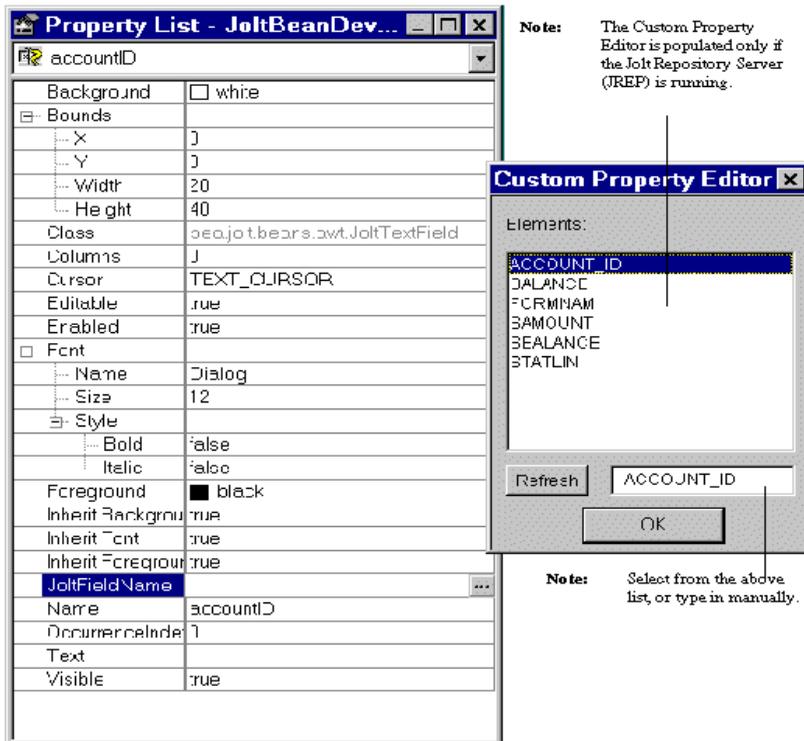
**Table 5-5 Required and Recommended Property Values (Continued)**

<b>Bean</b>	<b>Property</b>	<b>Value</b>
JoltTextField1	JoltFieldName	<b>ACCOUNT_ID</b>
JoltTextField2	Name	balance
JoltTextField2	JoltFieldName	<b>SBALANCE</b>
JoltSessionBean1	AppAddress	<i>//tuxserv:2010</i>
JoltServiceBean1	Name	inquiry
JoltServiceBean1	ServiceName	<b>INQUIRY</b>
button1	Label	Inquiry

**Note:** In this walkthrough, the default occurrenceIndex of 0 works for both JoltTextFields.

Refer to the following figure [“Example of JoltTextField Property List and Custom Property Editor”](#) and [“Using the Oracle Tuxedo Service Metadata Repository and Setting the Property Values”](#) on page 5-41 for general guidelines about JoltBean properties.

Figure 5-7 Example of JoltTextField Property List and Custom Property Editor



- To change the value of the JoltFieldName property, click on the ellipsis button of the JoltFieldName in the Property List.

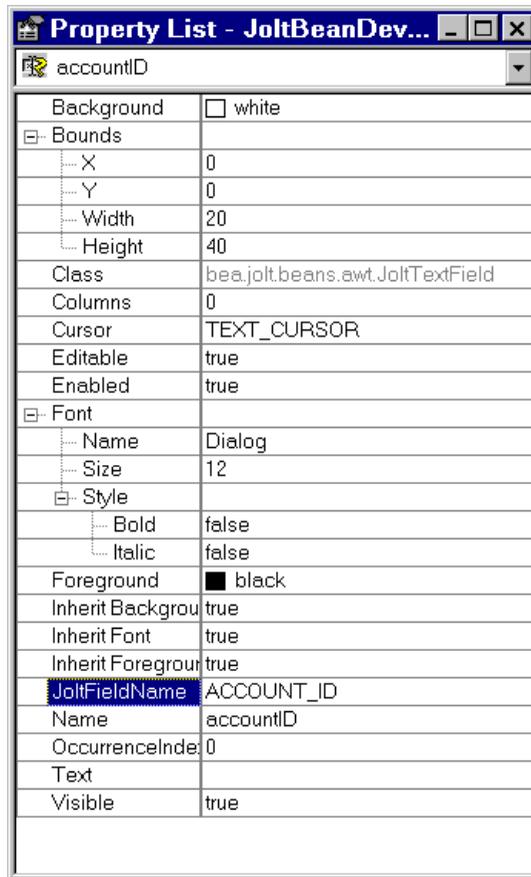
The Custom Property Editor is displayed.

- Select or type the new field name (in this example, “ACCOUNT\_ID”) and click **OK**.

The change is reflected in the Property List shown in the following figure “[Revised JoltFieldName in the JoltTextField Property List](#)” and on the text field shown on the figure “[Example of JoltBeans on the Form Designer with Property Changes](#)” on page 5-21.

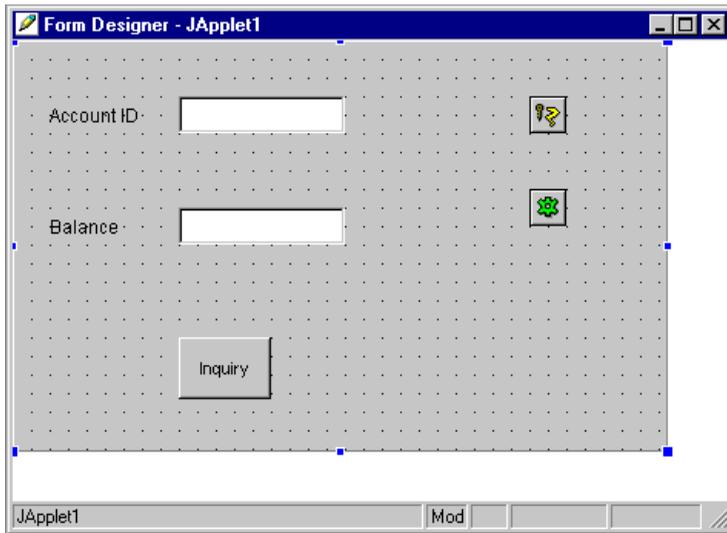
**Note:** The properties that are visible in the Custom Property Editor are cached locally; therefore, if the source database is modified you must use the Refresh button to see the current, available properties.

Figure 5-8 Revised JoltFieldName in the JoltTextField Property List



The following figure “[Example of JoltBeans on the Form Designer with Property Changes](#)” illustrates how the text on the button and the textfield changes after the text is added to the property list fields for these beans.

Figure 5-9 Example of JoltBeans on the Form Designer with Property Changes



7. After you set the properties to the right values (refer to the table [“Required and Recommended Property Values”](#) on page 5-18 for additional information), define how the beans will interact by wiring them together using the Visual Café Interaction Wizard. Refer to [“Wiring the JoltBeans Together”](#) for details.

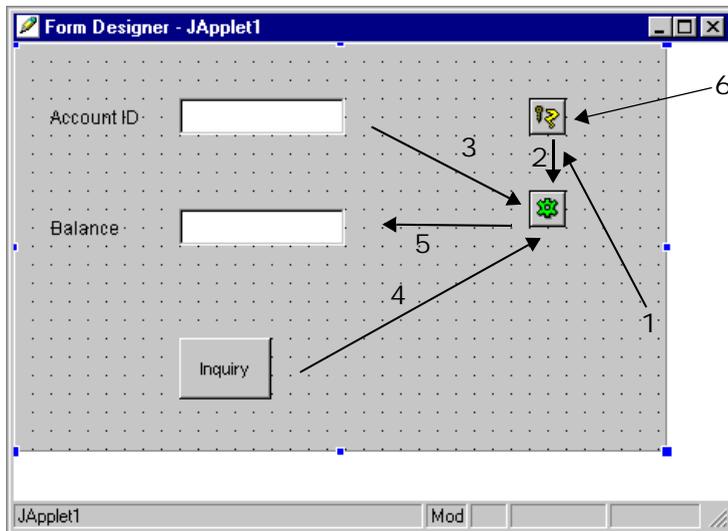
## Wiring the JoltBeans Together

After all the beans are positioned on your form and the properties are set, you must wire the beans and their events together. The following figure “[Sequence in Which JoltBeans Are Wired](#)” illustrates an example of the flow to help you determine the correct order in which to wire the beans.

Wiring the beans allows you to establish event source-listener relationships between various beans on the form. For example, the JoltServiceBean is a listener of ActionEvents from the button and invokes `callService()` when the event is received. Use the Visual Café Interaction Wizard to wire the beans together.

[Figure 5-10](#) shows the sequence in which you will wire the beans together to create this sample applet. The numbers in this figure correspond to the numbered steps that follow.

**Figure 5-10 Sequence in Which JoltBeans Are Wired**



The steps below correspond to the callouts shown in the figure “[Sequence in Which JoltBeans Are Wired](#)” on page 5-23. Each of the steps below is detailed in the sections that follow.

[Step 1: Wire the JoltSessionBean Logon](#)

[Step 2: Wire JoltSessionBean to JoltServiceBean Using PropertyChange](#)

[Step 3: Wire the accountID JoltTextField as Input to the JoltServiceBean Using JoltInputEvent](#)

Step 4: Wire Button to JoltServiceBean Using JoltAction

Step 5: Wire JoltServiceBean to the Balance JoltTextField Using JoltOutputEvent

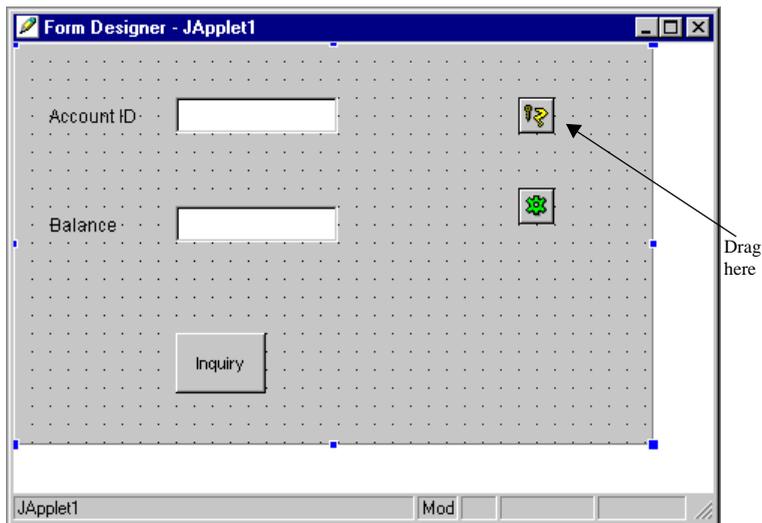
Step 6: Wire the JoltSessionBean Logoff

Step 7: Compile the Applet (not shown as a callout)

## Step 1: Wire the JoltSessionBean Logon

1. In the Form Designer window, click the Interaction Wizard button.
2. Click in the applet window and drag a line to the JoltSessionBean as shown in [Figure 5-11](#).

