BEA Tuxedo®

Using BEA Jolt

Version 10.0
Document Released: JSeptember 28, 2007

Contents

3. Configuring the BEA Jolt System

4. Using the Jolt Repository Editor

5. Using the Jolt Class Library

6. Using JoltBeans

7. Using Servlet Connectivity for BEA Tuxedo

A. BEA Jolt Exceptions

CHAPTER 1

Introducing BEA Jolt

BEA Jolt is a Java-based interface to the BEA Tuxedo system that extends the functionality of existing BEA Tuxedo applications to include Intranet- and Internet-wide availability. Using Jolt, you can now easily transform any BEA Tuxedo application so that its services are available to customers using an ordinary browser on the Internet. Jolt interfaces with existing and new BEA 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 BEA Tuxedo services, such as application messaging, component management, and distributed transaction processing.

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

This topic includes the following sections:

BEA Jolt Components

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

- **Jolt 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 BEA Tuxedo-based applications running on one or more BEA 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 BEA Tuxedo services. The Jolt class library includes functionality to set, retrieve, manage, and invoke communication attributes, notifications, network connections, transactions, and services.
- JoltBeans—BEA JoltBeans provides a JavaBeans-compliant interface to BEA Jolt. JoltBeans are Beans components that you can use in JavaBeans-enabled integrated development environments (IDEs) to construct BEA Jolt clients. Jolt Beans consists of two sets of Java Beans: JoltBeans toolkit (a JavaBeans-compliant interface to BEA 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 Repository**—a central repository contains definitions of BEA Tuxedo services. These repository definitions are used by Jolt at run time to access BEA Tuxedo services. You can export services to a Jolt client application or unexport services by hiding the definitions from the Jolt client. Using the Repository Editor, you can test new and existing BEA Tuxedo services independently of the client applications.
- **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 BEA 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).

Key Features

With BEA Jolt, you can leverage existing BEA 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.

BEA Jolt includes the following features:

- **Java-based API for simplified development—**with its Java-based API, BEA 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 BEA Tuxedo application and transaction services without having to understand detailed transactional semantics or rewrite existing BEA 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 BEA Tuxedo data types and buffers, and from BEA 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 BEA Tuxedo services through Jolt Repository—**the BEA Jolt Repository facilitates Java application development by managing and presenting BEA Tuxedo service definitions that you can use in your Java client. A Jolt Repository bulk loading utility lets you quickly integrate your existing BEA Tuxedo services into the Jolt development environment. Jolt and BEA Tuxedo simplify network and application scalability, while encouraging the reuse of application components.
- **GUI-Based maintenance and distribution of BEA Tuxedo services—**the Jolt Repository Editor lets you manage BEA Tuxedo service definitions such as service names, inputs and outputs. The Jolt Repository Editor provides support for different input and output names for services defined in the Jolt Repository.
- **Encryption for secure transaction processing—**BEA 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 BEA Jolt Internet Relay component to separate their Web server and BEA 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 BEA 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 BEA Tuxedo services and BEA Tuxedo clients. Jolt Event Subscription lets you subscribe to two types of BEA Tuxedo application events:
  - Unsolicited Event Notifications—a Jolt client can receive these notifications when a BEA Tuxedo client or service subscribes to unsolicited events and a BEA Tuxedo client issues a broadcast or a directly targeted message.
  - Brokered Event Notifications—the Jolt client receives these notifications through the BEA Tuxedo Event Broker. The Jolt client receives these notifications only when it subscribes to an event and any BEA Tuxedo client or server posts an event.

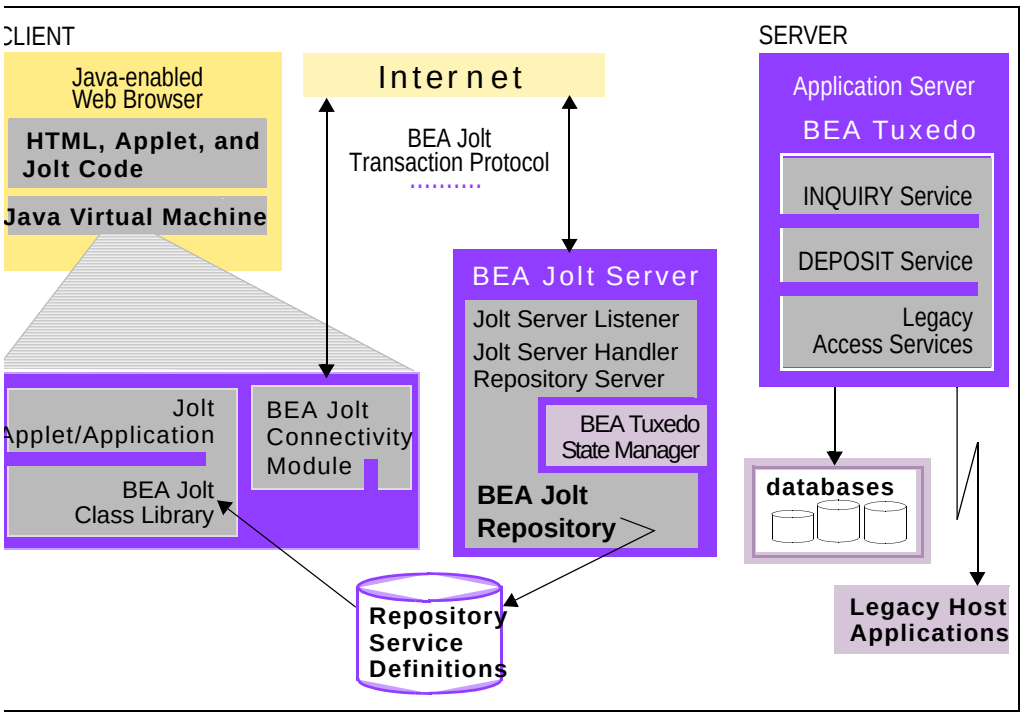
How BEA Jolt Works

BEA Jolt connects Java clients to applications that are built using the BEA Tuxedo system. The BEA 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 BEA Jolt architecture, as well as related BEA Tuxedo components and their interactions, is illustrated in the figure "BEA Jolt Architecture" on page 1-6.

Using this figure 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 BEA Tuxedo service on behalf of the client. The BEA Jolt Server accepts requests from the Jolt clients and maps those requests into BEA Tuxedo service requests.

Figure 1-1 BEA Jolt Architecture



Jolt Servers

The following Jolt Server components act in concert to pass Jolt client transaction processing requests to the BEA 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)**

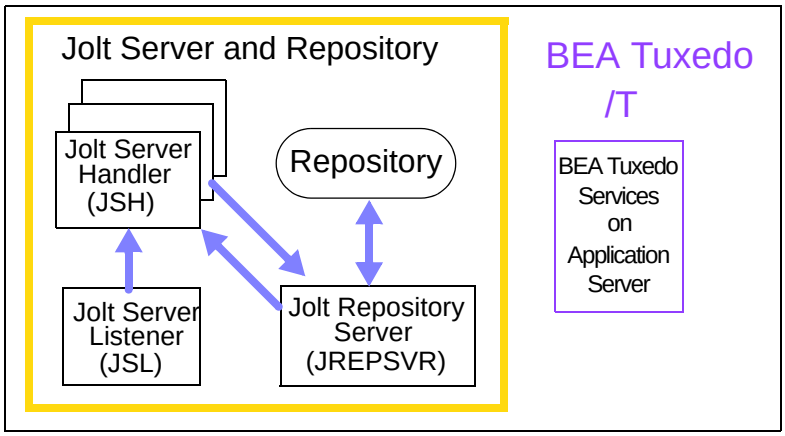
  The JSH manages network connectivity, executes service requests on behalf of the client and translates BEA Tuxedo buffer data into the Jolt buffer, as well as Jolt buffer data into the Tuxedo buffer.

- **Jolt Repository Server (JREPSVR)**

The JREPSVR retrieves Jolt service definitions from the Jolt Repository and returns the service definitions to the JSH. The JREPSVR also updates or adds Jolt service definitions.

The following figure illustrates the Jolt Server and Jolt Repository components.

Figure 1-2 Jolt Server and Repository Components



Jolt Class Library

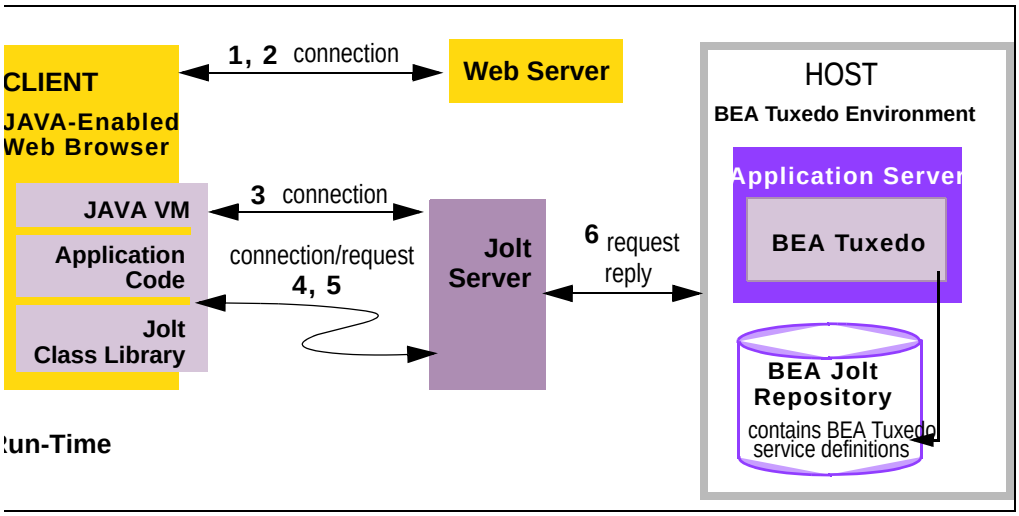
The BEA Jolt Class Library is a set of classes that you can use in your Java application or applet to make service requests to the BEA Tuxedo system from a Java-enabled client. You access BEA Tuxedo transaction services by using Jolt class objects.

When developing a Jolt client application, you only need to know about the classes that Jolt provides and the BEA Tuxedo services that are exported by the Jolt 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 Jolt Repository.

The following figure shows the flow of activity from a Jolt client to and from the BEA 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 BEA Tuxedo Services



The following table briefly describes the flow of activity involved in using the Jolt Class Library to access BEA Tuxedo services, as shown in the previous figure "Using the Jolt Class Library to Access BEA 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.
...	**3**	The first Java applet task is to open a separate connection to the Jolt Server.
Request	**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 a BEA Tuxedo request and forwards the request to the BEA Tuxedo environment.
Reply	**6**	The BEA Tuxedo system processes the request and returns the information to the Jolt Server, which translates it back to the Java applet.

JoltBeans

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

BEA JoltBeans provide a JavaBeans-compliant interface to BEA Jolt that enables you to develop a fully functional BEA 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 BEA Jolt Repository, which provides easy access to available BEA Tuxedo functions.

Jolt Server and Jolt Client Communication

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

1. BEA 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 BEA Tuxedo system, it is executed in exactly the same manner as requests issued by any other BEA Tuxedo client.
5. The results are then returned to the BEA 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.

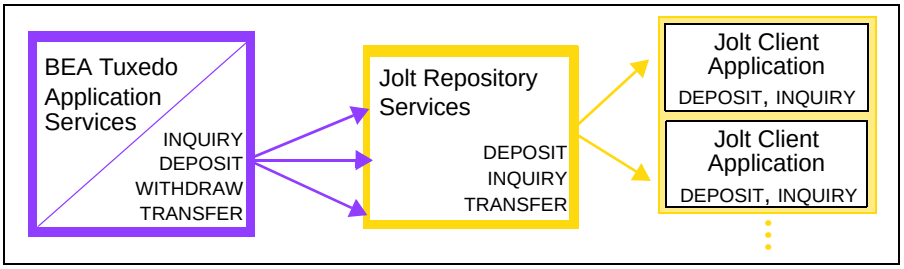
Jolt Repository

The Jolt Repository is a database where BEA Tuxedo services are defined, such as name, number, type, parameter size, and permissions. The repository functions as a central database of definitions for BEA Tuxedo services and permits new and existing BEA Tuxedo services to be made available to Jolt client applications. A BEA 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 Jolt Repository. Within the Jolt Repository, the developer or system administrator uses the Jolt Repository Editor to export these services to the Jolt client application.

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

The following figure illustrates how the Jolt Repository brokers BEA Tuxedo services to multiple Jolt client applications. (Four BEA 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 BEA Tuxedo Services Through Jolt



Jolt Repository Editor

The Jolt Repository Editor is a Java-based GUI administration tool that gives the application administrator access to individual BEA Tuxedo services. You use the Editor to define, test, and export services to Jolt clients.

Note: The Jolt Repository Editor only controls services for Jolt client applications. You cannot use it to make changes to the BEA Tuxedo application.

The Jolt Repository Editor lets you extend and distribute BEA Tuxedo services to Jolt clients without having to modify many lines of code. You can modify parameters for BEA Tuxedo services, logically group BEA Tuxedo services into packages, and remove services from created packages. You can also make the services available to browser-based Jolt applets or Jolt applications by exporting the services.

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 BEA Tuxedo system to run on the same machine as the Web server.

The JRAD is a BEA Tuxedo system server, but does not include any BEA Tuxedo services. It requires command-line arguments to allow it to work with the JSH and the BEA 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 BEA 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 a BEA Tuxedo system application.

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

2. Install the Jolt system.

   Refer to *Installing the BEA 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."