Figure 5-11 Wire the Applet to the Jolt Session Bean

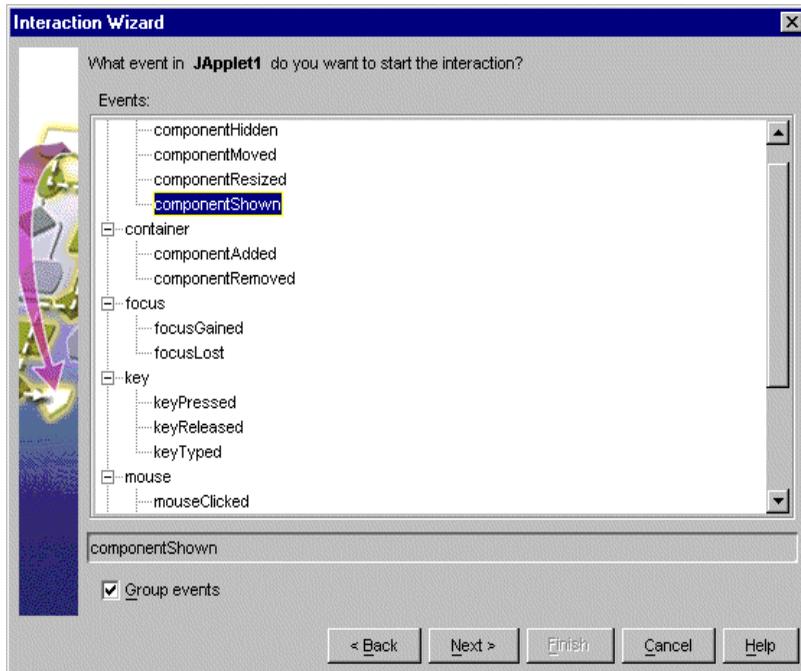


The Interaction Wizard window is displayed as shown in the figure “[Select ComponentShown Event](#)” on [page 5-25](#), with the prompt:

**What event in JApplet1 do you want to start the interaction?**

3. Select componentShown in the Interaction Wizard window as the event with which you want to start the interaction, as shown in [Figure 5-12](#).

Figure 5-12 Select ComponentShown Event



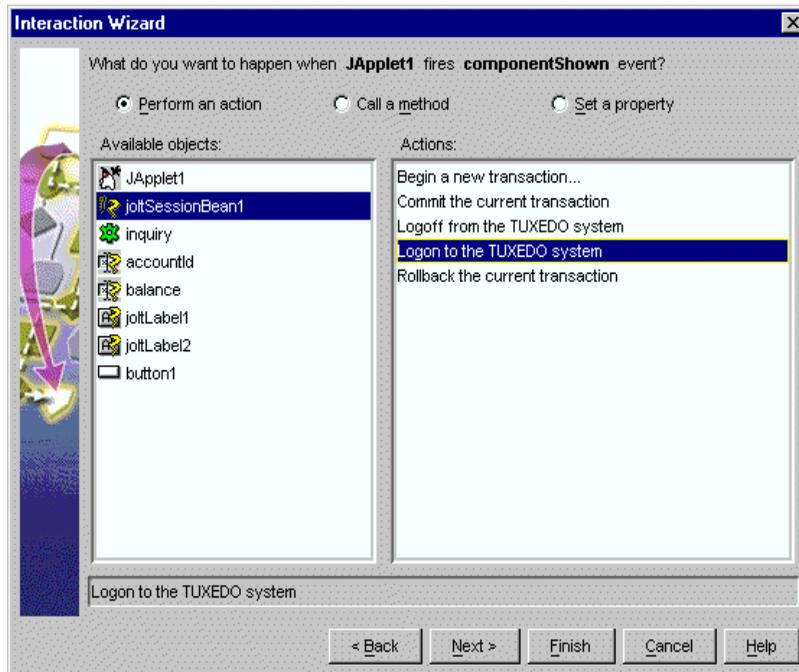
4. Click Next.

The Interaction Wizard window is displayed, as shown in the figure “[Select Logon to the Tuxedo System Action](#)” on page 5-26, with the prompt:

**What do you want to happen when JApplet1 fires componentShown event?**

5. With the **Perform an action** radio button enabled, select the action **Logon to the TUXEDO system**, as shown in [Figure 5-13](#).

Figure 5-13 Select Logon to the Tuxedo System Action

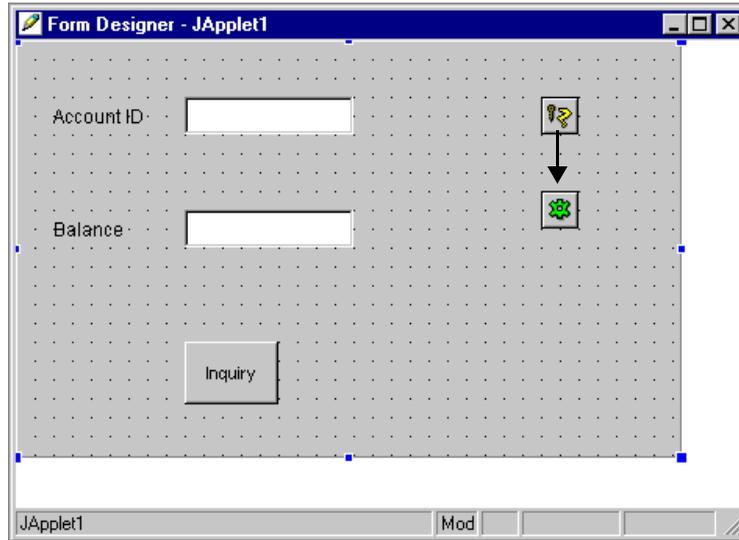


6. Click **Finish**.

Completing “[Step 1: Wire the JoltSessionBean Logon](#)” enables the `logon()` method of the `JoltSessionBean` to be triggered by an applet (for example, `ComponentShown`) that is sent when the applet is opened for the first time.

## Step 2: Wire JoltSessionBean to JoltServiceBean Using PropertyChange

1. Click the Interaction Tool icon in the toolbar of the Visual Café Form Designer window to display the bean components.
2. Click on the `JoltSessionBean` and drag a line to the `JoltServiceBean`, as shown in [Figure 5-14](#).

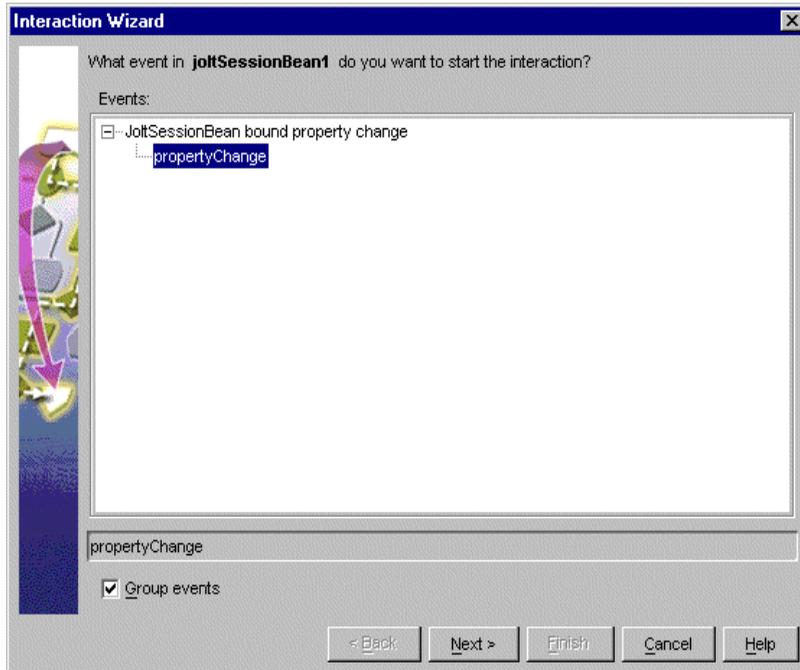
**Figure 5-14 Wire the JoltSessionBean to the JoltServiceBean**

The Interaction Wizard window is displayed as shown in the figure “[Select propertyChange Event](#)” on page 5-28, with the prompt:

**What event in joltSessionBean1 do you want to start the interaction?**

3. Select **propertyChange** as the event that starts the interaction, as shown in [Figure 5-15](#).

**Figure 5-15 Select propertyChange Event**



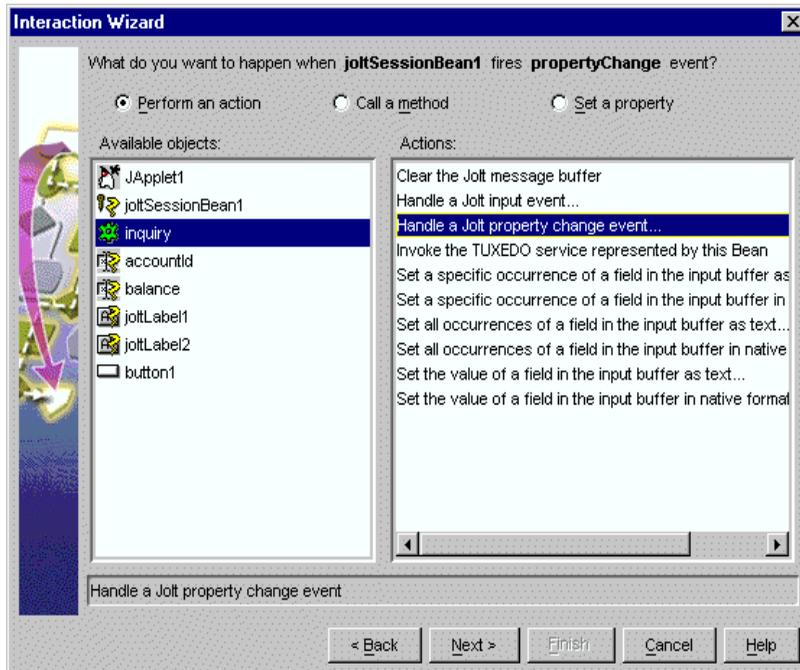
4. Click Next.

The Interaction Wizard window is displayed as shown in the figure “[Select Handle a Jolt property change event...](#)” on page 5-29, with the prompt:

**What do you want to happen when joltSessionBean1 fires propertyChange event?**

5. Select **Handle a Jolt property change event** as the method, as shown in [Figure 5-16](#).

Figure 5-16 Select Handle a Jolt property change event...



6. Click **Next**.

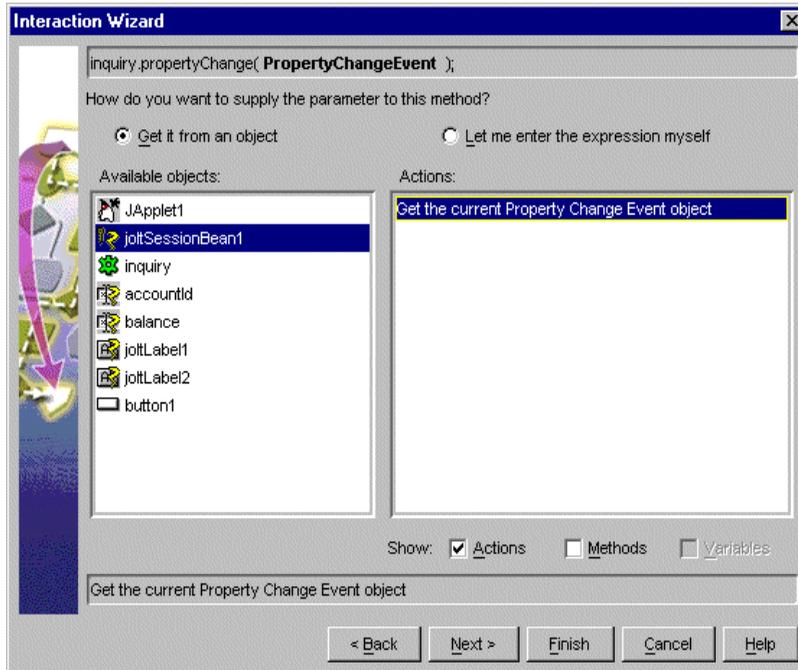
The Interaction Wizard window is displayed as shown in the figure “[Select joltSessionBean1](#)” on page 5-30, with the prompt:

**How do you want to supply the parameter to this method?**

and a list of available objects and actions from which to choose.

7. Select **joltSessionBean1** as the object that supplies the action, as shown in the following figure.
8. Select **Get the current Property Change Event** object as the action, also as shown in [Figure 5-17](#).

Figure 5-17 Select joltSessionBean1



9. Click Finish.

Completing “[Step 2: Wire JoltSessionBean to JoltServiceBean Using PropertyChange](#)” enables the JoltSessionBean to send a propertyChange event when logon () completes. The JoltServiceBean listens to this event and associates its service with this session.

### Step 3: Wire the accountId JoltTextField as Input to the JoltServiceBean Using JoltInputEvent

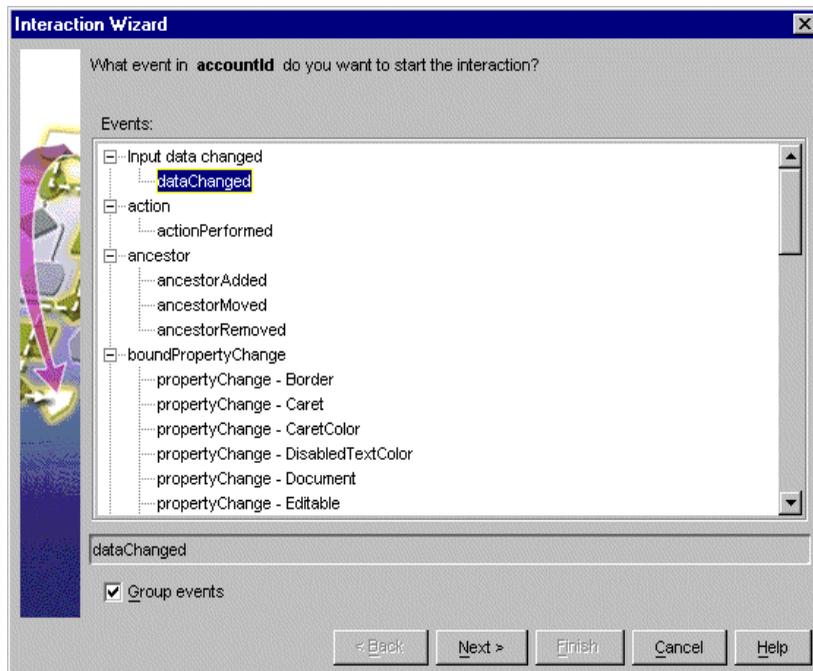
1. Click the Interaction Wizard icon in the Visual Café Form Designer window.
2. Select the **accountId JoltTextField** bean and drag a line to the JoltServiceBean.

The Interaction Wizard window is displayed, as shown in the following figure, with the prompt:

**What event in accountId do you want to start the interaction?**

3. Select **dataChanged** as the event, as shown in [Figure 5-18](#).

**Figure 5-18 Select dataChanged Event**



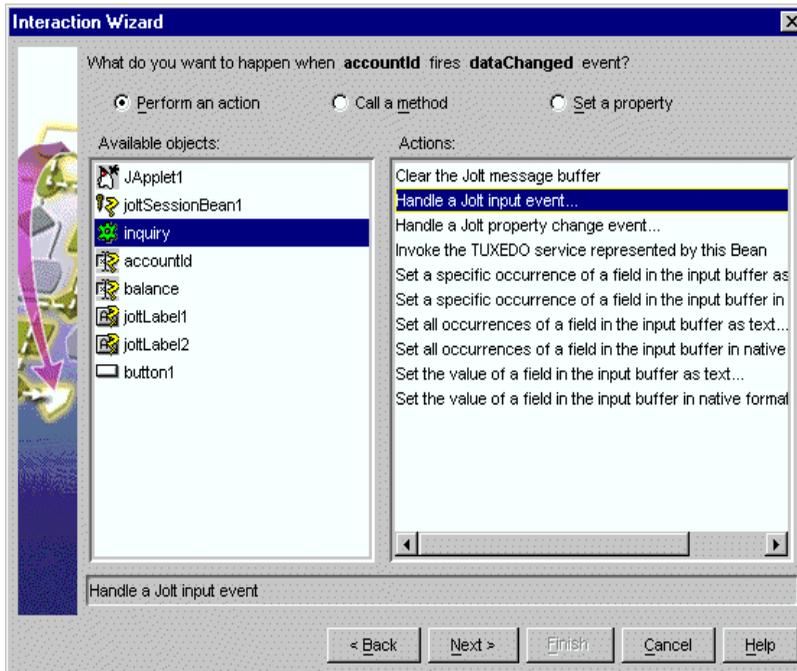
4. Click **Next**.

The Interaction Wizard window is displayed as shown in the figure [“Select inquiry Object and Handle a Jolt input event Action”](#) on [page 5-32](#), with the prompt:

**What do you want to happen when accountId fires dataChanged event?**

5. Select the joltServiceBean **inquiry** as the object supplying the parameter, as shown in the following figure.
6. Select Handle a jolt input event as the action, also as shown in [Figure 5-19](#).

Figure 5-19 Select inquiry Object and Handle a Jolt input event Action



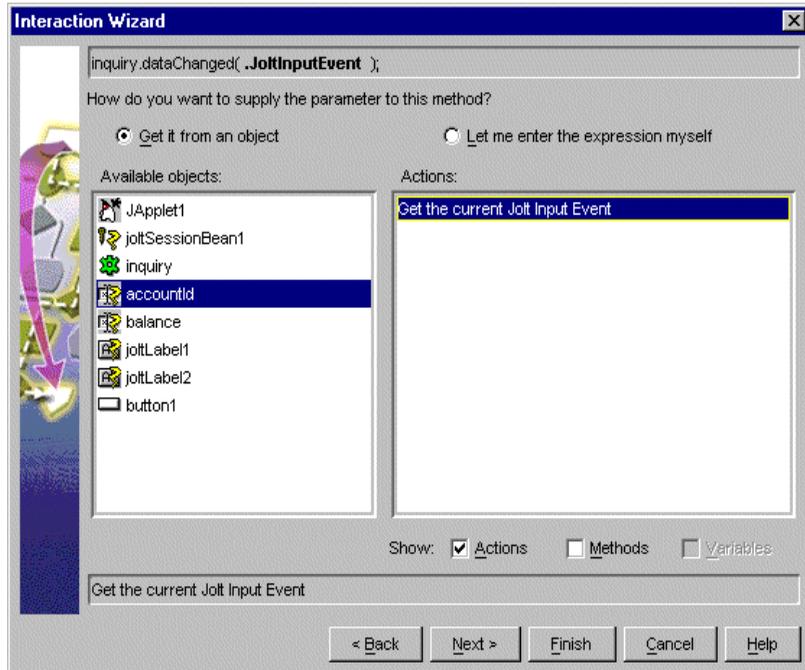
7. Click **Next**.

The Interaction Wizard window is displayed as shown in [“Select accountId Object and Get the current Jolt Input Event Action” on page 5-33](#), with the prompt:

**How do you want to supply the parameter to this method?**

and a list of available objects and actions from which to choose.

8. Select **accountId** as the object, as shown in the following figure.
9. Select **get the current Jolt Input Event** as the action, also as shown in [Figure 5-20](#).

**Figure 5-20 Select accountId Object and Get the current Jolt Input Event Action**

10. Click **Finish**.

Completing “[Step 3: Wire the accountId JoltTextField as Input to the JoltServiceBean Using JoltInputEvent](#)” enables you to type the account number in the first text field. The JoltFieldName property of this JoltTextField is set to “ACCOUNT\_ID”. Whenever the text inside this text box changes, it sends a JoltInputEvent to the JoltServiceBean. (The JoltServiceBean listens to JoltInputEvents from this text box.) The JoltInputEvent object contains the name, value, and occurrence index of the field.

## Step 4: Wire Button to JoltServiceBean Using JoltAction

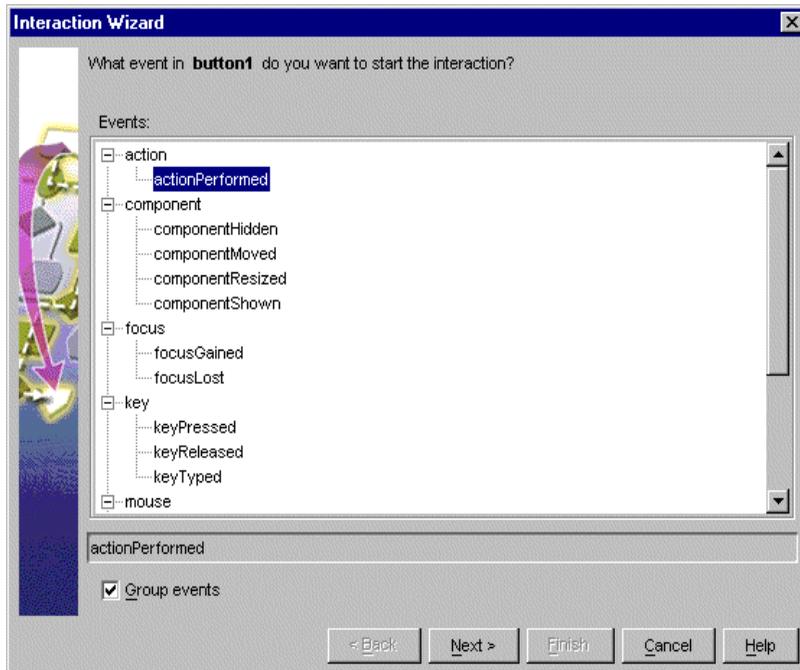
1. Click the Interaction Wizard icon in the Visual Café Form Designer window.
2. Click the Inquiry Button and drag a line to the JoltServiceBean.

The Interaction Wizard window is displayed as shown in the following figure, with the prompt:

**What event in button1 do you want to start the interaction?**

3. Select **actionPerformed** as the event, as shown in [Figure 5-21](#).

**Figure 5-21 Select action Performed Event**



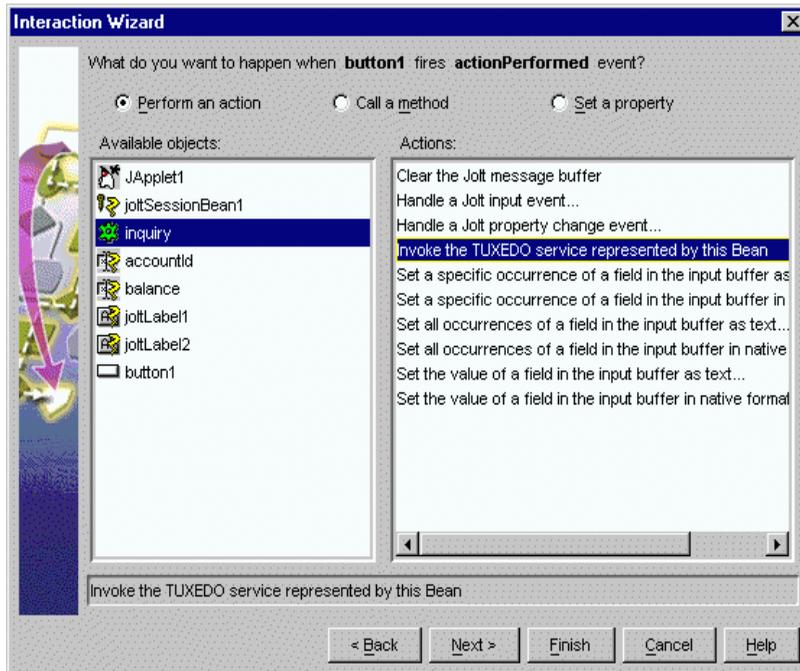
4. Click Next.