4. Configure and define services by using the Jolt Repository Editor.

   For information about configuring the Jolt Repository Editor and making BEA Tuxedo services available to Jolt, see Chapter 4, "Using the Jolt Repository Editor"

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

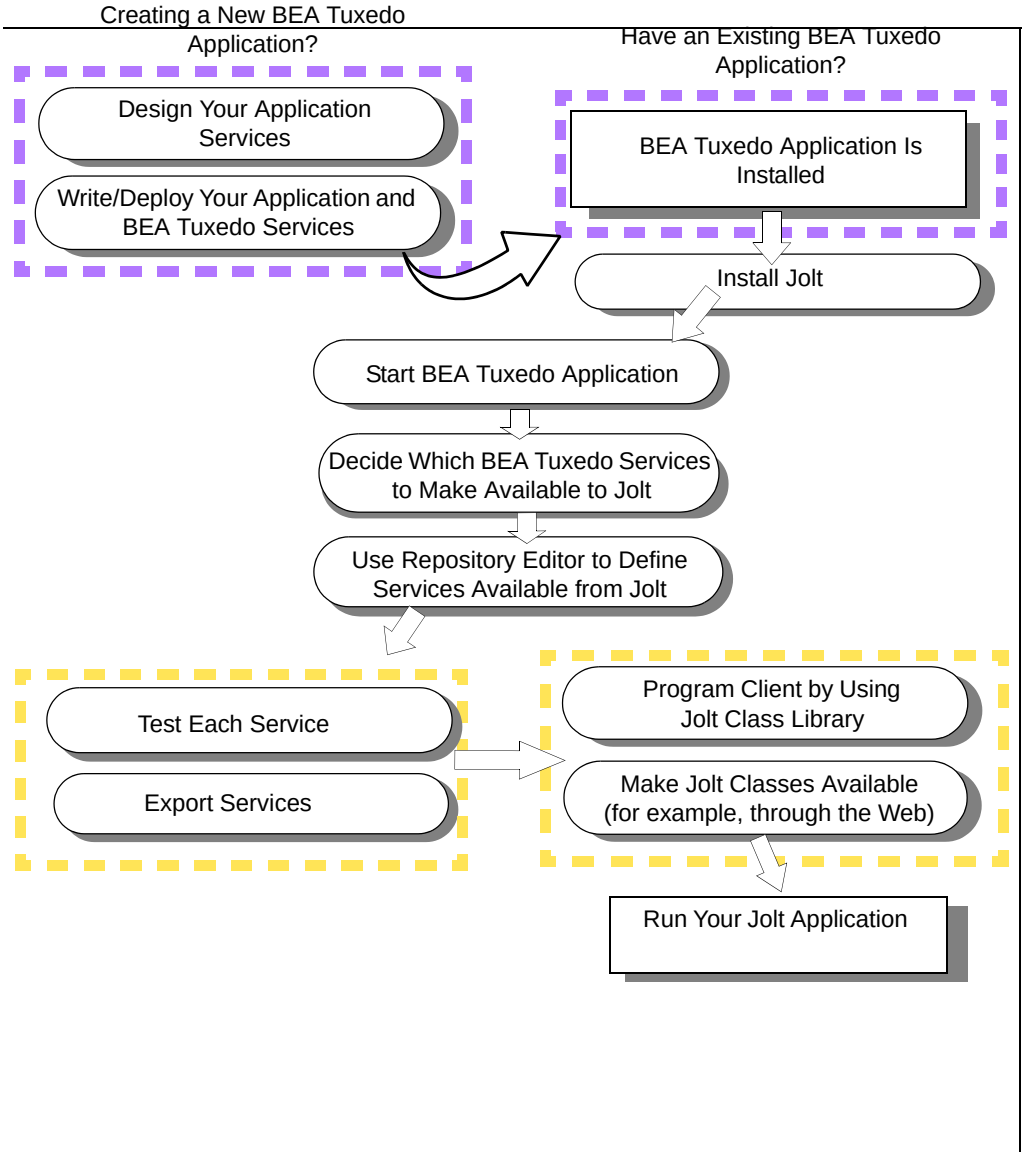
   The following documentation shows you how to program your client application using the Jolt Class Library:

   – Using the Jolt Class Library

   – BEA Jolt API Reference

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

Figure 1-5 Creating a Jolt Application



CHAPTER 2

Bulk Loading BEA Tuxedo Services

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

This topic includes the following sections:

Using the Bulk Loader

The `jbld` program is a Java application. Before running the `jbld` 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 the following table to correctly specify the command-line options.

Command-line Options

Table 2-1 Bulk Loader Command-line Options

Option	Description
`-u usrname`	Specifies the username (default is your account name). (Mandatory if required by security.)
`-r usrrole`	Specifies the user role (default is `admin`). (Mandatory if required by security.)
`-n`	Validates input file against the current repository; no updates are made to the repository. (Optional)
`-p package`	Repository package name (default is BULKPKG).
`//host:port`	Specifies the JRLY or JSL address (host name and IP port number). (Mandatory)
`filename`	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 Jolt Repository using the package name "BULKPKG" by default. The `-p` command overrides the default and you can give the package any name you choose. If another load is performed from a bulk loader file with the same

`-p` option, all the services in the original package are deleted and a new package is created with the services from the new bulk loader file.

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. See "Using the Jolt Repository Editor" on page 4-1 for additional information.

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 the following table.

Table 2-2 Guidelines for Using Keywords

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

Table 2-2 Guidelines for Using Keywords (Continued)

Guideline	Example

The input file can contain multiple service definitions.	`service=INQUIRY`
	`<service keywords and values>`
	`service=DEPOSIT`
	`<service keywords and values>`
	`service=WITHDRAWAL`
	`<service keywords and values>`
	`service=TRANSFER`
	`<service keywords and values>`

Each service definition consists of multiple keywords and values.	`service=DEPOSIT`
	`export=true`
	`inbuf=VIEW32`
	`outbuf=VIEW32`
	`inview=INVIEW`
	`outview=OUTVIEW`

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 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.

The following table 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
`service`	Any BEA Tuxedo service name
`export`	True or false (default is false)

`inbuf/outbuf`	Select one of these buffer types:
	`FML`
	`FML32`
	`VIEW`
	`VIEW32`
	`STRING`
	`CARRAY`
	`XML`
	`X_OCTET`
	`X_COMMON`
	`X_C_TYPE`

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

Keyword	Value
inview	Any view name for input parameters (This keyword is optional *only* if one of the following buffer types is used: VIEW, VIEW32, X_COMMON, X_C_TYPE.)
outview	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.

The following table contains the guidelines for use of the parameter-level keywords and acceptable values for each.

Table 2-4 Parameter-Level Keywords and Values

Keyword	Values
param	Any parameter name
type	byte short integer float double string carray xml
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.

Troubleshooting

If you encounter problems using the Bulk Loader utility, refer to the following table. For a complete list of Bulk Loader utility error messages and solutions, see “System Messages.”

Table 2-5 Bulk Loader Troubleshooting Table

If . . .	Then . . .
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

The following listing 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
```

CHAPTER 3

Configuring the BEA Jolt System

This chapter describes how to configure BEA 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 BEA Jolt.

This topic includes the following sections:

Quick Configuration

If you are already familiar with BEA Jolt and BEA Tuxedo, "Quick Configuration" provides efficient guidelines for the configuration procedure. If you have not used Jolt, refer to "Jolt Background Information" on page 3-11 before you begin any configuration procedures.

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

- Editing the UBBCONFIG File
- Configuring the Jolt Repository
- Initializing Services That Use BEA Tuxedo and the Repository Editor
- Logging On to the Repository Editor
- Exiting the Repository Editor
- Configuring the BEA 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 3-37 for additional information.

   MAX # of JSHs

   MIN # of JSHs

Configuring the Jolt Repository

The following sections assist you in configuring the Jolt Repository.

In the Groups Section

1. Specify the same identifiers given as the value of the LMID parameter in the MACHINES section.
2. Specify the value of the GRPNO, between 1 and 30,000.

In the Servers Section

The BEA Jolt Repository Server (JREPSVR) contains services for accessing and editing the Repository. Multiple JREPSVR instances share repository information through a shared file. Include JREPSVR in the SERVERS section of the UBBCONFIG file.

1. Indicate a new server identification with the SRVID parameter.
2. Specify the -W flag for one (and only one) JREPSVR to ensure that you can edit the repository. (Without this flag, the repository is read-only.)
3. Type the -P flag to specify the path of the repository file. (An error message is displayed in the BEA Tuxedo ULOG file if the argument for the -P flag is not entered.)
4. Add the file pathname of the Repository file (for example, /app/jrepository).
5. Boot the BEA Tuxedo system by using the tmloadcf and tmboot commands.

Initializing Services That Use BEA Tuxedo and the Repository Editor

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

1. Build the BEA Tuxedo server that contains the service.
2. Access the BEA Jolt Repository Editor.

Getting Started with the Repository Editor

Before you start the Repository Editor, make certain that you have installed all of the necessary BEA Jolt software.

Note: You cannot use the Repository Editor until `JREPSVR` and `JSL` are running.

To use the Repository Editor, you must:

1. Start the Repository Editor.

   You can start the Repository Editor from either the JavaSoft `appletviewer` or from your Web browser. Both of these methods are detailed in the following sections.

2. Log on to the Repository Editor.

Starting the Repository Editor Using the Java Applet Viewer

1. Set the `CLASSPATH` to include the Jolt class directory or the directory where the `*.jar` files reside.
2. If loading the applet from a local disk, type the following at the URL location:

   `appletviewer` *`full-pathname`*`/RE.html`

   If loading the applet from the Web server, type the following at the URL location:

   `http://`*`www.server/URL path`*`/RE.html`

3. Press **Enter**.

   The window is displayed as shown in the figure "BEA Jolt Repository Editor Logon Window" on page 3-6.

Starting the Repository Editor Using Your Web Browser

Use one of the following procedures to start the Repository Editor from your Web browser.

To start the Repository Editor from a local file

1. Set the `CLASSPATH` to include the Jolt class directory.
2. Type the following:

`file:`*`full-pathname`*`/RE.html`

3. Press **Enter**.

   The window is displayed as shown in the figure "BEA Jolt Repository Editor Logon Window" on page 3-6.

To start from a Web server

1. Ensure that the `CLASSPATH` does not include the Jolt class directory.
2. Remove the Jolt cases from `CLASSPATH`.
3. Type the following:

   `http://`*`www.server/URL path`*`/RE.html`

   Note: If `jolt.jar` and `admin.jar` are in the same directory as `RE.html`, the Web server provides the classes. If they are not in the same directory as `RE.html`, modify the applet code base.

4. Press **Enter**.

   The Repository Editor Logon window is displayed as shown in the figure "BEA Jolt Repository Editor Logon Window" on page 3-6.

Logging On to the Repository Editor

After starting the Jolt Repository Editor, follow these directions to log on:

Note: The "BEA Jolt Repository Editor Logon Window" on page 3-6 must be displayed before you log on. Refer to this figure as you perform the following procedure.

1. In the logon window, type the name of the Server machine designated as the "access point" to the BEA Tuxedo application and press **Tab**.
2. Type the Port Number and press **Enter**.

   The system validates the server and port information.

   Note: Unless you are logging on through Jolt Relay, the same port number is used to configure the Jolt Listener. Refer to your `UBBCONFIG` file for additional information.

3. Type the BEA Tuxedo Application Password and press **Enter**.

   Depending upon the authentication level, complete steps 5 and 6 as required.

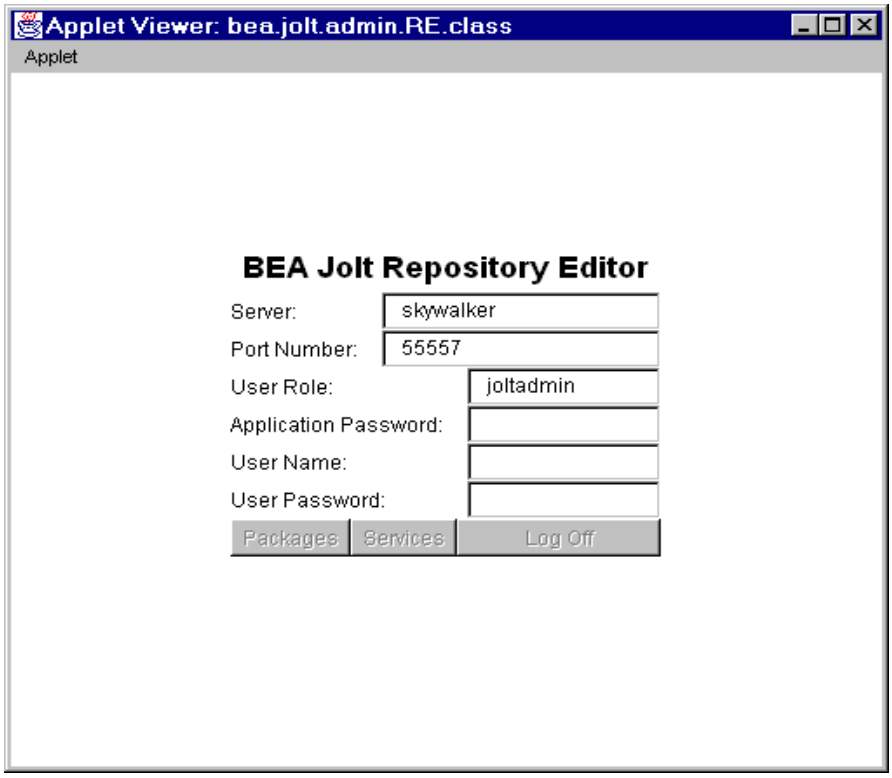
4. Type the BEA Tuxedo User Name and press **Tab**.

5. Type the BEA Tuxedo User Password and press **Enter**.

   The **Packages** and **Services** command buttons are enabled.

   Note: The BEA Jolt Repository Editor uses the hardcoded `joltadmin` for the User Role value.

Figure 3-1 BEA Jolt Repository Editor Logon Window



The following table, "Repository Editor Logon Window Description," contains details about each of the fields and buttons.

Repository Editor Logon Window Description

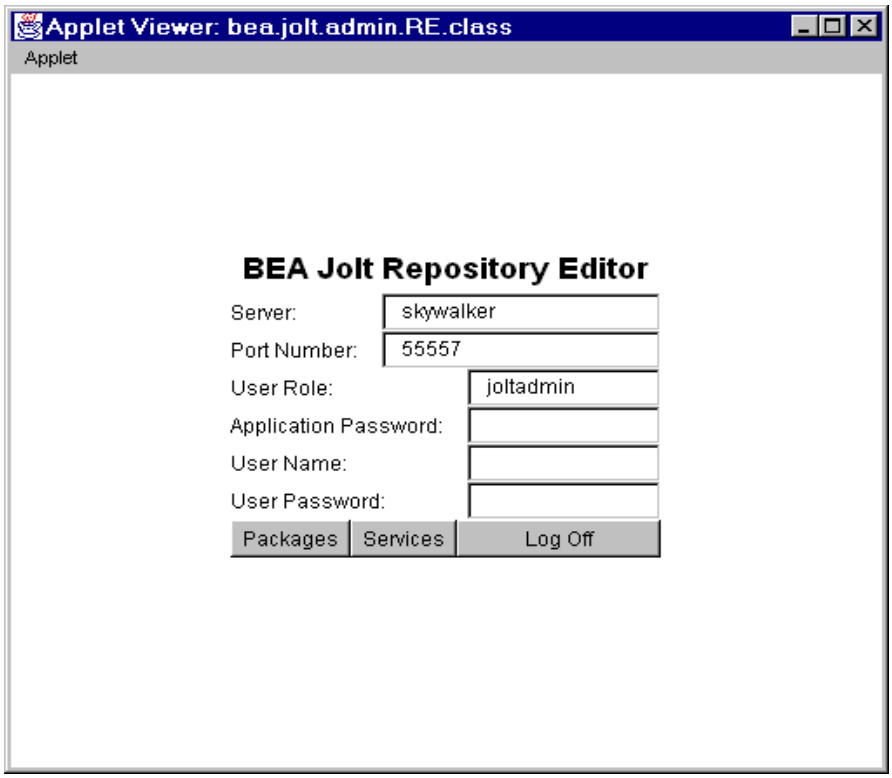
Table 3-1 Repository Editor Logon Window Description

Option	Description
Server	The server name.
Port Number	The port number in decimal value. **Note:** After the Server Name and Port Number are entered, the User Name and Password fields are activated. Activation is based on the authentication level of the BEA Tuxedo application.
User Role	BEA Tuxedo user role. Required only if BEA Tuxedo authentication level is USER_AUTH or higher.
Application Password	BEA Tuxedo administrative password text entry.
User Name	BEA Tuxedo user identification text entry. The first character must be an alpha character.
User Password	BEA Tuxedo password text entry.
Packages	Accesses the Packages window. (Enabled after the logon.)
Services	Accesses the Services window. (Enabled after the logon.)
Log Off	Terminates the connection with the server.

Exiting the Repository Editor

Exit the Repository Editor when you finish adding, editing, testing, or deleting packages, services, and parameters. Prior to exit, the window is displayed as shown in the figure "BEA Jolt Repository Editor Logon Window Prior to Exit" on page 3-8.

Figure 3-2 BEA Jolt Repository Editor Logon Window Prior to Exit



Note that only the **Packages**, **Services**, and **Log Off** command buttons are enabled. All of the text entry fields are disabled.

Follow the steps below to exit the Repository Editor.

1. Click **Back** in a previous window to return to the Repository Editor Logon window.
2. Click **Log Off** to terminate the connection with the server.

   The Repository Editor Logon window shows disabled fields.
3. Click **Close** from your browser menu to close the window.

Configuring the BEA Tuxedo TMUSREVT Server for Event Subscription

Jolt Event Subscription receives event notifications from either BEA Tuxedo services or other BEA Tuxedo clients. Configure the BEA 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 the following table.

Table 3-2 Host Name and Port Number Formats

Host Name/Port Number	Description
//Hostname:Port	Hostname is a string; Port is a decimal number.
IP:Port	IP is a dotted notation IP address; Port is a decimal number.

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 BEA 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 BEA Jolt system. The JSL runs as a BEA Tuxedo server.

Jolt Server Handler (JSH)—the JSH is a program that runs on a BEA 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 BEA 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 3-13 for additional information.)

System Administrator Responsibilities—the system administrator's responsibilities for the server components of BEA 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 a BEA Tuxedo Application at Run Time* or the *BEA Tuxedo Command Reference* for information about `tmloadcf` and `tmboot`.

Shutting Down the JSL

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