The Interaction Wizard window is displayed, as shown in the figure “[Select inquiry Object and Invoke the TUXEDO Service... Action](#)” on page 5-35, with the prompt:

**What do you want to happen when button1 fires actionPerformed event?**

5. Select **inquiry** as the object, as shown in the following figure.
6. Select **Invoke the TUXEDO Service represented by this Bean** as the action, also as shown in [Figure 5-22](#).

Figure 5-22 Select inquiry Object and Invoke the TUXEDO Service... Action



7. Click **Finish**.

Completing “[Step 4: Wire Button to JoltServiceBean Using JoltAction](#)” enables the `callService()` method of the `JoltServiceBean` to be triggered by an `ActionEvent` from the Inquiry button.

## Step 5: Wire JoltServiceBean to the Balance JoltTextField Using JoltOutputEvent

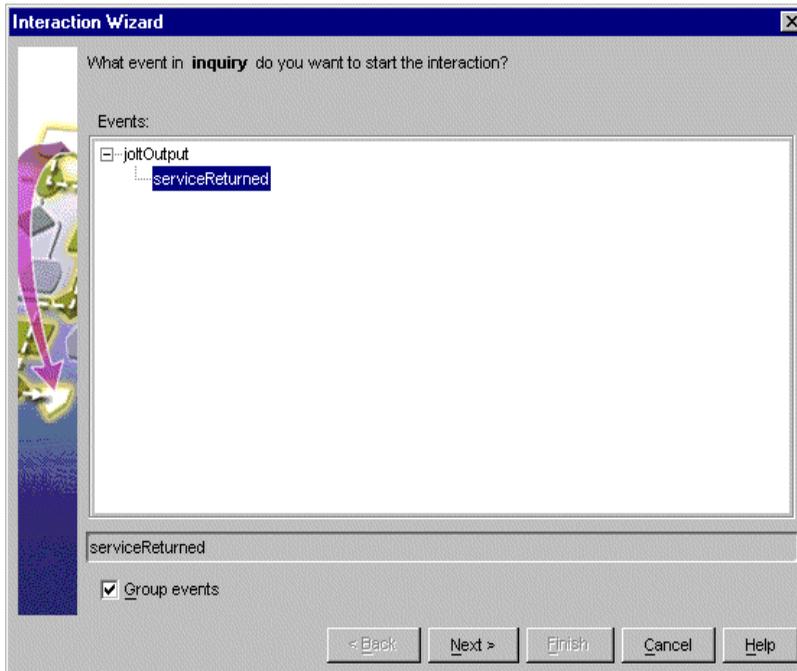
1. Click the Interaction Wizard icon in the Visual Café Form Designer window.
2. Select the `JoltServiceBean` and drag a line to the `AmountJoltTextField` bean.

The Interaction Wizard is displayed, as shown in the following figure, with the prompt:

**What event in inquiry do you want to start the interaction?**

3. Select **serviceReturned** as the event, as shown in [Figure 5-23](#).

Figure 5-23 Select ServiceReturned Event



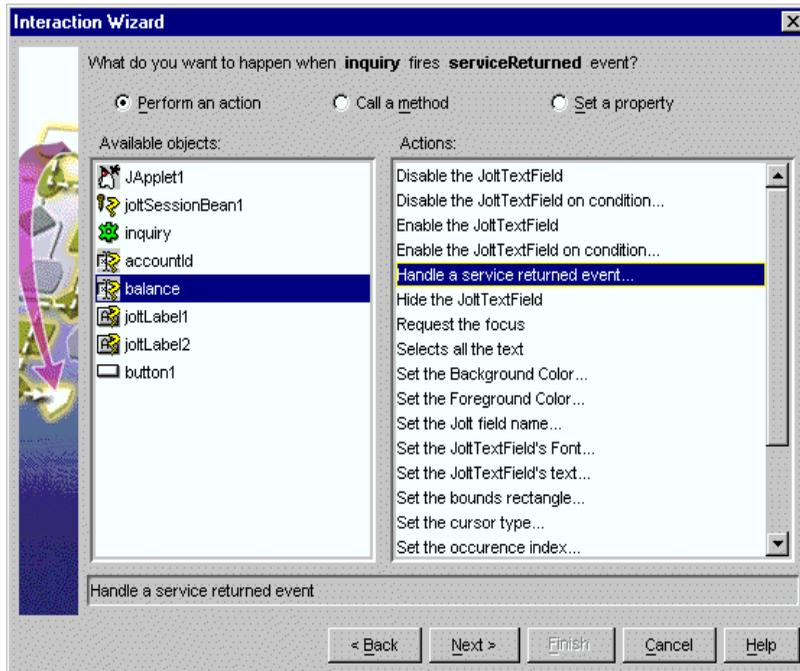
4. Click Next.

The Interaction Wizard window is displayed, as shown in the figure “[Select balance Object and Handle a service returned event Action](#)” on page 5-37, with the prompt:

**What do you want to happen when inquiry fires serviceReturned event?**

5. Select **balance** as the object, as shown in the following figure.
6. Select **Handle a service returned event...** as the action, also as shown in the following figure.

Figure 5-24 Select balance Object and Handle a service returned event Action



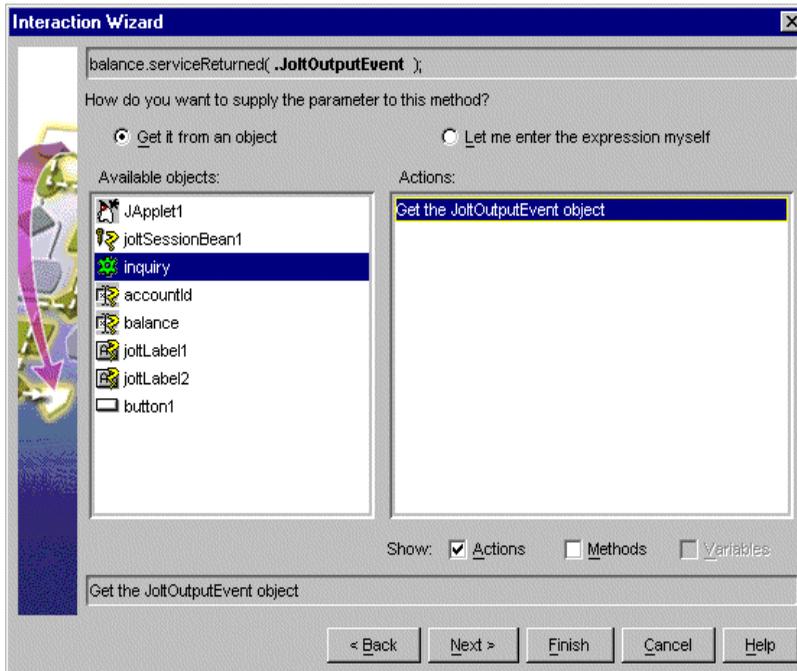
7. Click **Next**.

The Interaction Wizard window is displayed, as shown in the figure “[Select inquiry Object and Get the JoltOutputEvent object Action](#)” on page 5-38, with the prompt:

**How do you want to supply the parameter to this method?**

8. Select **inquiry** as the object, as shown in the following figure.
9. Select **Get the JoltOutputEvent** object as the action, also as shown in [Figure 5-25](#).

Figure 5-25 Select inquiry Object and Get the JoltOutputEvent object Action



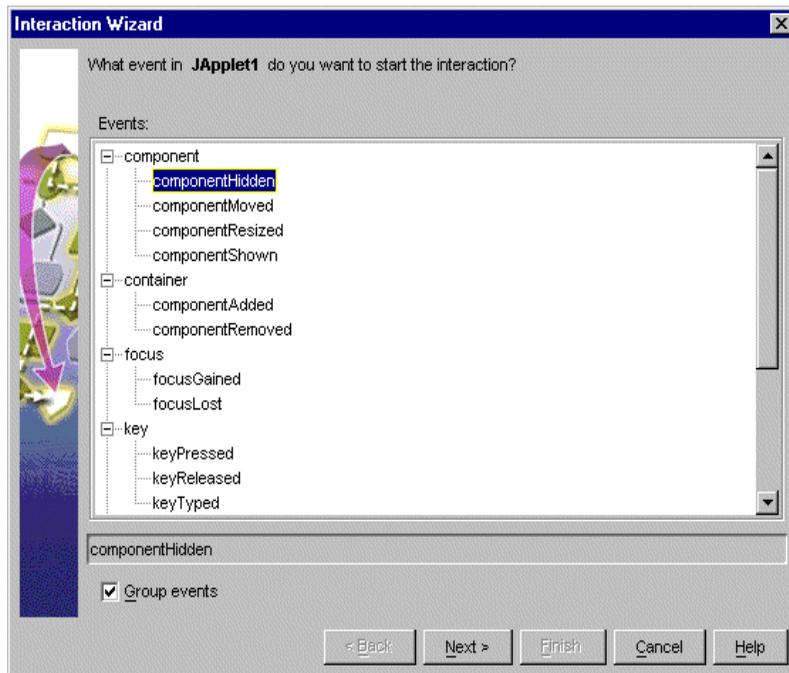
10. Click **Finish**.

Completing “[Step 5: Wire JoltServiceBean to the Balance JoltTextField Using JoltOutputEvent](#)” allows the JoltServiceBean to send a JoltOutputEvent when it receives reply data from the remote service. The JoltOutputEvent object contains methods to access fields in the output buffer. The JoltTextField displays the result of the INQUIRY service.

## Step 6: Wire the JoltSessionBean Logoff

1. Click the Interaction Wizard icon in the Visual Café Form Designer window.
2. Click in the applet window (not on another bean) and drag a line to the JoltSessionBean.  
The Interaction Wizard is displayed, as shown in the following figure, with the prompt:  
**What event in JApplet1 do you want to start the interaction?**
3. Select **componentHidden** as the event, as shown in [Figure 5-26](#).

Figure 5-26 Select componentHidden Event



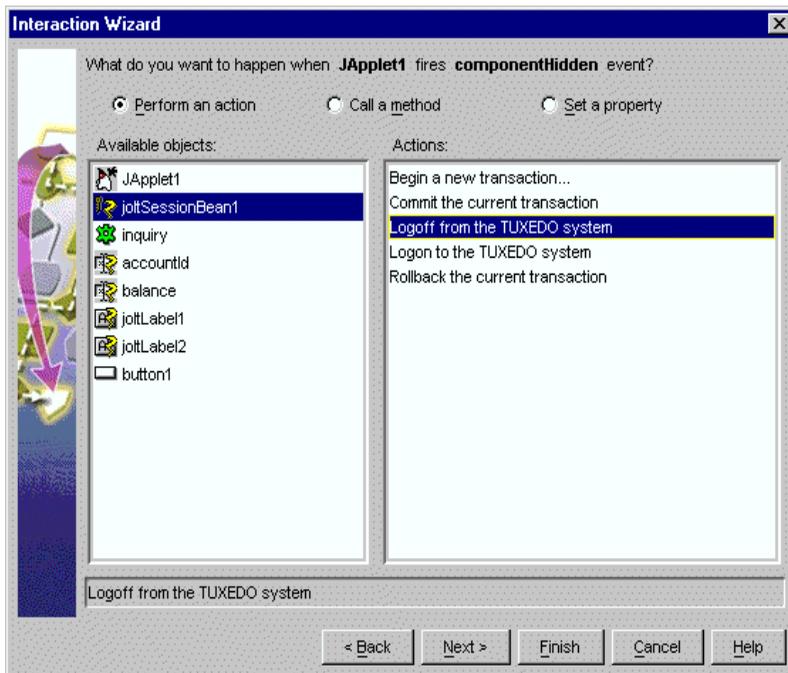
4. Click Next.

The Interaction Wizard window is displayed, as shown in the figure “[Select joltSessionBean1 Object and Logoff from the Tuxedo System Action](#)” on page 5-40, with the prompt:

**What do you want to happen when JApplet1 fires componentHidden event?**

5. Select **joltSessionBean1** as the object, as shown in the following figure.
6. Select **Logoff from the TUXEDO system** as the action, also as shown in [Figure 5-27](#).

Figure 5-27 Select joltSessionBean1 Object and Logoff from the Tuxedo System Action



7. Click **Finish**.

Completing “[Step 6: Wire the JoltSessionBean Logoff](#)” enables the `logoff()` method of the `JoltSessionBean` to be triggered by an applet (for example, `componentHidden`) that is sent when the applet gets hidden.

## Step 7: Compile the Applet

After wiring the JoltBeans together, compile the applet. It is also recommended that you fill in the empty catch blocks for exceptions. Check the message window for any compilation errors and exceptions.

For additional information see the following section “[Using the Oracle Tuxedo Service Metadata Repository and Setting the Property Values.](#)” Also refer to the table “[JoltBean Specific Properties](#)” on page 5-41 and the figure “[JoltServiceBean Property Editor](#)” on page 5-42.

## Running the Sample Application

To run the sample application, you must have the Oracle Tuxedo server running. Then enter an account number in the Account ID textfield. You can use any of the account numbers included in the BANKAPP database. Following are two examples of account numbers you can use to test the sample application:

- 80001
- 50050

# Using the Oracle Tuxedo Service Metadata Repository and Setting the Property Values

Custom Property Editors are provided for the following properties:

- JoltFieldName (Jolt-aware AWT beans)
- serviceName (JoltServiceBean)

The Property Editor, accessed from the Property List, includes dialog boxes that are used to add or modify the properties. You can invoke the boxes from the Property List by selecting the button with the ellipsis (...) that is next to the value of the corresponding property value.

Some JoltBeans require input to the Property List field. The beans are listed in [Table 5-6](#).

**Table 5-6 JoltBean Specific Properties**

JoltBean	Property	Input Description
JoltSessionBean	appAddress	e.g., //host:port
	userName, Password or AppPassword	Type your Oracle Tuxedo username and passwords.
JoltServiceBean	serviceName	INQUIRY, for example.
	isTransactional	Set to <code>true</code> if the service needs to be executed within a transaction. Set <code>isTransactional</code> to <code>false</code> if the service does not require a transaction.
JoltUserEventBean	eventName filter	Refer to the Oracle Tuxedo <code>tpssubscribe</code> calls.

**Table 5-6 JoltBean Specific Properties (Continued)**

JoltBean	Property	Input Description
All Jolt-aware GUI beans	joltFieldName occurrenceIndex	ACCOUNT_ID, for example Multiple fields of the same name. Index starts at 0.
JoltCheckbox	TrueValue and FalseValue	The field value corresponding to the state of the checkbox.

The property editor reads cached information from the repository and returns names of the available services and data elements in a list box. An example of the ServiceName property editor is shown in the following figure “[JoltServiceBean Property Editor.](#)”

To add or modify a property bean, follow these steps:

1. Select the service name by clicking on the ellipsis in the **ServiceName** field shown in [Figure 5-28](#).

**Figure 5-28 JoltServiceBean Property Editor**



The Custom Property Editor for ServiceName shown in [Figure 5-29](#) is displayed.

**Figure 5-29 Custom Property Editor for ServiceName**

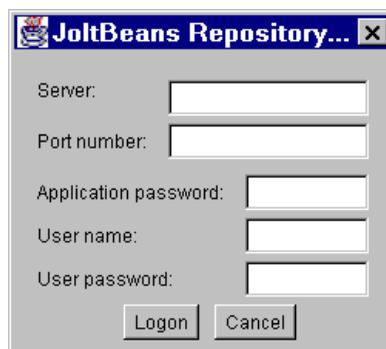


**Note:** If you cannot or do not want to connect to the Repository database, type the service name in the text box and skip to Step 7.

2. If you are not logged on, make sure the Jolt Server is running and select **Logon**.

The JoltBeans Repository Logon shown in [Figure 5-30](#) is displayed.

**Figure 5-30 JoltBeans Repository Logon**

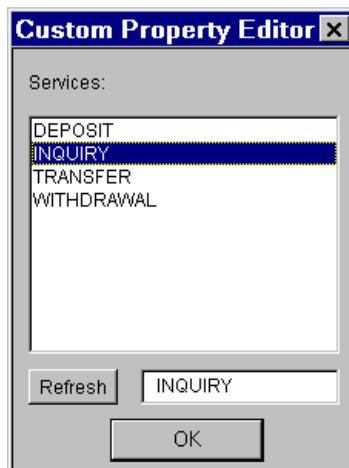


3. Type the Oracle Tuxedo or Jolt Relay Machine name in the Server field and the JSL or Jolt Relay in the Port number field.
4. Type the password and username information (if required) and click **Logon**.

The Custom Property Editor loads its cache from the repository and is displayed, as shown in the following figure “[Property Editor with Selected Service.](#)”

5. Select the appropriate service name from the list box, as shown in the following figure.
6. Enter the property value (service or field name) directly.  
A text box is provided.
7. Click **OK** in the Custom Property Editor dialog.  
The bean property is set with the contents of the text box.

**Figure 5-31 Property Editor with Selected Service**



8. Click **OK** in the Custom Property Editor dialog box again.

## JoltBeans Programming Tasks

Additional programming tasks include:

- [Using Transactions with JoltBeans](#)
- [Using Custom GUI Elements with the JoltService Bean](#)

## Using Transactions with JoltBeans

Your Oracle Tuxedo application services may have functionality that updates your database. If so, you can use transactions with JoltBeans (for example, in the sample, BANKAPP, the services TRANSFER and WITHDRAWAL update the database of BANKAPP). If your application service is read-only (such as INQUIRY), you do not need to use transactions.

The following example shows how to use transactions with JoltBeans.

1. The `setTransactional (true)` method is called on the `JoltServiceBean`. (`isTransactional` is a Boolean property of the `JoltServiceBean`.)
2. The `beginTransaction()` method is called on the `JoltSessionBean`.
3. The `callService()` method is called on the `JoltServiceBean`.
4. Depending on the outcome of the service call, the `commitTransaction()` or `rollbackTransaction()` method is called on the `JoltSessionBean`.

## Using Custom GUI Elements with the JoltService Bean

JoltBeans provides a limited set of Jolt-enabled GUI components. You can also use controls that are not Jolt-enabled together with the JoltServiceBean. You can link controls to the JoltServiceBean that display output information of the service represented by the JoltServiceBean. You can also link controls that display input information.

For example, a GUI element that uses an adapter class to implement the JoltOutputListener interface can listen to JoltOutputEvents. The JoltServiceBean as the event source for JoltOutputEvents calls the `serviceReturned()` method of the adapter class when it sends a JoltOutputEvent. Inside `serviceReturned()`, the control's internal data is updated using information from the event object.

The development tool generates the adapter class when the JoltServiceBean and the GUI element are wired together.

As another example, a GUI element can call the `setInputTextValue()` method on the JoltServiceBean. The GUI element contains input data for the Oracle Tuxedo service represented by the JoltServiceBean.

As a third example, a GUI element can implement the required methods (`addJoltInputListener()` and `removeJoltInputListener()`) to act as event sources for JoltInputEvents. The JoltServiceBean acts as an event listener for these events. The control sends a JoltInputEvent when its own state changes to keep the JoltServiceBean updated with the input information.

# Using Servlet Connectivity for Oracle Tuxedo

With Oracle Jolt servlet connectivity, you can use HTTP servlets to perform server-side Java tasks in response to HTTP requests. Jolt certifies servlet connectivity with the Java Web Server versions 1.1.3 and up, and supports most other standard servlet engines. Using the Jolt session pool classes, a simple HTML client can connect to any Web server that supports generic servlets. Thus, all Jolt transactions are handled by a servlet on the Web server rather than being handled by a client applet or application.

This capability enables HTML clients to invoke Oracle Tuxedo services without directly connecting to Oracle Tuxedo. HTML clients can instead connect to a Web server, through HTTP, where the Oracle Tuxedo service request is executed by a generic servlet. Using a Jolt session, the servlet on the Web server administers the Oracle Tuxedo service request by connecting to the Oracle Tuxedo Server through the Jolt Server Handler (JSH) or the Jolt Server Listener (JSL), which then makes the Oracle Tuxedo service request.

This capability allows many types of HTML clients to make remote Oracle Tuxedo service requests. All Jolt transactions are handled on the server side without requiring any change to the original HTML client. Thus, HTML clients are allowed to be very simple and require little maintenance.

This topic includes the following sections:

- [What Is a Servlet?](#)
- [How Servlets Work with Jolt](#)
- [Writing and Registering HTTP Servlets](#)

- [Jolt Servlet Connectivity Sample](#)
- [Additional Information on Servlets](#)

## What Is a Servlet?

A servlet is any Java class that can be invoked and executed on a server, usually on behalf of a client. A servlet works on the server, while an applet works on the client. An HTTP servlet is a Java class that handles an HTTP request and delivers an HTTP response. HTTP servlets reside on an HTTP server and must extend the JavaSoft `javax.servlet.http.HttpServlet` Class so that they can run in a generic servlet engine framework.

Some advantages of using HTTP servlets are:

- They are written in a well-formed, and compiled language (Java), so are more robust than “interpreted” scripts.
- They are an integral part of the HTTP server that supports them.
- They can be protected by the robust security of the server, unlike some CGI scripts that are hazardous.
- They interact with the HTTP request through a well-developed programmatic interface, and so are easier to write and less prone to errors.

## How Servlets Work with Jolt

With Jolt servlet connectivity, any generic HTTP servlet allows you to take advantage of the Jolt features. Jolt servlets handle HTTP requests using the following Jolt classes:

- **ServletDataSet**
- `ServletPoolManagerConfig`
- `ServletResult`
- `ServletSessionPool`
- `ServletSessionPoolManager`

## The Jolt Servlet Connectivity Classes

Following are descriptions of the Jolt servlet connectivity classes.

**ServletDataSet**

This class contains data elements that represent the input and output parameters of an Oracle Tuxedo service. It provides a method to import the HTML field names and values from a `javax.servlet.http.HttpServletRequest` object.

**ServletPoolManagerConfig**

This class is the startup class for a Jolt Session Pool Manager and one or more associated Jolt session pools. It creates the session pool manager if needed and starts a session pool with a minimum number of sessions. Jolt Session Pool Manager internally keeps track of one or more named session pools.

This class is derived from `bea.jolt.pool.PoolManagerConfig` and allows the caller to pass a `Properties` or `Hashtable` object to the static `startup()` method to create a session pool and the static `getSessionPoolManager()` method to get the session pool manager of `bea.jolt.pool.servlet.ServletSessionPoolManager` class.

**ServletResult**

This class provides methods to retrieve each field in a `ServletResult` object as a `String`.

**ServletSessionPool**

This class provides a session pool for use in a Java servlet. A session pool represents one or more connections (sessions) to an Oracle Tuxedo system. This class provides call methods that accept input parameters for an Oracle Tuxedo service as a `javax.servlet.http.HttpServletRequest` object.