```
tmshutdown -y
```

During shutdown:

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

Restarting the JSL

BEA Tuxedo monitors the JSL and restarts it in the event of a failure. When BEA 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 a BEA Tuxedo server responsible for distributing connection requests from Jolt to the Jolt Server Handler (JSH). BEA 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 BEA 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 3-33.

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 the following table.

Table 3-3 JSL Command-line Options

Option	Description
`[-a]`	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. If the JSH, gets a message with the caller's identity, it calls `impersonate_user()` 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.
`[-A]`	Specifies that certificate-based authentication should be required when accepting an SSL connection from a remote application. **Note:** The JSL `-A` option is equivalent to the ISL(5) and WSL(5) `-a` option. For more information see, *Section 5 - File Formats, Data Descriptions, MIBs, and System Processes Reference*.

Table 3-3 JSL Command-line Options

Option	Description
[-c *compression_threshold*]	Enables application data sent between a Jolt client and a Jolt server (JSH) to be compressed during transmission over the network. *compression_threshold* 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. The default is no compression; that is, if no compression threshold is specified, BEA Jolt does not compress messages on client or server.
[-d *device_name*]	The device for platforms using the Transport Layer Interface. There is no default. Required. (Optional for sockets)
[-H *external netaddr*]	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 `0x0002MMMMdddddddd` and the JSH network address is `0x00021111ffffffff`, the known (or external) network address is `0x00021111dddddddd`. 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. The external IP address mask must be specified in the following form: `-H //external ip address:MMMM` (Optional for JSL in BEA Tuxedo 6.4 and 6.5)
[-I *init-timeout*]	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)
[-j *connection_mode*]	The following connection modes from clients are allowed: RETAINED—the network connection is retained for the full duration of a session. RECONNECT—the client establishes and brings down a connection when an idle timeout is reached, reconnecting for multiple requests within a session. ANY—the server allows a client to request either a RETAINED or RECONNECT type of connection for a session. 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)

Table 3-3 JSL Command-line Options

Option	Description
[-m *minh*]	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)
[-M *maxh*]	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 MAXWSCLIENTS divided by the -x multiplexing factor (MPX), with the result rounded up. If specified, the -M option takes a value from 1 to 32,767. (Optional)
[-n *netaddr*]	Network address used by the BEA Jolt listener with BEA Tuxedo 6.4 and 6.5, and WebLogic Enterprise 4.2. TCP/IP addresses may be specified in the following formats: "*//host.name:port_number*" "*//#.#.#.#:port_number*" In the first format, the domain finds an address for *hostname* by using the local name resolution facilities (usually DNS). *hostname* must be the local machine, and the local name resolution facilities must unambiguously resolve hostname to the address of the local machine. 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, *port_number* is the TCP port number at which the domain process listens for incoming requests. *port_number* can either be a number between 0 and 65535 or a name.
[-R renegotiation-interval]	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. **Note:** 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.

Table 3-3 JSL Command-line Options

Option	Description
[-S *Client-timeout*]	The idle time (in minutes) when the client does not have any outstanding requests. In other words, when the client is "snoozing." This option can be used together with the -T option. When either timeout reached, JSH will close the session. If a parameter is not specified, the default is no timeout. (Optional)
[-s secure-port]	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. This option cannot be used if the JRLY and JRAD processes are used. The JSL -s option is equivalent to the ISL(5) and WSL(5) -S option. For more information see, *Section 5 - File Formats, Data Descriptions, MIBs, and System Processes Reference*.
[-T *Client-timeout*]	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. 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. If a parameter is not specified, the default is no timeout. (Optional)
[-w *JSH*]	This command-line option indicates the Jolt Server Handler. Default is JSH. (Optional)
[-x *mpx-factor*]	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)
[-Z 0|56|128|256]	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. The DH key exchange is needed to generate keys. Session keys are not transmitted over the network. The default value is 0.

Security and Encryption

When LLE is used for Jolt security and encryption, authentication and key exchange data are transmitted beween Jolt clients and the JSL/JSH using the Diffie-Hellman key echange. 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.

If the SSL section of the Tuxedo license file specifies `STRENGTH=56`, only 0-bit and 56-bit SSL ciphers can be used. If the SSL section of the Tuxedo license file specifies `STRENGTH=128`, then 0-, 56-, 128-, and 256-bit SSL ciphers can be used.

Jolt Relay

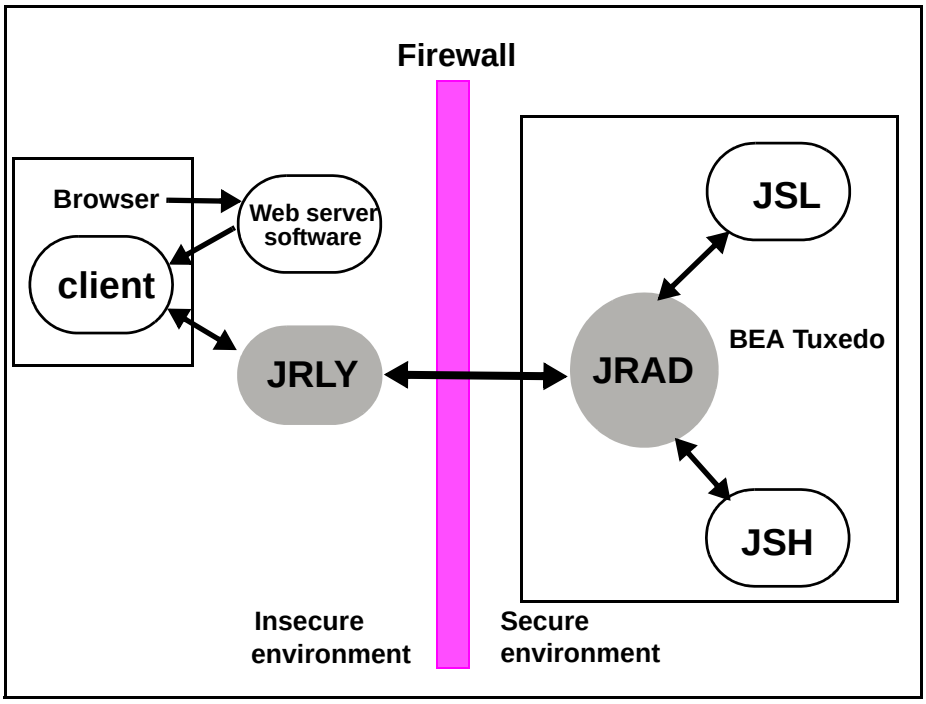
The combination of the Jolt Relay (JRLY) and its associated Jolt Relay Adapter (JRAD) is typically referred to as the Internet Relay. Jolt Relay routes messages from a Jolt client to a JSL or JSH. This eliminates the need for the JSH and BEA Tuxedo to run on the same machine as the Web server (which is generally considered insecure). The Jolt Relay consists of the two components illustrated in the figure "Jolt Internet Relay Path" on page 3-18.

- **Jolt Relay (JRLY)**—the JRLY is the Jolt Relay front-end. It is not a BEA Tuxedo client or server and is not dependent on the BEA 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 a BEA Tuxedo system server, but does not include any BEA Tuxedo services. It requires command-line arguments to allow it to work with the JSL and the BEA 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 3-3 Jolt Internet Relay Path



This figure illustrates how a browser connects to the Web server software and downloads the BEA 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 BEA 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 the following table:

Table 3-4 JRLY Command-line Options for Windows 2003

Option	Description

`jrly -install [display_suffix]`	**Install `jrly` as a Windows 2003 service.**
	
	Example 1:
	`jrly -install`
	In this example, the default JRLY is installed as a Windows 2003 Service and is displayed in the Service Control Manager (SCM) as **Jolt Relay**.
	
	Example 2:
	`jrly -install MASTER`
	In this case, an instance of JRLY is installed as a Windows 2003 Service and is displayed in the SCM as **Jolt Relay_MASTER**. The suffix, MASTER, does not have any significance; it is only used to uniquely identify various instances of JRLYs.
	
	At this point, this instance of JRLY is not ready to start. It must be assigned the configuration file (see the `set` command discussion) that specifies the listening TCP/IP port, JSH connection TCP/IP port, log files, and `sockettimeout`. This file should not be shared between various instances of JRLY.

Table 3-4 JRLY Command-line Options for Windows 2003 (Continued)

`jrly -remove [display_suffix] | -all`	Remove one or all instances of JRLY from Windows 2003 service.
	If `[display_suffix]` is specified, this command removes the specified JRLY service.
	If `[display_suffix]` is not specified, this command removes the default JRLY from being a Windows 2003 Service.
	If the `-all` option is specified, all JRLY Windows 2003 Services are removed. Related Windows 2003 registry entries under
	`HKEY_LOCAL_MACHINE\System\ CurrentControlSet\Services\BEA JoltRelay`
	and
	`HKEY_LOCAL_MACHINE\Software\ BEA Systems\Jolt\x.x`
	are removed.

`jrly -set [-d display_suffix] -f config_file`	Update the registry with the full path of a new configuration file.
	Example 1:
	`jrly -set -f c:\tux71\udataobj\jolt\jrly.con`
	In this example, the default JRLY Windows 2003 Service (Jolt Relay) is assigned a configuration file called `jrly.con` that is located in: `c:\tuxdir\udataobj\jolt` directory.
	Example 2:
	`jrly -set -d MASTER -f c:\tuxdir\udataobj\jolt\master.con`
	Here, the JRLY Windows 2003 Service instance, called **Jolt Relay_MASTER** is assigned a configuration file called `jrly_master.con` that is located in `c:\tuxdir\udataobj\jolt` directory.

`jrly -manual [display_suffix]`	Set the start/stop to manual. This command sets the specified JRLY instance to be manually controlled, using either the command-line options or the SCM.

Table 3-4 JRLY Command-line Options for Windows 2003 (Continued)

`jrly -auto [display_suffix]`	Set the start/stop to automatic. 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.
`jrly -start [display_suffix]`	Start the specified JRLY.
`jrly -stop [display_suffix]`	Stop the specified JRLY.
`jryl -version`	Print the current version of JRLY binary.
`jrly -help`	Print command-line options with brief descriptions.

JRLY Command-line Option for UNIX

There is only one JRLY command-line option for UNIX:

Table 3-5 JRLY Command-line Option for UNIX

Option	Description
`jrly -f config_file_path`	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. The following listing 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.

The following listing 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 the following table.

Table 3-6 Host Name and Port Number Formats

Host Name/Port Number	Descriptions
Hostname:Port	*Hostname* is a string, *Port* is a decimal number
//Hostname:Port	*Hostname* is a string, *Port* is a decimal number
IP:Port	*IP* is a dotted notation IP address, *Port* is a decimal number

Jolt Relay Adapter

The Jolt Relay Adapter (back-end relay) is a BEA Tuxedo system server. The Jolt Relay Adapter (JRAD) server may or may not be located on the same BEA 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 BEA 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 BEA 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 3-26.

The following table contains additional information about the `CLOPT` parameters.

Table 3-7 JRAD CLOPT Parameter Descriptions

CLOPT Parameter	Description
`-l` *netaddr*	Port to listen for the JRLY to connect on behalf of the client.

Table 3-7 JRAD CLOPT Parameter Descriptions (Continued)

CLOPT Parameter	Description
`-c` *`netaddr`*	The address of the corresponding JSL to which JRAD connects.
`-H` *`netaddr`*	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 `-H` argument to an applet, instructing it to connect to the proxy address instead of the JRLY address. **Note:** Unlike the JSL `-H` option, the JRAD `-H` option is not used as a network address translator, nor is it used as an address mask.

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

`//hostname:port`

`0x0002ppppphhhhhhhh`
(where `pppp` is the port number and `hhhhhhhh` is the hexadecimal IP address)

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 the following table and record the information to minimize the possibility of misconfiguration.

Table 3-8 Jolt Internet Relay Network Address Configuration Criteria

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

Jolt Repository

The Jolt Repository contains BEA Tuxedo service definitions that allow Jolt clients to access BEA Tuxedo services. The Jolt Repository files included with the installation contain service definitions used internally by BEA Jolt. See "Using the Jolt Repository Editor" on page 4-1 for detailed instructions on how to add definitions to the application services.

Configuring the Jolt Repository

To configure the BEA Jolt Repository, modify the application `UBBCONFIG` file. The `UBBCONFIG` file is an ASCII version of the BEA Tuxedo configuration file. Create a new `UBBCONFIG` file for each application. See the *BEA Tuxedo Command Reference* for information regarding the syntax of the entries for the file. The following listing shows relevant portions of the `UBBCONFIG` file.

Listing 3-6 Sample UBBCONFIG File