**ServletSessionPoolManager**

This class is a servlet-specific session pool manager. It manages a collection of one or more session pools of class `ServletSessionPool`. This class provides methods that are used to create both the `ServletSessionPoolManager` itself and the session pools that it contains. These methods are part of the administrative API for a session pool.

## Writing and Registering HTTP Servlets

Before writing and registering HTTP servlets, you must first import the packages that support Jolt servlet connectivity (`jolt.jar`, `joltjse.jar`, `servlet.jar`). HTTP servlets must extend `javax.servlet.http.HttpServlet`. After you write your HTTP servlets, you register them with a Web server that supports generic servlets. Your custom servlets are treated exactly like the standard HTTP servlets that provide the HTTP capabilities.

Each HTTP servlet is registered against a specific URL pattern, so that when a matching URL is requested, the corresponding servlet is called upon to handle the request.

Refer to the documentation for your particular Web server for instructions on how to register servlets.

# Jolt Servlet Connectivity Sample

The Jolt software includes three sample applications that demonstrate servlet connectivity using the Jolt servlet classes. The three samples are:

- [SimpApp Sample](#)
- [BankApp Sample](#)
- [Admin Sample](#)

Refer to these samples to see code examples of how to use the Jolt servlet classes in your own servlets.

## Viewing the Sample Servlet Applications

To view the code for the Jolt sample applications, you need to install the Jolt API client classes (usually chosen as an option when installing Jolt). Once the classes are installed in your directory of choice, navigate to the following directory to see the sample application files:

```
<Installation directory>\udataobj\jolt\examples\servlet
```

To view the sample code, use a text editor such as Microsoft Notepad to open the Java files for each sample application.

## SimpApp Sample

A sample application named `simpapp` is included with Jolt. The `simpapp` application illustrates how the servlet uses Servlet Connectivity for Oracle Tuxedo. The following servlet tasks are illustrated by the `SimpApp` sample:

- Using a property file to create a session pool
- Getting the session pool manager
- Retrieving the session pool by name
- Invoking an Oracle Tuxedo service
- Processing the result set

This example demonstrates how a servlet can connect to Oracle Tuxedo and call upon one of its services; it should be invoked from the `simpapp.html` file. The servlet creates a session pool manager at initialization, which is used to obtain a session when the `doPost()` method is

invoked. This session is used to connect to a service in Oracle Tuxedo with a name described by the posted “SVCNAME” argument. In this example the service is called "TOUPPER", which transposes the posted “STRING” argument text into uppercase, and returns the result to the client browser within some generated HTML.

**Note:** The WebLogic Server is used in this example.

## Requirements for Running the SimpApp Sample

The requirements for running the SimpApp sample are:

- Any Web application server with Servlet JSDK 1.1 or above
- Oracle Tuxedo 8.0 or later with SimpApp sample running
- Oracle Jolt

## Installing the SimpApp Sample

1. Install the Jolt class library (`jolt.jar`) and Servlet Connectivity for Oracle Tuxedo class library (`joltjse.jar`) on the Web application server. Extract the class files if it is required by your Web application server.
2. Compile the `SimpAppServlet.java`. Make sure that you include the standard JDK 1.1.x `classes.zip`, JSDK 1.1 classes, Jolt class library, and Servlet Connectivity for Oracle Tuxedo class library in the classpath.

```
javac -classpath $(JAVA_HOME)/lib/classes.zip:$(JSDK)/lib/servlet.jar:  
           $(JOLTHOME)/jolt.jar:$(JOLTHOME)/joltjse.jar:./classes  
       -d ./classes SimpAppServlet.java
```

**Note:** The package name of the `SimpAppServlet` is `examples.jolt.servlet.simpapp`.

3. Put the `simpapp.html` and `simpapp.properties` files in the public HTML directory.
4. Modify the `simpapp.properties` file. Change the “appaddrlist” and “failoverlist” with the proper Jolt server hosts and ports. Specify the proper Oracle Tuxedo authentication information if the SimpApp has security turned on. For example:

```
#simpapp  
#Fri Apr 16 00:43:30 PDT 1999  
  
poolname=simpapp  
  
appaddrlist=//host:7000, //host:8000
```

```
failoverlist=//backup:9000  
minpoolsize=1  
maxpoolsize=3  
userrole=tester  
apppassword=appPass  
username=guest  
userpassword=myPass
```

5. Register “Simpapp” for the SimpAppServlet. Consult your Web application server for details. If you are using Oracle WebLogic Server, add the following section of the `config.xml` file:

```
<Application  
  Deployed="true"  
  Name="simpapp"  
  Path=".\\config\\mydomain\\applications"  
>  
  <WebAppComponent  
    Name="simpapp"  
    Targets="myserver"  
    URI="simpapp"  
  />  
</Application>
```

6. To access the SimpApp initial page “simpapp.html,” type:

```
http://mywebserver:8080/simpapp.html
```

# BankApp Sample

The `bankapp` application illustrates how the servlet is written with `PageCompiledServlet` with Servlet Connectivity for Oracle Tuxedo. `bankapp` illustrates how to:

- Use a property file to create a session pool
- Get the session pool manager
- Retrieve a session pool by name
- Invoke an Oracle Tuxedo service
- Process the result set

## Requirements for Running the BankApp Sample

Following are the requirements for running the BankApp sample:

- Any Web application server with Servlet JSDK 1.1 or above
- Oracle Tuxedo 8.0 or later with BankApp sample running
- Oracle Jolt

## Installation Instructions

1. Install the Jolt class library (`jolt.jar`) and Servlet Connectivity for Oracle Tuxedo class library (`joltjse.jar`) to the Web application server. Extract the class files if it is required by your Web application server.
2. Copy all HTML, JHTML and `bankapp.properties` files to the public HTML directory of the Web application server (for example, `$WEBLOGIC/myserver/public_html` for WebLogic):

```
bankapp.properties
tellerForm.html
inquiryForm.html
depositForm.html
withdrawalForm.html
transferForm.html
InquiryServlet.jhtml
```

```
DepositServlet.jhtml  
WithdrawalServlet.jhtml  
TransferServlet.jhtml
```

3. **Modify the `bankapp.properties` file. Change the “`appaddrlist`” and “`failoverlist`” with the proper Jolt server hosts and ports. Specify the proper Oracle Tuxedo authentication information if the BankApp has security turned on. For example:**

```
#bankapp  
#Fri Apr 16 00:43:30 PDT 1999  
poolname=bankapp  
appaddrlist=//host:8000, //host:7000  
failoverlist=//backup:9000  
minpoolsize=2  
maxpoolsize=10  
userrole=teller  
apppassword=appPass  
username=JaneDoe  
userpassword=myPass
```

4. If applicable, turn on the automatic page compilation for JHTML from your servlet engine. Consult the user manual of your Web application server for details.
5. To access BankApp through Servlet Connectivity for Oracle Tuxedo, use the following URL in your favorite browser:

```
http://mywebserver:8080/tellerForm.html
```

## Admin Sample

The Admin sample application illustrates the following servlet tasks:

- Using the administrative API to control the session pools
- Retrieving the statistics through PageCompiledServlet in Servlet Connectivity for Oracle Tuxedo

## Requirements for Running the Admin Sample

Following are the requirements for running the Admin sample:

- Any Web application server with Servlet JSDK 1.1 or above
- Oracle Jolt

## Installation Instructions

1. Install the Jolt class library and Servlet Connectivity for Oracle Tuxedo class library on the Web application server.
2. Copy all JHTML files to the public HTML directory (for example, `$WEBLOGIC/myserver/public_html` for WebLogic):

```
PoolList.jhtml
```

```
PoolAdmin.jhtml
```

3. To get a list of session pools, use the following URL in your favorite browser:

```
http://mywebserver:8080/PoolList.jhtml
```

## Additional Information on Servlets

For more information on writing and using servlets, refer to the following sites:

### **Oracle WebLogic Servlet Documentation**

<http://e-docs.bea.com/wls/docs81/adminguide/index.html>

<http://e-docs.bea.com/wls/docs81/servlet/index.html>

<http://e-docs.bea.com/wls/docs81/javadocs/index.html>

### **Java Servlets**

[http://jserv.java.sun.com/products/java-server/documentation/webserver1.1/index\\_developer.html](http://jserv.java.sun.com/products/java-server/documentation/webserver1.1/index_developer.html)

### **Servlet Interest Group**

<http://servlet-interest@java.sun.com>



# Migrating from Jolt Repository to Oracle Tuxedo Service Metadata Repository

Jolt Repository is deprecated in this release. All service definitions stored in Jolt repository can be loaded in the Tuxedo metadata repository using the bulk loader tool.

The original Jolt repository server `JREPSVR` is also deprecated, and all services that `JREPSVR` provided are now provided by `TMMETADATA`. If `tmloadcf` detects presence of `JREPSR` in `ubbconfig`, it automatically removes `JREPSVR` and adds `TMMETADATA` if not already configured.

Using one repository (Tuxedo metadata repository) and one server (`TMMETADATA`) improve operational effectiveness and reduces the risk of service definitions getting out of sync.

This topic includes the following sections:

- [Replacing JREPSVR with TMMETADATA in UBBCONFIG](#)
- [Loading Jolt Repository to Oracle Tuxedo Service Metadata Repository](#)
- [Sample: joltapp Migration](#)

## Notes:

- `TMMETADATA` must run on Oracle Tuxedo 12c Release 2 (12.2.2) or later.
- Oracle Tuxedo Service Metadata Repository does not support package function.
- Bulk Loader is still supported but some behaviors are changed.
  - Do not support for deleting service definitions. You should use `tmloadrepos -d service[...]` or the Metadata Editor to delete services.

- `service` (that represents the service entry stored in the Metadata Repository) and `tuxservice` (that represents the actual Oracle Tuxedo service name) must be same; otherwise you cannot update this service definition and JSH in Oracle Tuxedo 12c Release 1 (12.1.3) or earlier cannot work.
- The `servicetype` must be "service"; otherwise you cannot update this service definition.

## Replacing JREPSVR with TMMETADATA in UBBCONFIG

Oracle Tuxedo command `tmloadcf` browses your UBBCONFIG and creates a new one (you can compare these two UBBCONFIG files to see the difference after invoking `tmloadcf`).

In your UBBCONFIG,

- If JREPSVR is configured, `tmloadcf` automatically replaces JREPSVR with TMMETADATA when creating the new UBBCONFIG.

In this scenario, `tmloadcf` prints a prompt like this:

```
CMDTUX_CAT:8401: WARNING: The JoltRepository Server has been deprecated
and replaced with the Tuxedo Servicata Server
```

- If both JREPSVR and TMMETADATA are configured, `tmloadcf` just ignores JREPSVR when creating the new UBBCONFIG.

In this scenario, `tmloadcf` prints a prompt like this:

```
CMDTUX_CAT:8401: WARNING: The JoltRepository Server has been deprecated
and replaced with the Tuxedo Service Metadata Server
```

You can use `tmloadrepos/tmunloadrepos` to load Jolt Repository file to Metadata Repository file; this loading is necessary because you can access to Jolt Repository only after it is loaded to Metadata Repository. If you want to update the loaded Metadata Repository, you should remove `-r` option from TMMETADATA in UBBCONFIG. For more information, see [tmloadcf\(1\)](#).

## Loading Jolt Repository to Oracle Tuxedo Service Metadata Repository

You should use Oracle Tuxedo command `tmunloadrepos` to display Jolt Repository in plain text, which `tmloadrepos` can parse. See [tmloadrepos\(1\)](#) and [tmunloadrepos\(1\)](#) for more information.

Two parameter-level keywords, `fieldname` and `fieldindex`, are added to the Oracle Tuxedo Service Metadata Repository for this loading (and for FML/FML32 only). See [Creating The Oracle Tuxedo Service Metadata Repository](#) for more information.

## Sample: joltapp Migration

This is a sample for migrating joltapp from Jolt Repository to Oracle Tuxedo Service Metadata Repository.

1. Use `tmloadcf` to compile `UBBCONFIG`, which handles `JREPSVR` automatically. See [Listing 1-1](#) for an example.
2. Use `tmunloadrepos` to load Jolt Repository file named `jrep` (see [Listing 1-2](#)) to Oracle Tuxedo Service Metadata Repository file named `jolt.metarepos` (see [Listing 1-3](#)).
3. Use `tmloadrepos` to insert the above `jolt.metarepos` file into the original Metadata Repository file. See [Listing 1-4](#) for an example.

### Listing 1-1 Example for Compiling UBBCONFIG

---

```
$ tmloadcf -y ubb
$ tmunloadcf      #to check whether JREPSVR still existed.
```

---

### Listing 1-2 Example for jrep

---

```
$ cat jrep
!JOLT1.0
add SVC/.NUMRECS:vs=1:ex=0:bt=FML32:\
    bp:pn=NRECS:pt=integer:pf=33554435:pa=rd:ep:\
    bp:pn=STATLIN:pt=string:pf=167772163:pa=rd:ep:\
add SVC/.GETREC:vs=1:ex=0:bt=FML32:\
    bp:pn=REPNAME:pt=string:pf=167772161:pa=wr:ep:\
    bp:pn=REPVALUE:pt=string:pf=167772162:pa=rd:ep:\
    bp:pn=STATLIN:pt=string:pf=167772163:pa=rd:ep:\
```

```

add SVC/.GETSVC:vs=1:ex=0:bt=FML32:\
    bp:pn=REPNAME:pt=string:pf=167772161:pa=wr:ep:\
    bp:pn=REPVALUE:pt=string:pf=167772162:pa=rd:ep:\
    bp:pn=STATLIN:pt=string:pf=167772163:pa=rd:ep:
add SVC/.ADDREC:vs=1:ex=0:bt=FML32:\
    bp:pn=REPVALUE:pt=string:pf=167772162:pa=wr:ep:\
    bp:pn=STATLIN:pt=string:pf=167772163:pa=rd:ep:
add SVC/.DELREC:vs=1:ex=0:bt=FML32:\
    bp:pn=REPNAME:pt=string:pf=167772161:pa=wr:ep:\
    bp:pn=STATLIN:pt=string:pf=167772163:pa=rd:ep:
add SVC/.GETKEYS:vs=1:ex=1:bt=FML32:\
    bp:pn=PATTERN:pt=string:pf=167772164:pa=wr:ep:\
    bp:pn=NRECS:pt=integer:pf=33554435:pa=rd:ep:\
    bp:pn=REPNAME:pt=string:pf=167772161:po=0:pa=rd:ep:
add SVC/.GETALL:vs=1:ex=0:bt=FML32:\
    bp:pn=PATTERN:pt=string:pf=167772164:pa=wr:ep:\
    bp:pn=NRECS:pt=integer:pf=33554435:pa=rd:ep:\
    bp:pn=REPVALUE:pt=string:pf=167772162:po=0:pa=rd:ep:
add SVC/.FLUSHCACHE:vs=1:ex=0:bt=FML32:\
    bp:pn=REPNAME:pt=string:pf=167772161:po=200:pa=wr:ep:
add SVC/.GARBAGECOLLECT:vs=1:ex=0:bt=FML32:\
    bp:pn=STATLIN:pt=string:pf=167772163:pa=rd:ep:
add SVC/INQUIRY:vs=1:ex=1:bt=FML:\
    bp:pn=ACCOUNT_ID:pt=integer:pf=33554436:pa=wr:ep:\
    bp:pn=SBALANCE:pt=string:pf=167772164:pa=rd:ep:\
    bp:pn=STATLIN:pt=string:pf=167772163:pa=rd:ep:\
    bp:pn=FORMNAM:pt=string:pf=167772165:pa=rd:ep:

```

```

add SVC/WITHDRAWAL:vs=1:ex=1:bt=FML:\
    bp:pn=ACCOUNT_ID:pt=integer:pf=33554436:pa=wr:ep:\
    bp:pn=SAMOUNT:pt=string:pf=167772166:pa=wr:ep:\
    bp:pn=SBALANCE:pt=string:pf=167772164:pa=rd:ep:\
    bp:pn=STATLIN:pt=string:pf=167772163:pa=rd:ep:\
    bp:pn=FORMNAM:pt=string:pf=167772165:pa=rd:ep:
add SVC/DEPOSIT:vs=1:ex=1:bt=FML:\
    bp:pn=ACCOUNT_ID:pt=integer:pf=33554436:pa=wr:ep:\
    bp:pn=SAMOUNT:pt=string:pf=167772166:pa=wr:ep:\
    bp:pn=SBALANCE:pt=string:pf=167772164:pa=rd:ep:\
    bp:pn=BALANCE:pt=float:pf=100663303:pa=rd:ep:\
bp:pn=STATLIN:pt=string:pf=167772163:pa=rd:ep:\
bp:pn=BALANCE:pt=float:pf=100663303:pa=rd:ep:\
    bp:pn=STATLIN:pt=string:pf=167772163:pa=rd:ep:\
    bp:pn=FORMNAM:pt=string:pf=167772165:pa=rd:ep:
add SVC/TRANSFER:vs=1:ex=1:bt=FML:\
bp:pn=ACCOUNT_ID:pt=integer:pf=33554436:po=2:pa=wr:ep:\
bp:pn=SAMOUNT:pt=string:pf=167772166:pa=wr:ep:\
bp:pn=SBALANCE:pt=string:pf=167772164:po=2:pa=rd:ep:\
bp:pn=STATLIN:pt=string:pf=167772163:pa=rd:ep:\
bp:pn=FORMNAM:pt=string:pf=167772165:pa=rd:ep:
add PKG/BANKAPP:INQUIRY:DEPOSIT:WITHDRAWAL:TRANSFER:
add SVC/TOUPPER:vs=1:ex=1:bt=STRING:\
    bp:pn=STRING:pt=string:pf=167772161:pa=rw:ep:
add PKG/SIMPSERV:TOUPPER:

```

---

**Listing 1-3 Example for jolt.metarepos**

---

```
$ tmunloadrepos jrep >jolt.metarepos

# #Generated from Repository file :
/u01/common/patches/huchchen/TUX13c64/LC/bld/qa/sanity_tests/apps/joltapp/
jrep
#
#####
# service : .GETALL #
#####
service=.GETALL
export=N
inbuf=FML32
outbuf=FML32
version=1
param=PATTERN
type=string
access=in
param=NRECS
type=integer
access=out
param=REPVALUE
count=0
type=string
access=out

#####
```

```
# service : .GETKEYS #
#####

service=.GETKEYS
export=Y
inbuf=FML32
outbuf=FML32
version=1
param=PATTERN
type=string
access=in
param=NRECS
type=integer
access=out
param=REPNAME
count=0
type=string
access=out

#####

# service : .ADDREC #
#####

service=.ADDREC
export=N
inbuf=FML32
outbuf=FML32
version=1
param=REPVALUE
```

```
type=string
access=in
param=STATLIN
type=string
access=out

#####
# service : TRANSFER #
#####
service=TRANSFER
export=Y
inbuf=FML
outbuf=FML
version=1
param=ACCOUNT_ID
count=2
type=integer
access=in
param=SAMOUNT
type=string
access=in
param=SBALANCE
count=2
type=string
access=out
param=STATLIN
type=string
```

```
access=out
param=FORMNAM
type=string
access=out

#####
# service : WITHDRAWAL #
#####

service=WITHDRAWAL
export=Y
inbuf=FML
outbuf=FML
version=1
param=ACCOUNT_ID
type=integer
access=in
param=SAMOUNT
type=string
access=in
param=SBALANCE
type=string
access=out
param=STATLIN
type=string
access=out
param=FORMNAM
type=string
```

access=out

#####

# service : .GARBAGECOLLECT #

#####

service=.GARBAGECOLLECT

export=N

inbuf=FML32

outbuf=FML32

version=1

param=STATLIN

type=string

access=out

#####

# service : TOUPPER #

#####

service=TOUPPER

export=Y

inbuf=STRING

outbuf=STRING

version=1

param=STRING

type=string

access=inout

#####

```
# service : .DELREC #
#####

service=.DELREC
export=N
export=N
inbuf=FML32
outbuf=FML32
version=1
param=REPNAME
type=string
access=in
param=STATLIN
type=string
access=out

#####

# service : .FLUSHCACHE #
#####

service=.FLUSHCACHE
export=N
inbuf=FML32
outbuf=FML32
version=1
param=REPNAME
count=200
type=string
access=in
```

```
#####  
# service : .NUMRECS #  
#####  
  
service=.NUMRECS  
export=N  
inbuf=FML32  
outbuf=FML32  
version=1  
param=NRECS  
type=integer  
access=out  
param=STATLIN  
type=string  
access=out  
  
#####  
# service : INQUIRY #  
#####  
  
service=INQUIRY  
export=Y  
inbuf=FML  
outbuf=FML  
version=1  
param=ACCOUNT_ID  
type=integer  
access=in
```

param=SBALANCE

type=string

access=out

param=STATLIN

type=string

access=out

param=FORMNAM

type=string

access=out

#####

# service : DEPOSIT #

#####

service=DEPOSIT

export=Y

inbuf=FML

outbuf=FML

version=1

param=ACCOUNT\_ID

type=integer

access=in

param=SAMOUNT

type=string

access=in

param=SBALANCE

type=string

access=out

param=BALANCE

type=float

access=out

param=STATLIN

type=string

access=out

param=FORMNAM

type=string

access=out

#####

# service : .GETSVC #

#####

service=.GETSVC

export=N

inbuf=FML32

outbuf=FML32

version=1

param=REPNAME

type=string

access=in

param=REPVALUE

type=string

access=out

param=STATLIN

type=string

access=out

```
#####  
# service : .GETREC #  
#####  
  
service=.GETREC  
export=N  
inbuf=FML32  
outbuf=FML32  
version=1  
param=REPNAME  
type=string  
access=in  
param=REPVALUE  
type=string  
access=out  
param=STATLIN  
type=string  
access=out
```

---

**Listing 1-4 Example for Inserting jolt.metarepos to the Original Metadata Repository File**

---

```
tmloadrepos -i meta.data jolt.metarepos
```

---

# Oracle Jolt Exceptions

This appendix describes all the Oracle Jolt exceptions that you may encounter. Keep in mind that the Jolt Class Library returns both Oracle Jolt and Oracle Tuxedo exceptions.

For details about Oracle Tuxedo exceptions, refer to the appropriate document in the following list:

- [\*Oracle Tuxedo Command Reference\*](#)
- [\*Oracle Tuxedo ATMI C Function Reference\*](#)
- [\*Oracle Tuxedo ATMI COBOL Function Reference\*](#)
- [\*Oracle Tuxedo ATMI FML Function Reference\*](#)
- [\*File Formats, Data Descriptions, MIBs, and System Processes Reference\*](#)

The Jolt Class Library exceptions are listed for each class, constructor, and method listed in the *Oracle Jolt API Reference*.