```
*GROUPS
JREPGRP         GRPNO=94 LMID=SITE1
*SERVERS
JREPSVR SRVGRP=JREPGRP SRVID=98
```

```
RESTART=Y GRACE=0 CLOPT="-A -- -W -P /app/jrepository"
JREPSVR SRVGRP=JREPGRP SRVID=97
RESTART=Y RQADDR=JREPQ GRACE=0 CLOPT="-A -- -P /app/jrepository"
JREPSVR SRVGRP=JREPGRP SRVID=96
RESTART=Y RQADDR=JREPQ REPLYQ=Y GRACE=0 CLOPT="-A -- -P /app/jrepository"
```

Note: For UNIX systems, use the slash (/) when setting the path to the `jrepository` file (for example, `app/repository`). For Windows 2003 systems, use the backslash (\) and specify the drive name (for example, `c:\app\repository`).

Change the sections of the `UBBCONFIG` file as indicated in the following table:

Table 3-9 UBBCONFIG File

Section	Parameters to be specified
GROUPS	`LMID, GRPNO`
SERVERS	`SRVGRP, SRVID`

GROUPS Section

A `GROUPS` entry is required for the group that includes the BEA Jolt Repository. The group name parameter is a name selected by the application.

1. Specify the same identifiers given as the value of the `LMID` parameter in the `MACHINES` section.
2. Specify the value of the `GRPNO` between 1 and 30,000 in the `GROUPS` section.

SERVERS Section

The Jolt Repository Server, `JREPSVR`, contains services for accessing and editing the repository. Multiple `JREPSVR` instances share repository information through a shared file. Include `JREPSVR` in the `SERVERS` section of the `UBBCONFIG` file.

1. Indicate a new server identification (for example, `98`) with the `SRVID` parameter.
2. Specify the `-W` flag for one `JREPSVR` to ensure that you can edit the Repository. The Repository is read-only without this flag.

Note: You must install only one writable `JREPSVR` (that is, only one `JREPSVR` with the `-W` flag). Multiple read-only `JREPSVR`s can be installed on the same host.

3. Type the `-P` flag to specify the path of the repository file. An error message is displayed in the BEA Tuxedo `ULOG` file if the argument for the `-P` flag is not entered.
4. Add the file pathname of the repository file (for example, `/app/jrepository`).
5. Boot the BEA Tuxedo system using the `tmloadcf` command (for example, `tmloadcf -y ubbconfig`) and `tmboot` command. See *Administering a BEA Tuxedo Application at Run Time* for information about `tmloadcf` and `tmboot`.

Repository File

A repository file, `jrepository`, is available with BEA Jolt. This file includes `bankapp` services and the repository services that you can modify, test, and delete using the Repository Editor.

Note: If you are upgrading from version 1.*x* of BEA Jolt, you must use the Bulk Loader to regenerate the `jrepository` file in order to ensure compatibility with the current version.

Start with the `jrepository` file provided with the installation, even if you are not going to test the `bankapp` application with BEA Jolt. Delete the `bankapp` packages or services that you do not need.

The pathname of the file must match the argument of the `-P` option.

WARNING: Do not modify the repository files manually or you will not be able to use the Repository Editor. Although the `jrepository` file can be modified and read with any text editor, the BEA Jolt system does not have integrity checks to ensure that the file is in the proper format. Any manual changes to the `jrepository` file might not be detected until run time. See "Using the Jolt Repository Editor" on page 4-1 for additional information.

Initializing Services By Using BEA Tuxedo and the Repository Editor

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

1. Build the BEA Tuxedo server containing the service. See *Administering a BEA Tuxedo Application at Run Time* or *Programming BEA Tuxedo ATMI Applications Using C* for additional information on the following:

- Building the BEA Tuxedo application server
- Editing the `UBBCONFIG` file
- Updating the `TUXCONFIG` file
- Administering the `tmboot` command

2. Access the BEA Jolt Repository Editor. See "Using the Jolt Repository Editor" on page 4-1 for additional information on the following:
   - Adding a Service
   - Saving Your Work
   - Testing a Service
   - Exporting and Unexporting Services

Event Subscription

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

- **Unsolicited Event Notifications—**a Jolt client receives these notifications as a result of a BEA Tuxedo client or service subscribing to unsolicited events, and a BEA 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 BEA Tuxedo Event Broker. The notifications are only received when both Jolt clients subscribe to an event and any BEA Tuxedo client or server posts an event using `tppost()`. Brokered event notifications require the `TMUSREVT` server.

Configuring for Event Subscription

Configure the BEA Tuxedo `TMUSREVT` server and modify the application `UBBCONFIG` file. The following listing shows the relevant sections of `TMUSREVT` parameters in the `UBBCONFIG` file. See *Programming BEA Tuxedo ATMI Applications Using C* for information about the syntax of the entries for the file.

Listing 3-7 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 BEA Tuxedo FML or VIEW Buffers

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

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

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

Buffer Types

Table 3-10 BEA 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.

Table 3-10 BEA Tuxedo Buffer Types

Buffer Type	Description
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 3-32 shows an example that uses the FML buffer. The FML field definition table is made available to BEA Tuxedo by setting the FIELDTBLS and FLDTBLDIR variables.

To filter a field found in the my.flds 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 3-8 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 3-31), 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 BEA Tuxedo system.

For additional information on event subscriptions and the BEA Jolt Class Library, refer to Chapter 5, "Using the Jolt Class Library.".

BEA Tuxedo Background Information

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

Configuration File

The BEA 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 a BEA 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 `BEA Tuxedo` configuration, refer to *Administering a BEA Tuxedo Application at Run Time*.

The following listing shows relevant portions of the `UBBCONFIG` file.

Listing 3-9 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 the following table.

Table 3-11 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 BEA 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. BEA 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 BEA 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 BEA Tuxedo parameters, including `RESTART, RQADDR`, and `REPLYQ`, with the JSL. (See *Administering a BEA 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 3-13 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 BEA Jolt Online Resources

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

`http://www.bea.com/products/jolt/index.htm`

These samples demonstrate and utilize BEA 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.

CHAPTER 4

Using the Jolt Repository Editor

Use the Jolt Repository Editor to add, modify, test, export, and delete BEA Tuxedo service definitions from the Repository based on the information available from the BEA Tuxedo configuration file. The Jolt Repository Editor accepts BEA Tuxedo service definitions, including the names of the packages, services, and parameters.

This topic includes the following sections:

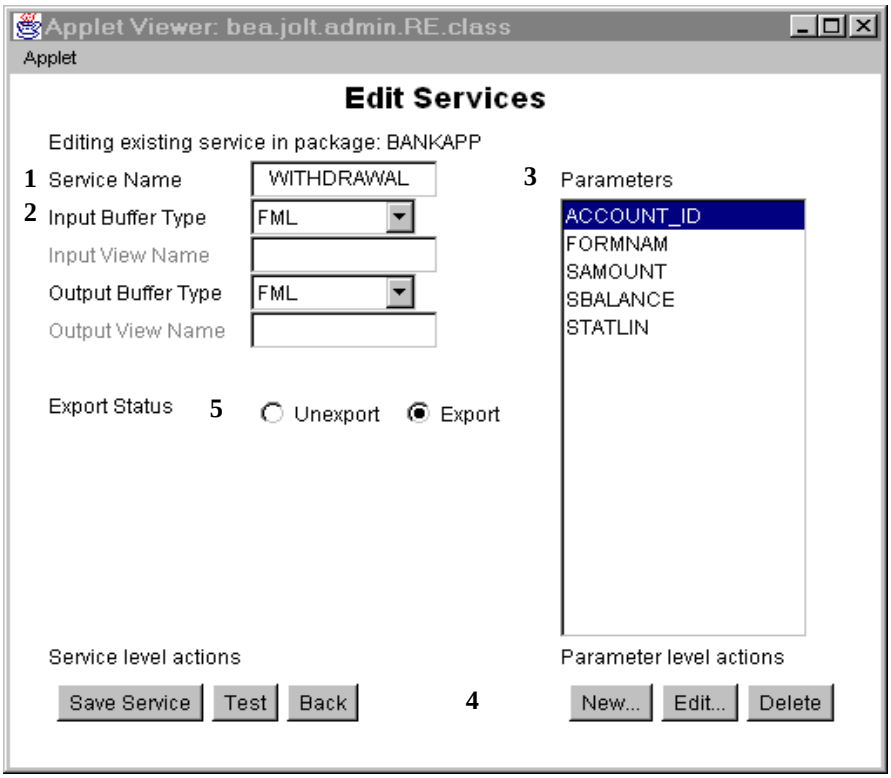
Introduction to the Repository Editor

The Jolt Repository is used internally by Jolt to translate Java parameters to a BEA Tuxedo type buffer. The Repository Editor is available as a downloadable Java applet. When a BEA Tuxedo service is added to the repository, it must be exported to the Jolt server to ensure that the client requests can be made from a Jolt client.

Repository Editor Window

Repository Editor windows contain entry fields, scrollable displays, command buttons, status, and radio buttons. The figure "Sample Repository Editor Window" on page 4-3 illustrates the parts of the window. The table "Repository Editor Window Parts" on page 4-4 contains details about each part.

Figure 4-1 Sample Repository Editor Window



Repository Editor Window Description

The following table details the parts of the Repository Editor window shown in the previous figure.

Table 4-1 Repository Editor Window Parts

Part	Function
1 Text boxes	Enter text, numbers, or alphanumeric characters such as “Service Name,” “Input View Name,” server names, or port numbers. In the previous figure, “Service Name.”
2 Drop-down arrow	View lists that extend beyond the display using an arrow button. In the previous figure, “Input Buffer Type” or “Output Buffer Type.”
3 Display list	Select from a list of predefined items such as the Parameters list or select from a list of items that have been defined.
4 Command buttons	Activate an operation such as displaying the Packages window, Services window, or Package Organizer. In the previous figure, command buttons include: “Save Service,” “Test,” “Back,” “New,” “Edit,” “Delete.”
5 Radio buttons	Select one of a number of options. Only one of the radio buttons can be activated at a time. For example, Export Status can only be “Unexport” or “Export.”

Getting Started

Before starting the Repository Editor, make sure that you install the minimally required components, the Jolt Server and the Jolt Client.

To use the Repository Editor:

1. Start the Repository Editor.

   You can start the Repository Editor from either the JavaSoft `appletviewer` or from your Web browser. Both of these methods are detailed in the following sections.

2. Log on to the Repository Editor.

Note: For information about exiting the Repository Editor after you enter information, refer to "Exiting the Repository Editor" on page 4-8.

Starting the Repository Editor Using the Java Applet Viewer

1. Set the `CLASSPATH` to include the Jolt class directory.
2. If loading the applet from a local disk, type the following at the URL location:

   ```
   appletviewer <full-pathname>/RE.html
   ```

   If loading the applet from the Web server, type the following at the URL location:

   ```
   appletviewer http://<www.server>/<URL path>/RE.html
   ```

3. Press **Enter**.

   The window is displayed as shown in the figure "BEA Jolt Repository Editor Logon Window" on page 4-7.

Starting the Repository Editor from Your Web Browser

Use one of the following procedures to start the Repository Editor from your Web Browser.

To Start from a Local File

1. Set the `CLASSPATH` to include the Jolt class directory.
2. Type the following:

   ```
   file:<full-pathname>/RE.html
   ```

3. Press **Enter**.

The editor is displayed as shown in "BEA Jolt Repository Editor Logon Window" on page 4-7.

To Start from a Web Server

1. Ensure that the `CLASSPATH` does not include the Jolt class directory.
2. Unset the `CLASSPATH`.
3. Type the following:

   ```
   http://<www.server>/<URL path>/RE.html
   ```

   Note: Before opening the file, modify the `applet codebase` parameter in `RE.html` to match your Jolt Java classes directory.

4. Press **Enter**.

   The editor is displayed as shown in the "BEA Jolt Repository Editor Logon Window" on page 4-7.

Logging On to the Repository Editor

Note: If you are using the JDK 1.3 appletviewer to start the Jolt Repository Editor, you will *not* be able to connect to a remote machine, only to a local host JSL. This is due to a security restriction imposed in the JDK 1.3 appletviewer. Also, for JDK 1.2, you must use the `-nosecurity` option in the appletviewer if you are connecting to a remote machine JSL.

1. Complete the appropriate steps to start the Repository Editor.

   The "BEA Jolt Repository Editor Logon Window" on page 4-7 must be displayed before you continue with step 2. Refer to this figure as you perform the following procedure.

2. Type the name of the Server machine designated as the "access point" to the BEA Tuxedo application and press **Tab**.
3. Type the Port Number and press **Enter**.

   The system validates the server and port information.

   Note: Unless you are logging on through the Jolt Relay, the same port number is used to configure the Jolt Listener. Refer to your `UBBCONFIG` file for additional information.

4. Type the BEA Tuxedo Application Password and press **Enter**.

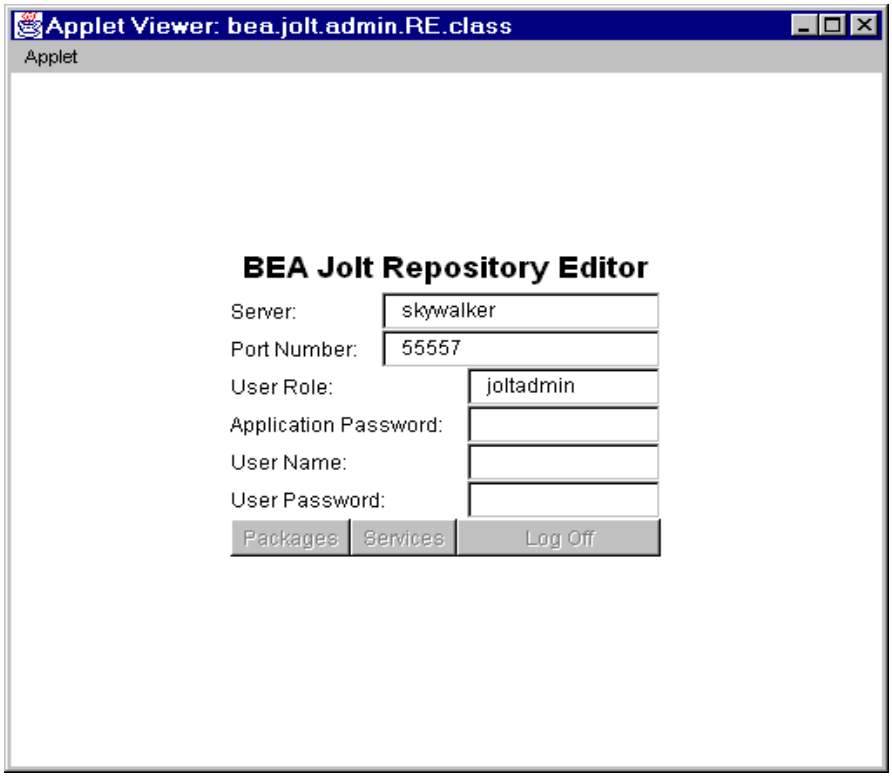
   Depending upon the authentication level, complete steps 5 and 6 as required.

5. Type the BEA Tuxedo User Name and press **Tab**.
6. Type the BEA Tuxedo User Password and press **Enter**.

   The **Packages** and **Services** command buttons are enabled.

   Note: See the `JoltSessionClass` for additional information.

Figure 4-2 BEA Jolt Repository Editor Logon Window



The following table, "Repository Editor Logon Window Description," describes Repository Editor logon window elements.

Repository Editor Logon Window Description

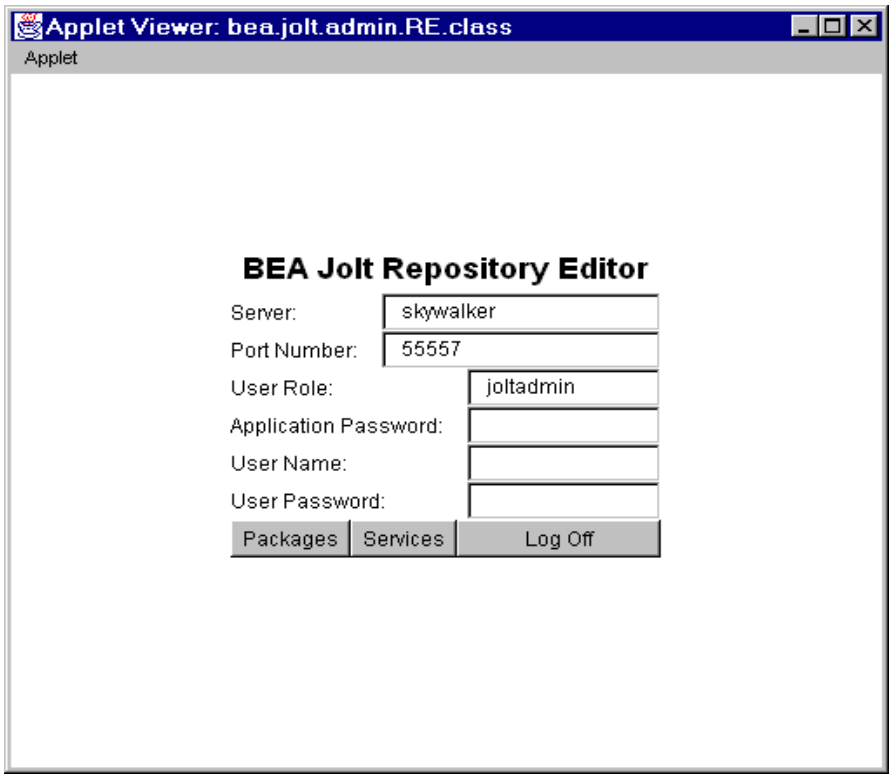
Table 4-2 Repository Editor Logon Window Description

Option	Description
Server	Server name.
Port Number	Port number in decimal value. **Note:** After the Server Name and Port Number are entered, the User Name and Password fields are activated. Activation is based on the authentication level of the BEA Tuxedo application.
User Role	BEA Tuxedo user role. Required only if BEA Tuxedo authentication level is USER_AUTH or higher.
Application Password	BEA Tuxedo administrative password text entry.
User Name	BEA Tuxedo user identification text entry. The first character must be an alpha character.
User Password	BEA Tuxedo password text entry.
Packages	Accesses the Packages window. (Enabled after the logon.)
Services	Accesses the Services window. (Enabled after the logon.)
Log Off	Terminates the connection with the server.

Exiting the Repository Editor

Exit the Repository Editor when you finish adding, editing, testing, or deleting packages, services, and parameters. Prior to exit, the window is displayed as shown in the figure "BEA Jolt Repository Editor Logon Window Prior to Exit" on page 4-9.

Figure 4-3 BEA Jolt Repository Editor Logon Window Prior to Exit



Note that only the **Packages**, **Services**, and **Log Off** command buttons are enabled. All of the text entry fields are disabled.

Follow the steps below to exit the Repository Editor:

1. Click **Back** to return to the Repository Editor Logon window.
2. Click **Log Off** to terminate the connection with the server.

   The Repository Editor Logon window continues to be displayed with disabled fields.
3. Select **Close** from your browser menu to close the window.

Main Components of the Repository Editor

The Repository Editor allows you to add, modify, or delete any of the following components:

- Packages
- Services

  You can also test and group services.

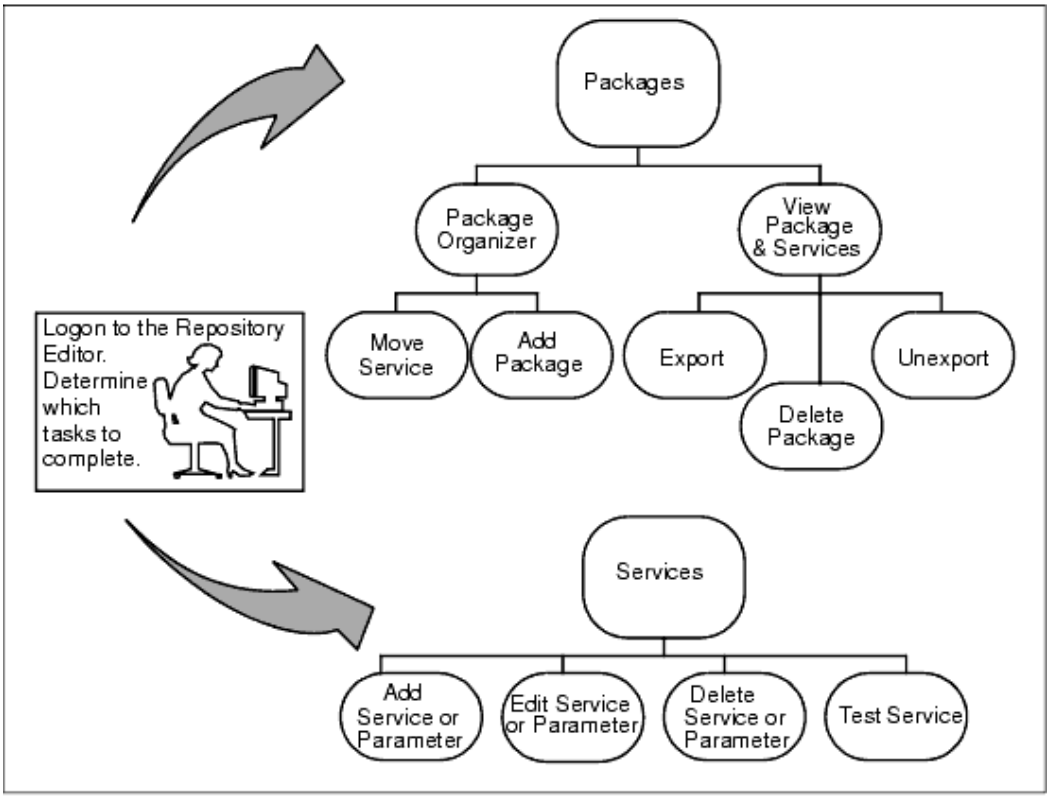
- Parameters

Repository Editor Flow

After you log on to the Repository Editor, two buttons are enabled, **Packages** and **Services**.

The following figure illustrates the Repository Editor flow to help you determine which of these two buttons to select.

Figure 4-4 Repository Editor Flow Diagram



Select **Packages** to open the Packages window and perform the following functions:

- View packages and services
  - Make a service available using **Export** or **Unexport**
  - Select a package to delete
- Access the Package Organizer to:
  - Move services from one package to another
  - Create a new package

Refer to "What Is a Package?" on page 4-12 for complete details.

Select **Services** to open the Services window and perform the following functions:

- Create, add, edit, or delete service definitions
- Create, add, edit, or delete parameters
- Test the services and parameters

Refer to "What Is a Service?" on page 4-15 for complete details.

What Is a Package?

Packages provide a convenient method for grouping services for Jolt administration. (A service consists of parameters, such as pin number, account number, payment, rate, term, age, or Social Security number.)