The following table lists the Oracle Jolt and Oracle Tuxedo exceptions that you may encounter while running Oracle Jolt. Each exception includes a possible cause (or causes) and a recommended action (wherever possible) to help resolve the situation

<b>1. TPEABORT</b>	<b>A transaction could not commit.</b>	
	<b>Cause</b>	This exception occurs because a transaction could not commit on the server side. This exception may also occur if the JSH performs a message resend for a commit that has timed out due to a previous blocking condition. In Oracle Tuxedo, you can get this exception if <code>tpcommit()</code> is called with outstanding replies or open conversation connections.
	<b>Action</b>	Check transaction failures on the server side. Oracle Jolt clients should resend the request after the transaction problem has been fixed on the server side.
<b>2. TPEBADDESC</b>	<b>This exception should not occur in Oracle Jolt.</b>	
	<b>Cause</b>	In Oracle Tuxedo, this exception usually occurs when an invalid caller descriptor is given to <code>tpgetreply()</code> or <code>tpsend()</code> .
	<b>Action</b>	None.
<b>3. TPEBLOCK</b>	<b>A blocking condition has occurred and the TPNOBLOCK flag is specified in Oracle Tuxedo.</b>	
	<b>Cause</b>	This exception occurs because the server is backed up.
	<b>Action</b>	You may need to re-examine and re-architect the application to handle extreme load cases.
<b>4. TPEINVAL</b>	<b>Invalid arguments were given by the application.</b>	

	<b>Cause</b>	This exception occurs if a new JoltSession class is processed before performing the security protocol. In Jolt's URL handler routine, this exception occurs when a invalid challenge response is received by the <code>openConnection()</code> method. The <code>TPEINVAL</code> exception can also occur if you specified a hexadecimal address for the <code>JSL -H</code> option without a leading "0x", or if you entered a wrong address in <code>UBBCONFIG</code> file. In addition, the <code>GETREC()</code> , <code>DELREC()</code> and <code>GETSVC()</code> services in <code>JREPSVR</code> can return <code>TPEINVAL</code> if the <code>REPNAME</code> is missing. Also, the <code>ADDREC()</code> service in <code>JREPSVR</code> can return <code>TPEINVAL</code> if the <code>REPVAL</code> is not specified.
	<b>Action</b>	This type of exception should have been handled during the application development cycle. You should not receive this exception in a production environment.
<b>5. TPELIMIT</b>	<b>The maximum number of outstanding requests or subscriptions has been reached.</b>	
	<b>Cause</b>	The maximum number of outstanding requests has been reached. This exception could also mean that the Oracle Tuxedo System Event Broker's maximum number of subscriptions (50 internally defined for now) has been reached.
	<b>Action</b>	You may need to re-examine and re-architect the application to handle load extreme cases.
<b>6. TPEOENT</b>	<b>The requested service is not available.</b>	
	<b>Cause</b>	Usually, the requested service is not booted or advertised on the Oracle Tuxedo server side. It is also possible that the requested service is not defined in the Oracle Tuxedo Service Metadata Repository. This exception could also indicate that you could not access the Oracle Tuxedo System Event Broker.
	<b>Action</b>	You need to check the server side to see if the service is booted or advertised. Otherwise, check to see if the requested service is defined in the Oracle Tuxedo Service Metadata Repository. After the service is available on the server side, Jolt clients should resend the request.
<b>7. TPEOS</b>	<b>An operating system exception has occurred.</b>	

<b>Cause</b>	The exact nature of the problem is described in the ULOG file. Typically, you can get this exception due to the memory allocation failures, wrong network address, or failure to attach to the Bulletin Board for the JSL.
<b>Action</b>	Try fixing the problem as described in the ULOG file. Jolt clients might need to reconnect or resend the request after the problem has been fixed.

#### 8. TPEPERM

#### There is a permission problem when attempting to join a session.

<b>Cause</b>	In the JoltSession class, this exception occurs because the Jolt client does not have the permission to join the application. Permission may be denied based on an invalid application password, failure to pass application specific authentication, or the use of restricted client names. In the Jolt URL handler routing, this exception occurs when a bad challenge response is received on the <code>openConnection()</code> method. If the Oracle Tuxedo Service Metadata Repository is set to read-only, the <code>ADDREC()</code> and <code>DELREC()</code> services, or the <code>GARBAGECOLLECT()</code> service in <code>JREPSVR</code> , also return the TPEPERM exception.
<b>Action</b>	This type of exception should have been handled during the application development cycle. You should not receive this exception in a production environment.

#### 9. TPEPROTO

#### A function was called in an improper context.

<b>Cause</b>	For this exception, an improper context could include a <code>rollback()</code> or <code>commit()</code> method called by a participant, an unsubscribe event that is called while “unsubscribe all” is in progress, or when the caller is not a client.
<b>Action</b>	This type of exception should have been handled during the application development cycle. You should not receive this exception in a production environment.

#### 10. TPESVCERR

#### A service routine has encountered an exception during `tpreturn()` or `tpforward()` in Oracle Tuxedo.

	<b>Cause</b>	The service routine is returning application-level failures, which may include any of the following: an application calls <code>tpreturn()</code> or <code>tpforward()</code> with invalid flags, the caller descriptor is no longer valid, or there are invalid return values.
	<b>Action</b>	This type of exception should have been handled during the application development cycle. You should not receive this exception in a production environment.
<b>11. TPESVCFAIL</b>	<b>The service routine sending the caller's reply called <code>tpreturn()</code> with <code>TPFAIL</code>.</b>	
	<b>Cause</b>	The service routine is returning application-level failures.
	<b>Action</b>	This type of exception should have been handled during the application development cycle. You should not receive this exception in a production environment.
<b>12. TPESYSTEM</b>	<b>An Oracle Tuxedo system exception has occurred.</b>	
	<b>Cause</b>	The exact nature of the exception is written to the ULOG file. For example, when performing the Diffie-Hellman encryption, this exception occurs if the JSH is unable to send negotiation parameters. The JSL fails to send the reply challenge call to the Jolt client. The Jolt client sends an incorrect timestamp value, an incorrect number of encrypted bits value, an incorrect ticket value, or timestamp mismatches in reconnect protocol. The JSL fails to initialize network protocol information, or could not establish a listening address on a network. The JSH receives a network message with an unknown context or receives a message with a different connection.
	<b>Action</b>	In most cases, you need to find out the exact nature of the exception from the ULOG file on the server side. In case of hardware or network failures, you can try to reconnect if a hardware or network failover is available.
<b>13. TPETIME</b>	<b>A transaction timeout has occurred.</b>	
	<b>Cause</b>	There is a transaction timeout on the server side.
	<b>Action</b>	This type of exception should be addressed on the application server side. Jolt clients should resend the request after the server side problem has been resolved.
<b>14. TPETRAN</b>	<b>The requested service belongs to a server that does not support transactions and <code>TPNOTRAN</code> is not set.</b>	

	<b>Cause</b>	A transaction is not supported for the requested service.
	<b>Action</b>	This type of exception should be addressed on the application server side. Jolt clients should resend the request after the server side problem has been resolved.
<b>15. TPGOTSIG</b>	<b>An unexpected signal was received.</b>	
	<b>Cause</b>	A signal was received and the TPSIGSTRT flag was not specified.
	<b>Action</b>	None.
<b>16. TPERMERR</b>	<b>A resource manager failed to open or close correctly on the server side.</b>	
	<b>Cause</b>	The resource manager might not be available; or all the resource might not be released or committed before close.
	<b>Action</b>	Check the ULOG file for reasons why the resource manager failed to open or close on the server side.
<b>17. TPEITYPE</b>	<b>For the JoltRemoteService class, the requested Oracle Tuxedo service does not recognize the type and subtype of the input data.</b>	
	<b>Cause</b>	The type and subtype of input data is not defined in the Oracle Tuxedo Service Metadata Repository.
	<b>Action</b>	The type and subtype of input data should be defined in the Oracle Tuxedo Service Metadata Repository. This type of exception should have been handled during the application development cycle. You should not receive this exception in a production environment.
<b>18. TPEOTYPE</b>	<b>For the JoltRemoteService class, the Oracle Tuxedo caller does not recognize the type and the subtype of the reply data.</b>	
	<b>Cause</b>	The type and subtype of output data is not defined in the Oracle Tuxedo Service Metadata Repository.
	<b>Action</b>	The type and subtype of output data should be defined in the Oracle Tuxedo Service Metadata Repository. This type of exception should have been handled during the application development cycle. You should not receive this exception in a production environment.
<b>19. TPERELEASE</b>	<b>This exception should not occur in Oracle Jolt.</b>	

	<b>Cause</b>	Usually, this exception occurs when an unsolicited notification message is sent from a server with the <code>TPACK</code> flag set, and the target is a Jolt client from an older release of Oracle Jolt that does not support the acknowledgment protocol.
	<b>Action</b>	Verify that the correct version of Oracle Jolt is installed on your machine. This type of exception should have been handled during the application development cycle. You should not receive this exception in a production environment.
<b>20 . TPEHAZARD</b>	<b>Due to some failure, the work done on behalf of the transaction may have been heuristically completed.</b>	
	<b>Cause</b>	Check the ULOG file on the server side for details.
	<b>Action</b>	None.
<b>21 . TPEHEURISTIC</b>	<b>Due to a heuristic decision, the work done on behalf of the transaction was partially committed and partially aborted.</b>	
	<b>Cause</b>	Check the ULOG file on the server side for details.
	<b>Action</b>	None.
<b>22 . TPEEVENT</b>	<b>This exception should not occur in Oracle Jolt.</b>	
	<b>Cause</b>	Usually, this exception means that an event has occurred when sending or receiving a message in a conversational connection in Oracle Tuxedo. However, conversational server connections are not available in Oracle Jolt.
	<b>Action</b>	None.
<b>23 . TPEMATCH</b>	<b>The JoltUserEvent class has implemented a subscription to an asynchronous notification event, but the subscription has failed because it matches an existing subscription.</b>	
	<b>Cause</b>	The subscription failed because it matched one already listed with the Oracle Tuxedo System Event Broker.
	<b>Action</b>	None.
<b>24 . TPEDIAGNOSTIC</b>	<b>This exception should not occur in Oracle Jolt.</b>	

	<b>Cause</b>	Usually, this exception occurs when enqueueing or dequeuing a message from the specified queue fails in Oracle Tuxedo. However, enqueueing and dequeuing of messages is not available in Oracle Jolt.
	<b>Action</b>	None.
<b>25. TPEMIB</b>	<b>This exception should not occur in Oracle Jolt.</b>	
	<b>Cause</b>	Usually, this exception occurs when an administrative request via <code>tpadmcall()</code> has failed in Oracle Tuxedo. However, TMIB calls are not available in Oracle Jolt.
	<b>Action</b>	None.
<b>26. TPEJOLT</b>	<b>This exception indicates there is a problem in Oracle Jolt.</b>	
	<b>Cause</b>	<p>The TPEJOLT exception could occur for any of the following reasons:</p> <ul style="list-style-type: none"> <li>• JoltSession class—the <code>send()</code>, <code>recv()</code> or <code>cancel()</code> methods throw TPEJOLT if the session object or message ID is invalid.</li> <li>• JoltSession class—throws TPEJOLT when TPINIT data conversion fails.</li> <li>• <code>bea.jolt.pool.connection</code> class—throws TPEJOLT when a run-time exception occurs.</li> <li>• JoltRemoteService—the <code>call()</code> method throws TPEJOLT when the buffer conversion between Oracle Jolt and Oracle Tuxedo fails, the requested service is not defined in the Oracle Tuxedo Service Metadata Repository, the requested service does not the right version, or the reply data conversion fails.</li> <li>• JoltUserEvent class—throws TPEJOLT when event name conversion fails, an invalid message ID is encountered, or unsolicited message data conversion fails.</li> </ul>
	<b>Action</b>	This type of exception should have been handled during the application development cycle. You should not receive this exception in a production environment.