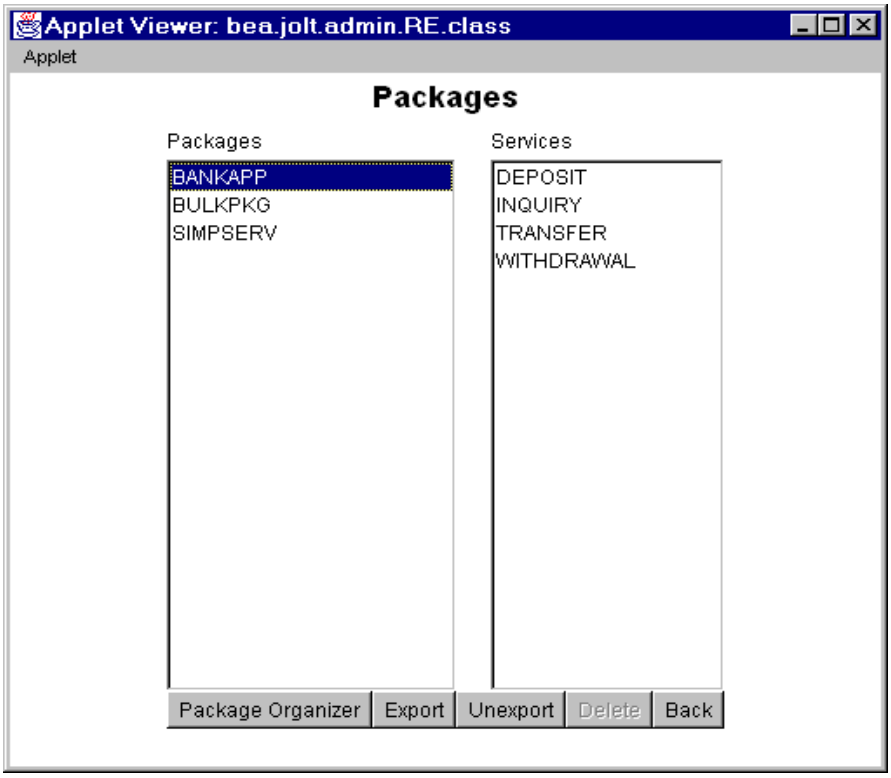
You use the **Packages** window to perform the following:

- View packages and services
- Export or unexport services
- Delete packages
- Access Package Organizer to:
  - Move services
  - Create a new package

Click the **Packages** button in the Jolt Repository Editor logon window to display the available packages. When you select a specific package from the display list, its services within that package are displayed.

The following figure contains a sample Packages window. The BANKAPP package is selected, and the services within the BANKAPP package is displayed.

Figure 4-5 Sample Packages Window



Packages Window Description

Table 4-3 Packages Window Description

Option	Description
Packages	Lists available packages.
Services	Lists available services within the selected package.
Package Organizer	Accesses the Package Organizer window to review available packages and services. Use this window to move the services among the packages or add a new package.
Option	**Description**

Table 4-3 Packages Window Description (Continued)

Export	Makes the most current services available to the client. This option is enabled when a package is selected.
Unexport	Select this option before testing an existing service. This option is enabled when a package is selected.
Delete	Deletes a package. This option is enabled when a package is selected and the package is empty (no services contained within the package).
Back	Returns the user to the previous window.

Instructions for Viewing a Package

1. Click **Packages** in the Repository Editor Logon window.

   The Packages window opens and displays the list of available packages.

   In the figure "Sample Packages Window" on page 4-13, `BANKAPP`, `BULKPKG`, and `SIMPSERV` are the available packages.

2. Refer to "Instructions for Viewing a Parameter" on page 4-17 for additional information.

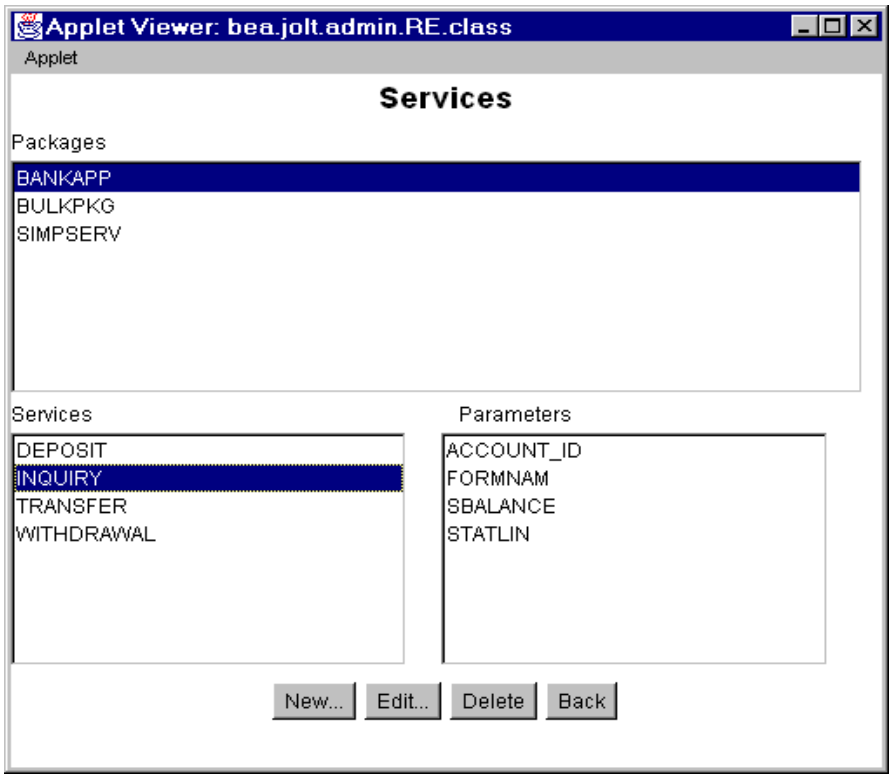
What Is a Service?

A service is a definition of an available BEA Tuxedo service. Services include parameters such as pin number, account number, payment, and rate. Adding or editing a Jolt service does not affect an existing BEA Tuxedo service.

You use the Services Window to add, edit, or delete services.

The following figure is an example of a Services window with the BANKAPP package selected, and the display list of services and parameters available for this package (parameters are detailed later).

Figure 4-6 Sample Services Window



Services Window Description

Table 4-4 Services Window Description

Option	Description
Packages	Lists the available packages.
Services	Lists the services in the selected package, which you can edit or delete. Selecting a service displays the parameters within the service.
Parameters	Displays the parameters of the selected service.
New	Displays the Edit Services window for adding a new service.
Edit	Displays the Edit Services window for editing an existing service. This button is enabled only if a service has been selected.
Delete	Deletes a service. This button is only enabled if a service has been selected.
Back	Returns the user to the previous window.

Instructions for Viewing a Service

1. Select **Services** from the Repository Editor Logon window.

   The Services window opens and displays the list of available packages.

2. Select a package.

   The list of available services for the selected package is displayed.

   In the figure "Sample Services Window" on page 4-15, `BANKAPP` is the selected package. `DEPOSIT`, `INQUIRY`, `TRANSFER`, and `WITHDRAWAL` are the available services for `BANKAPP`.

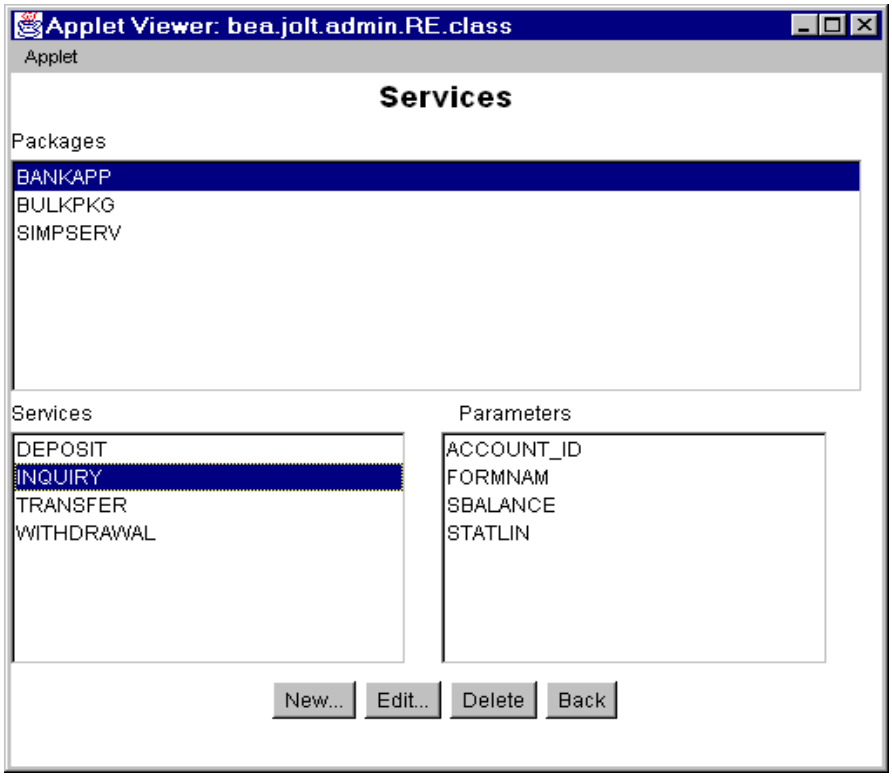
3. Refer to "Instructions for Viewing a Parameter" on page 4-17 for additional information.

Working with Parameters

A service contains parameters, which may be a pin number, account number, payment, rate, term, age, or Social Security number. The following figure shows a Services window displaying a selected service and its parameters.

Note: Adding or editing a parameter does not modify or change an existing BEA Tuxedo Service.

Figure 4-7 Sample Services Window with Parameters List



Instructions for Viewing a Parameter

1. Select **Services** from the Repository Editor Logon window.

   The Services window opens and displays the list of available packages.

2. Select a package.

   The list of available services for the selected package is displayed.

   In the preceding figure, BANKAPP is the selected package.

3. Select a service.

   The list of available parameters for the selected service is displayed.

   In the preceding figure, INQUIRY is the selected service.

4. View the parameters for a selected service in the Parameters display list.

   In the preceding figure, ACCOUNT_ID, FORMNAM, SBALANCE, and STATLIN are the available parameters for the INQUIRY service.

5. Refer to "Adding a Parameter" on page 4-24 for additional information.

Setting Up Packages and Services

This section includes the necessary steps for setting up a package and its services:

- Saving your work
- Adding a package
- Adding a service
- Adding a parameter

Saving Your Work

As you create and edit services and parameters, it is important to regularly save information to avoid losing input. Clicking **Save Service** in the Edit Services window can prevent the need to re-enter information in the event of a system failure.

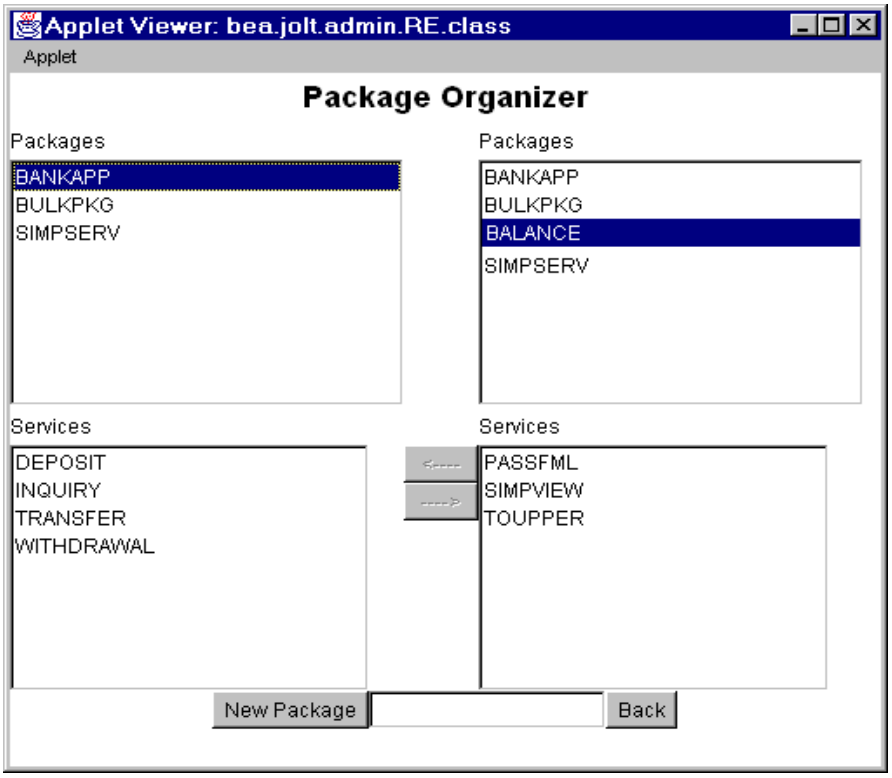
Caution: When you add or edit the parameters of a service, you must select **Add** before choosing **Back** from the Edit Parameters window and returning to the Edit Services window.

If adding a new service or modifying an existing service in the Edit Services window, be sure to select **Save Service** before choosing **Back**. If you select **Back** before you save the modified information, a warning is briefly displayed on the status line at the bottom of the window.

Adding a Package

When you need to add a new group of services, you create a new package before adding the services. The "Package Organizer Window" on page 4-19 and the following procedure show how to add a new package, BALANCE, to the Packages listing.

Figure 4-8 Package Organizer Window



Instructions for Adding a Package

1. Click **Packages** in the Repository Editor Logon window to display the Packages window.
2. Select **Package Organizer** to display the Package Organizer window, similar to that shown in the figure "Package Organizer Window" on page 4-19.

   For a description of contents of this window, see "Package Organizer Window Description" on page 4-30.

3. Click the **New Package** button in the Package Organizer window.

   The text field is activated.

4. Type the name of the new package (not to exceed 32 characters) and press **Enter**.

   The new name (shown in the preceding figure as `BALANCE`) is displayed on the Packages list in random order.

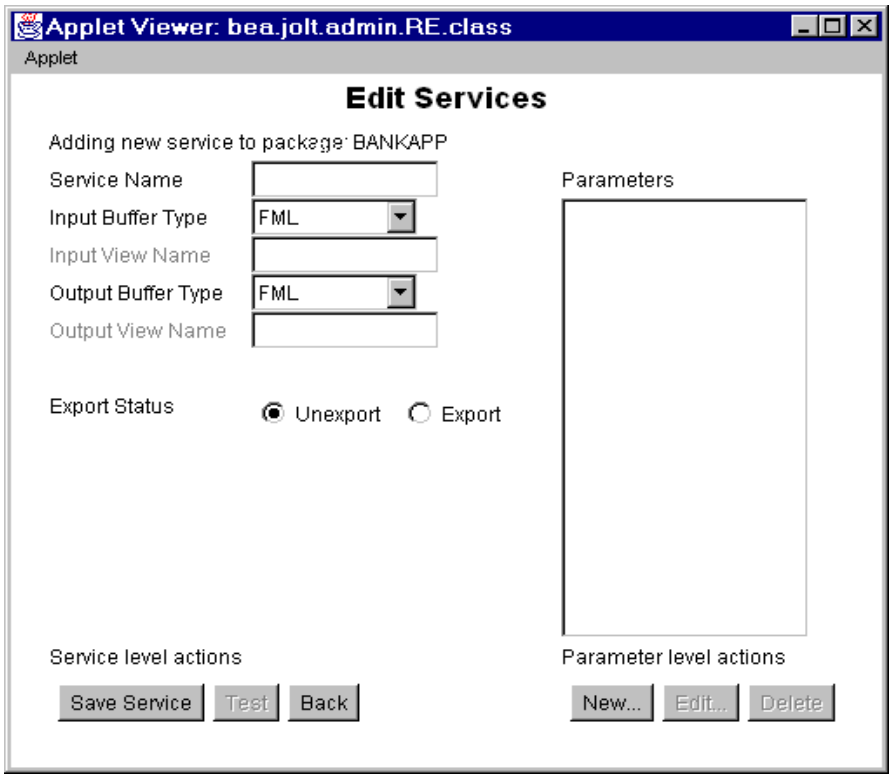
Adding a Service

Services are definitions of available BEA Tuxedo services and can only be a part of a Jolt package.You must create the service as a part of a new or existing package.

The Repository Editor accepts the new service name exactly as it is typed (that is, all uppercase letters, abbreviations, misspellings are accepted). Service names must not exceed 30 characters.

The following figure shows the Edit New Services window for adding a service.

Figure 4-9 Edit Services Window: Add a New Service to a Package



Adding a Service Window Description

The following table describes the options for adding services to a package in a package window.

	Option	Description
Edit Services Selections	**Service Name**	Name of the new service to be added to the Jolt Repository.
	Input Buffer Type/Output Buffer Type	VIEW— C-structure and 16-bit integer field. Contains subtypes that have a particular structure. X_C_TYPE and X_COMMON are equivalent. X_COMMON is used for COBOL and C. VIEW32—similar to VIEW, except 32-bit field identifiers are associated with VIEW32 structure elements. CARRAY—array of uninterrupted binary data that is neither encoded nor decoded during transmission; it may contain null characters. X_OCTET is equivalent. XML—well-formed XML document. Similar to CARRAY. FML—type in which each field carries its own definition. FML32—similar to FML except the ID field and length field are 32 bits long. STRING—character array terminated by a null character that is encoded or decoded.
	Input View Name/Output View Name	Unique name assigned to the Input View Buffer and Output View Buffer types. These fields are only enabled if VIEW or VIEW32 are the selected buffer types.
Export Status	**Unexport** **Export**	Radio button with current status of the service. EXPORT or UNEXPORT status is checked. UNEXPORT is the default.
Service Level Actions	**Save Service**	Saves the newly created service in the Repository.
	Test	Tests the service. This command button is disabled until a new service is created or edits to an existing service are saved.
	Back	Returns you to the previous window.
Parameter	**Parameters**	List of service parameters to edit or delete.
Parameter Level Actions	**New**	Adds new parameters to the service.
	Edit	Used to edit an existing parameter. This command button is disabled until a new parameter is selected.
	Delete	Deletes a parameter. This option is disabled until a parameter is selected.

Instructions for Adding a Service

1. Select **Services from the Repository Editor Logon window**.

   The Services window opens, similar to the figure shown in "Sample Services Window" on page 4-15.

2. Select the package to which you will add the service.

   If you later decide that another package should contain the new service, use the Package Organizer to move the service to a different package. (See "Grouping Services Using the Package Organizer" on page 4-29 for additional information.)

3. From the Services window, select **New** to display the Edit Services window, as shown in "Edit Services Window: Add a New Service to a Package" on page 4-21.

4. Select the **Service Name** text field to activate it.

5. Type the name of the new service you want to add.

6. Select the input buffer type.

   Although the same buffer type selected for the Input Buffer is automatically selected for the Output Buffer, you can select a different Output Buffer type.

   – If `VIEW` or `VIEW32` is selected, you must type the Input View Name and Output View Name in the associated text fields.

   – If another buffer type is selected, the Input View Name and Output View Name text fields are disabled.

   – If `CARRAY` or `STRING` is selected, refer to "Selecting CARRAY or STRING as a Service Buffer Type" on page 4-23 for additional instructions.

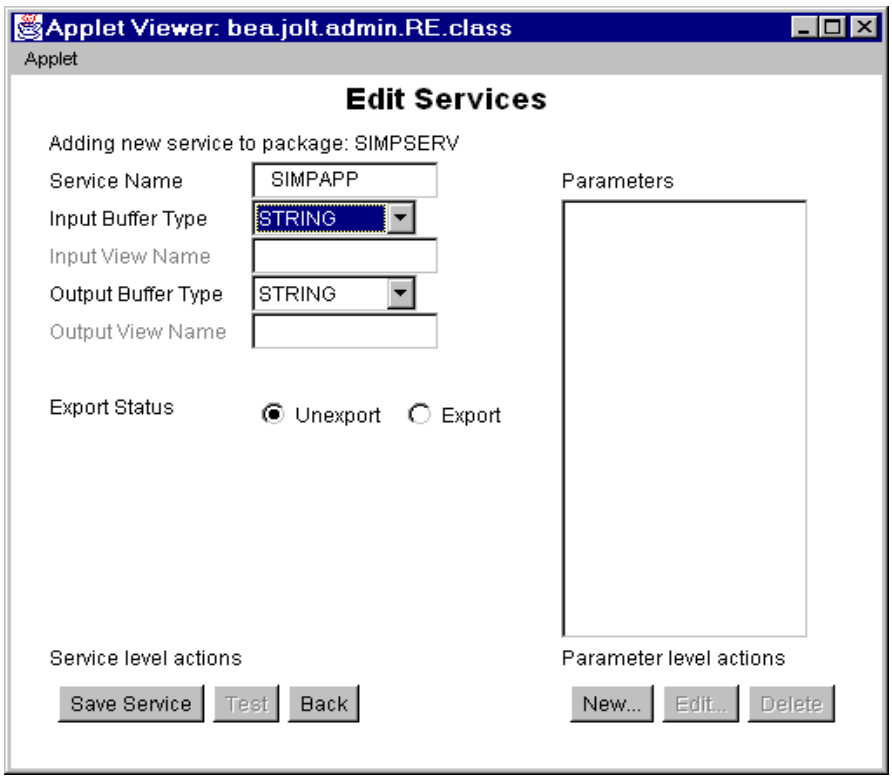
7. Select **Save Service** to save the newly created service.

Selecting CARRAY or STRING as a Service Buffer Type

If `CARRAY` or `STRING` is selected as the buffer type for a new service, only `CARRAY` or `STRING` can be added as the data type for the accompanying parameters. See also "Adding a Parameter" on page 4-24 and "Selecting CARRAY or STRING as a Parameter Data Type" on page 4-27. For additional information, refer to Chapter 5, "Using the Jolt Class Library.".

The following figure shows an example Edit Services window with `STRING` selected as the buffer type for the service `SIMPAPP`.

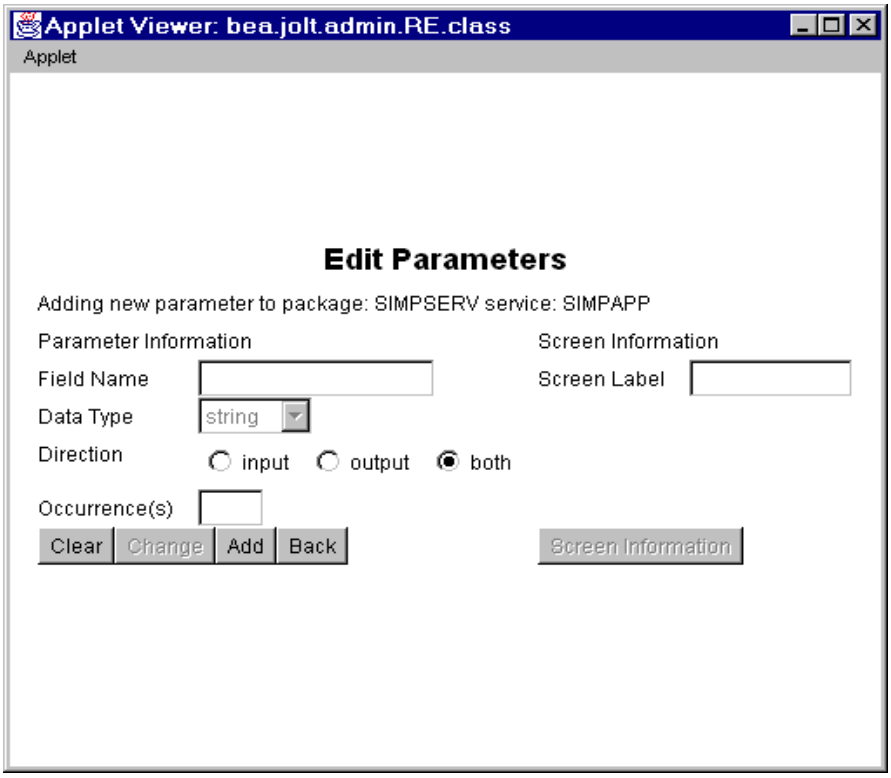
Figure 4-10 Edit Services Window: Select STRING Buffer Type



Adding a Parameter

Clicking **New under the label Parameter level actions** in the Edit Services window is displayed in the Edit Parameters window. Review the features in the following figure. Use this window to enter the parameter and screen information for a service.

Figure 4-11 Edit Parameters Window: Add a Parameter



Adding a Parameter Window Description

Option	Description
Field Name	Adds the field name (for example, asset, inventory).
Data Type	Lists data type choices: byte—8-bit short—16-bit integer—32-bit float—32-bit double—64-bit string—null-terminated character array carray—variable length 8-bit character array
Direction	Radio button choices for direction of information: Input—information is directed from the client to the server. Output—information is directed from the server to the client. Both—information is directed from the client to the server, and from the server to the client.
Occurrence(s)	Number of times that an identical field name can be used. If 0, the field name can be used an unlimited number of times. Occurrences are used by Jolt to build test screens; not to limit information sent or retrieved by BEA Tuxedo.
Screen Information	This button is disabled when the window is launched.
Clear	Clears the fields of the window.
Change	Is disabled while new parameters are added.
Add	Adds new parameters to the service. The parameters are saved when the service is saved.
Back	Returns the user to the previous window.

Instructions for Adding a Parameter

1. Select **Field Name** to activate the field, and type the field name.

   Note: If the buffer type is FML or VIEW, the field name must match the corresponding parameter field name in FML or VIEW.

2. Select the data type.
3. Specify a direction by selecting the **input**, **output**, or **both** radio buttons.
4. Select the **Occurrences** text field to activate it, and then enter the number of occurrences.
5. Select **Add** to append the information. **Add** does not save the parameter.
6. In the Edit Services window, click **Save Service** to save the parameter as a part of the service.

   WARNING: If you do not click **Save Service** before you click **Back**, the parameters are not saved as part of the service.

7. Click **Back** to return to the Edit Services window.

Selecting CARRAY or STRING as a Parameter Data Type

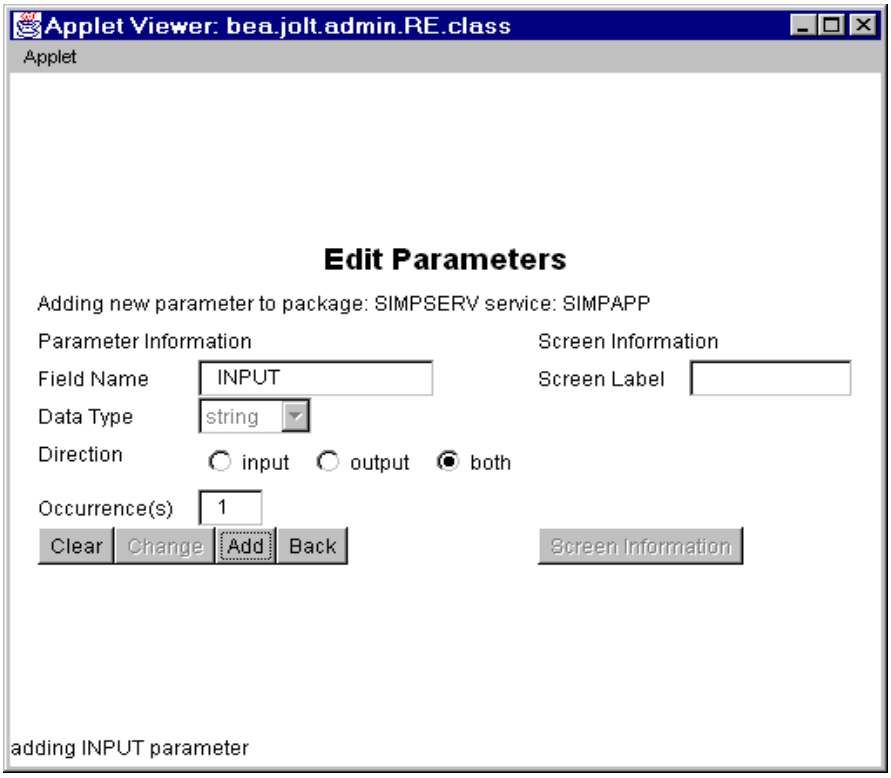
If CARRAY or STRING is the selected buffer type for a new service, only CARRAY or string can be added as the data type for the accompanying parameters.

In this case, only one parameter can be added. It is recommended that you use the parameter name “CARRAY” for a CARRAY buffer type, and the parameter name “STRING” for a STRING buffer type.

See also “Instructions for Adding a Service” on page 4-23 and “Selecting CARRAY or STRING as a Service Buffer Type” on page 4-23. For additional information, refer to Chapter 5, “Using the Jolt Class Library.”.

The following figure is an example of the Edit Parameters window with string as the selected data type for the parameter. The **Data Type** defaults to string and does not allow you to modify that particular data type. The **Field Name** can be any name.

Figure 4-12 Edit Parameters Window: string Data Type



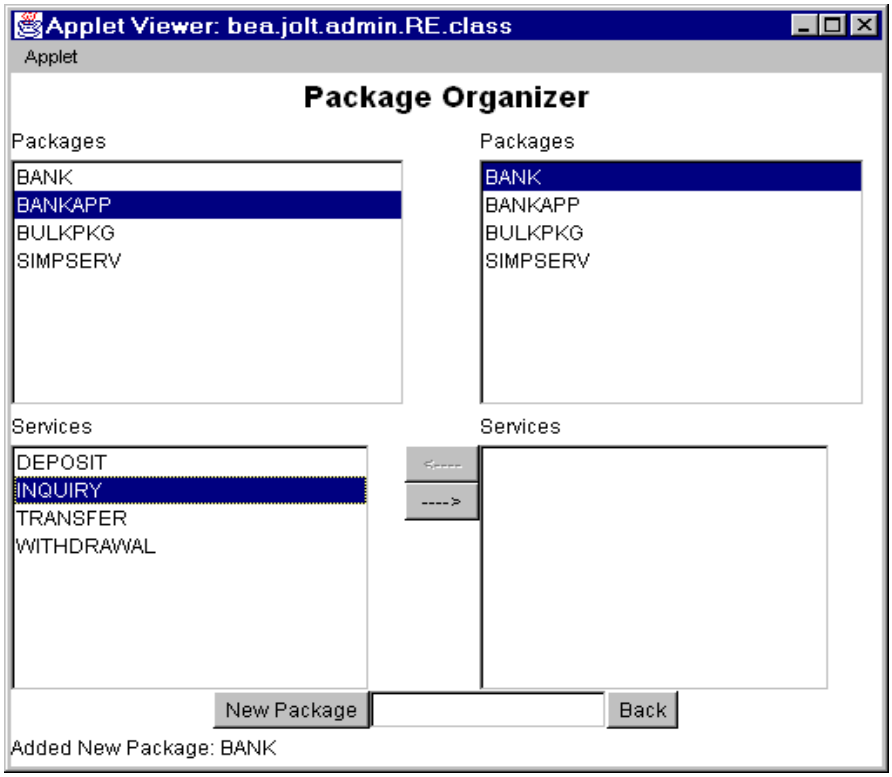
Grouping Services Using the Package Organizer

The Package Organizer moves services between packages. You may want to group related services in a package (for example, WITHDRAWAL services that are exported only at a certain time of the day can be grouped together in a package).

Use the Package Organizer arrow buttons to move a service from one package to another. These buttons are useful if you have several services to move between packages. The packages and services display listings to help track a service within a particular package.

The following figure is an example of a Package Organizer window with a service selected for transfer to another package.

Figure 4-13 Package Organizer Window



Package Organizer Window Description

Option	Description
Packages (left display list)	Lists packages containing services in the selected package.
Packages (right display list)	Lists packages available as destinations for the selected service.
Services (left display list)	Lists available services for the selected package.
Services (right display list)	Lists available services of the highlighted package that you moved.
Left arrow	Moves services (one service at a time) to the package highlighted on the left.
Right arrow	Moves services (one service at a time) to the package highlighted on the right.
New Package	Adds the name of a new package.
Back	Returns user to the previous window.

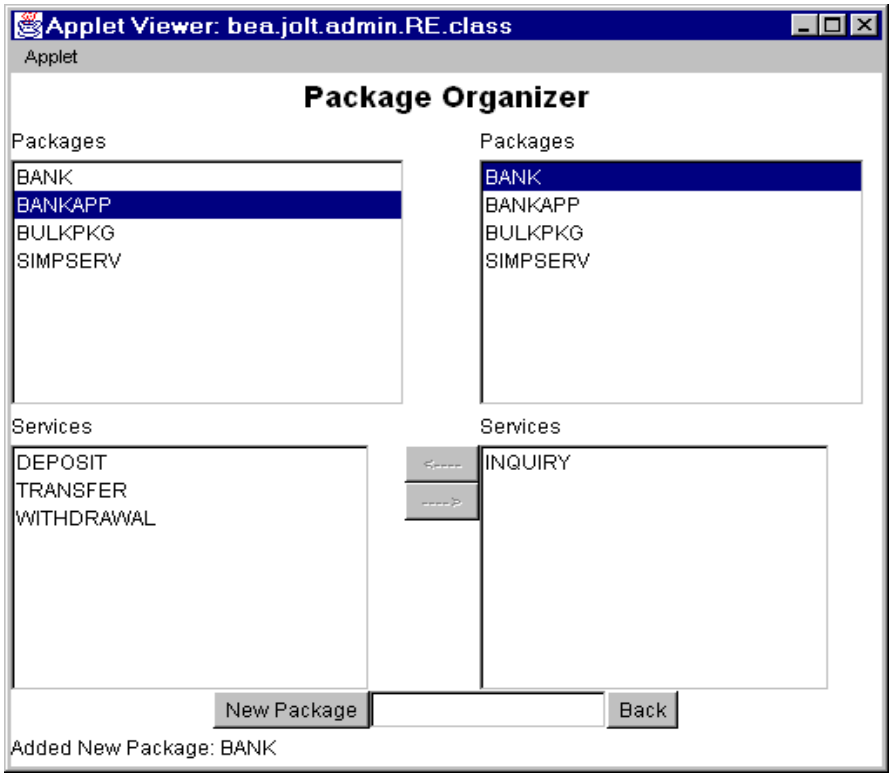
Instructions for Grouping Services with the Package Organizer

1. In the Packages window, click Package Organizer.
2. In the Package Organizer window, select the package containing the services to be moved from the Packages left display window.
3. Select the service to be moved from the Services left display window.

   In the previous figure, `INQUIRY` is the selected service in the `BANKAPP` package.
4. Select the package to receive the service from the Packages right display window.

   The previous figure shows the selected service, `INQUIRY`, and the selected package, `BANK`, to which the `INQUIRY` service will be moved.

Figure 4-14 Example of a Moved Service



5. To move the service between the packages, select the left arrow **(←)** or right arrow **(→)**.

   These keys are activated only when both packages (left and right are displayed) and a service are selected. The keys are only active in the direction of the package where the service is to be moved. The previous figure, "Example of a Moved Service," shows that the `INQUIRY` service has been moved to the `BANK` package on the right.

   Note: You cannot select the same package in both the left and right display lists.

Modifying Packages, Services, and Parameters

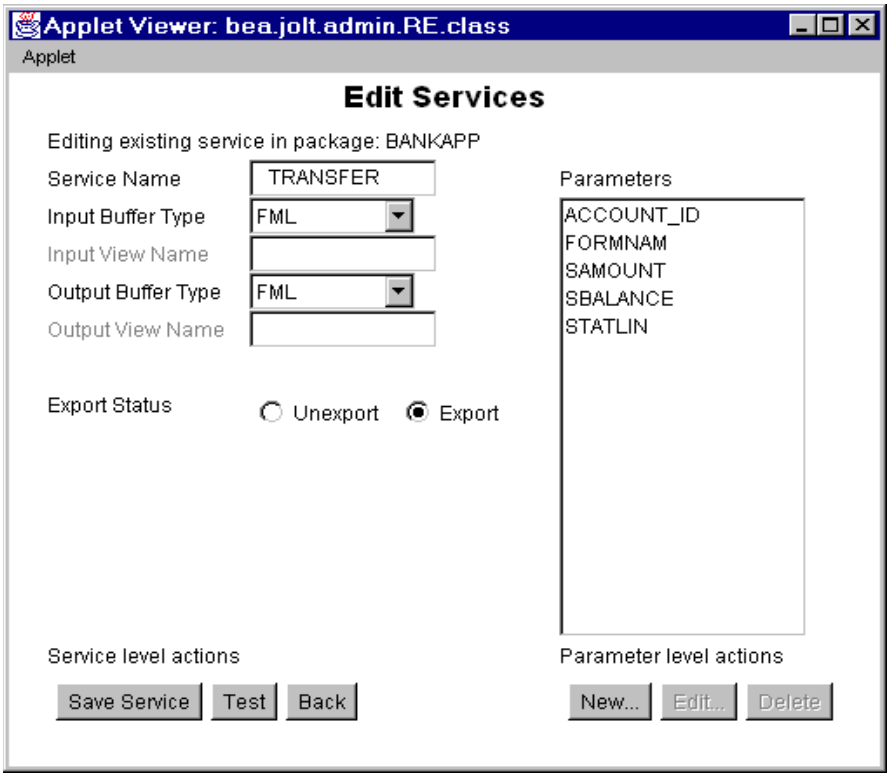
You can make the following changes to packages, services, and parameters:

- Edit a service
- Edit a parameter
- Delete a parameter, service, or package

Editing a Service

You can edit an existing service name or service information, or access the window to add new parameters to an existing service. For a description of the Edit Services window, see "Adding a Service Window Description" on page 4-21. The following figure is an example of the Edit Services window.

Figure 4-15 Edit Services Window



Instructions for Editing a Service

Follow these steps to edit a service:

1. From the Services window, select the package containing the service that requires editing.

   The services available for the selected package are displayed.

2. Select the service to edit.

   The parameters available for the selected service are displayed.

3. Click **Edit** to display the Edit Services window, as shown in the previous figure.

4. Type or select the new information, and click **Save Service**.

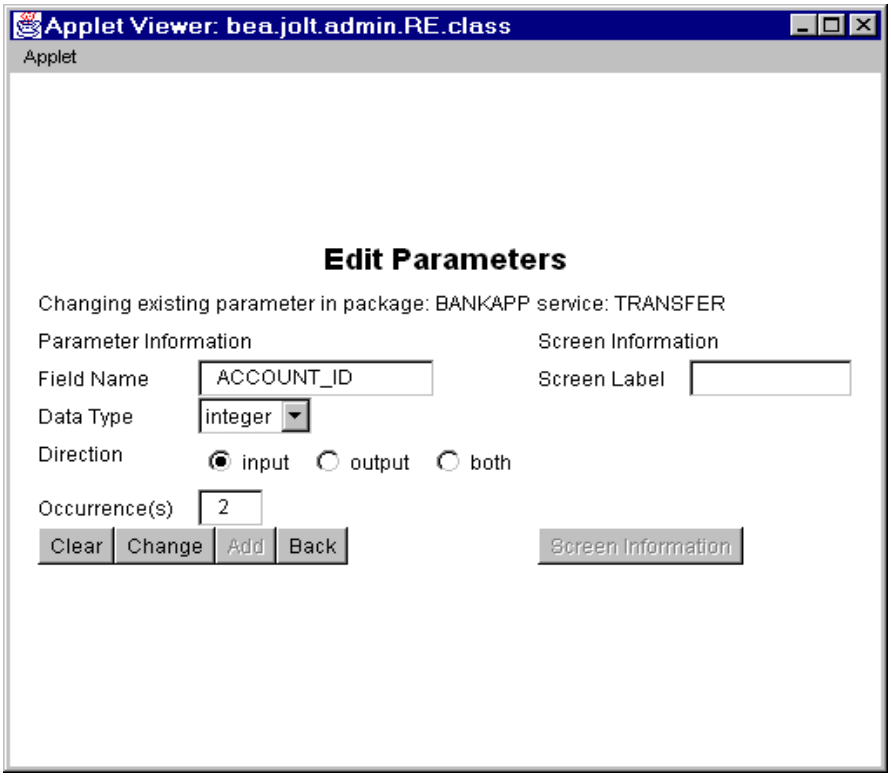
Editing a Parameter

All parameter elements can be changed, including the name of the parameter.

WARNING: If you create a new parameter using an existing name, the system overwrites the existing parameter.

The following figure is an example of the Edit Parameters window.

Figure 4-16 Edit Parameters Window



Instructions for Editing a Parameter

Follow these steps to change a parameter:

1. In the Services window (see "Sample Services Window with Parameters List"), select the package and service that contain the parameter you want to change.

2. Click Edit to display the Edit Services window.
3. Select the Parameter you want to edit from the Parameters display list and click **Edit**.

   The Edit Parameters Window is displayed as shown in the previous figure.
4. Type the new information and click **Change**.
5. Click **Back** to return to the previous window.

Deleting Parameters, Services, and Packages

This section describe how to delete a package. Before deleting a package, all the services must be deleted from the package. The **Delete** option is not enabled until all components of the package or service are deleted.

WARNING: The system does not display a prompt to confirm that items are to be deleted. Be certain that the parameter, service, or package is scheduled to be deleted or has been moved to another location before selecting **Delete**.

Deleting a Parameter

Determine which parameters to delete and follow these steps:

1. In the logon window, click Services to display the Services window.
2. In the Services window, select the package and service that contain the parameter you want to delete.
3. Click **Edit** to display the Edit Services window.
4. Select the parameter you want to delete from the Parameters display list.
5. Under Parameter Level Actions, click **Delete**.

Deleting a Service

Determine which services to delete and follow these steps:

Note: Make certain that all parameters within this service are deleted before selecting this option.

1. Select the package containing the service you want to delete.
2. Select the service you want to delete.

Delete is enabled.

3. Click **Delete**. The service is deleted.

Deleting a Package

Determine which packages to delete and follow these instructions. Make sure all services contained in this package are deleted or moved to another package before selecting this option.

1. In the Repository Editor Logon window, click **Packages** to display the Packages window.
2. Select a package.
3. Click **Delete**.

   The package is deleted.

Making a Service Available to the Jolt Client

To make a service available to a Jolt client, you export it. All services in a package must be exported or unexported as a group. A service is made available by using the **Export** and **Unexport** radio buttons.

This topic includes the following sections:

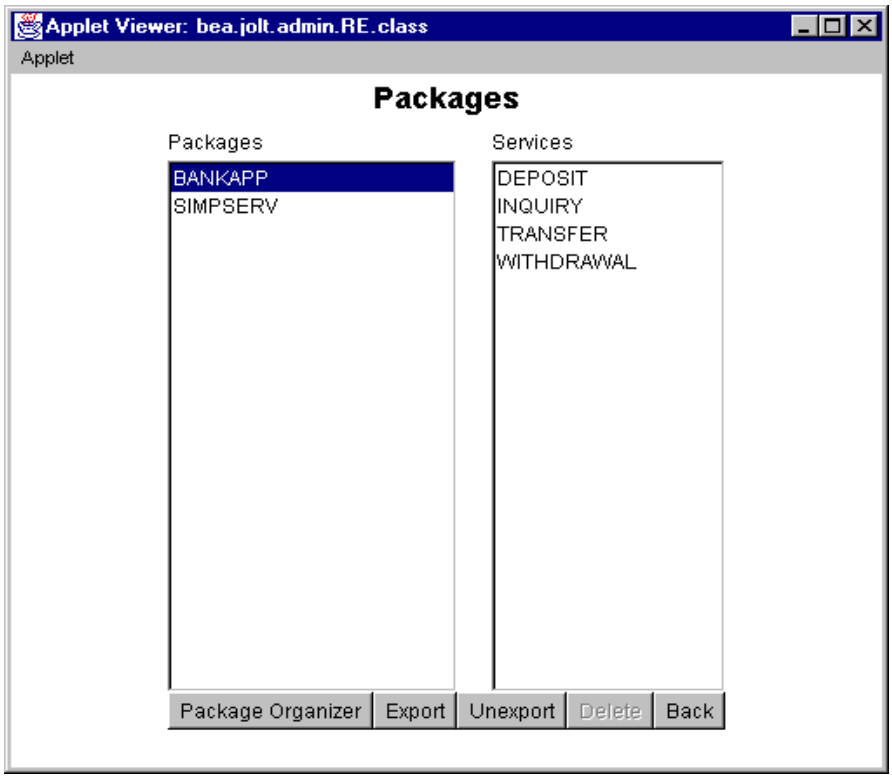
- Exporting and Unexporting Services
- Reviewing the Exported and Unexported Status

Exporting and Unexporting Services

Determine which services are being made available or unavailable to the Jolt client. Services are exported to ensure that the Jolt client can access the most current service definitions from the Jolt server.

The following figure shows the Packages window, where you can **export** and **unexport** services.

Figure 4-17 Packages Window: Export and Unexport Buttons



Follow these steps to export or unexport a service:

1. From the Repository Editor Logon window, select **Packages** to display the Packages window.
2. Select a package.

   The Export and **Unexport** buttons are enabled.
3. To make the services in the selected package available, click **Export**.

   To make the services in the selected package unavailable, select **Unexport**.

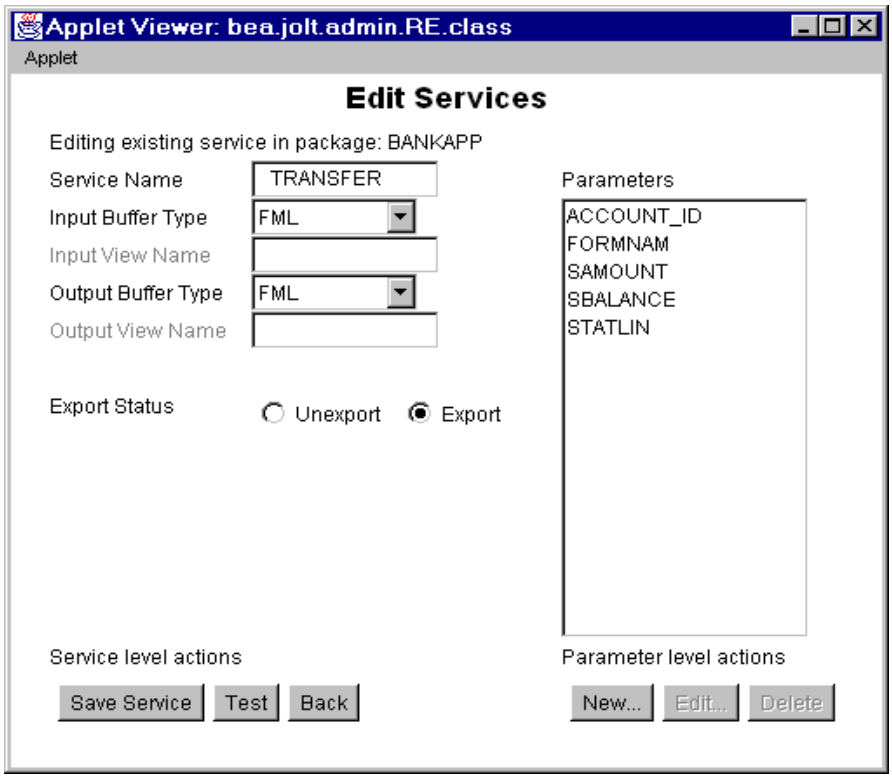
Caution: The system does not display a confirmation message indicating that the service is exported or unexported. See "Reviewing the Exported and Unexported Status" for additional information.

Reviewing the Exported and Unexported Status

When a service is exported or unexported, you can review its status from the Edit Services window.

The following figure displays the **Export** radio button as active, for **Export Status**; therefore, the current status for the service `TRANSFER` is exported.

Figure 4-18 Export Status



To review the current exported or unexported status of a service, follow these steps:

1. From the Repository Editor Logon window, select **Services** to display the Services window shown in the "Sample Services Window" on page 4-15.
2. Select a package from the Package display list.

   The Services display list of available services for the selected package is displayed.

3. Select the service you want to review.
4. Click **Edit**.

   The Edit Services window is displayed as shown in the figure "Edit Services Window" on page 4-33.

   One of the radio buttons (**Unexport** or **Export**) next to the **Export Status** label will be active, indicating the current status of the service.

Testing a Service

Test a service and its parameters before you make them available to Jolt clients. You can test currently available services without making changes to the services and parameters.

Note: The Jolt Repository Editor allows you to test an existing BEA Tuxedo service with Jolt, without writing a line of Java code.

An exported or unexported service can be tested; if you need to change a service and its parameters, unexport the service prior to editing.

This topic includes the following sections:

- Jolt Repository Editor Service Test Window
- Testing a Service

Jolt Repository Editor Service Test Window

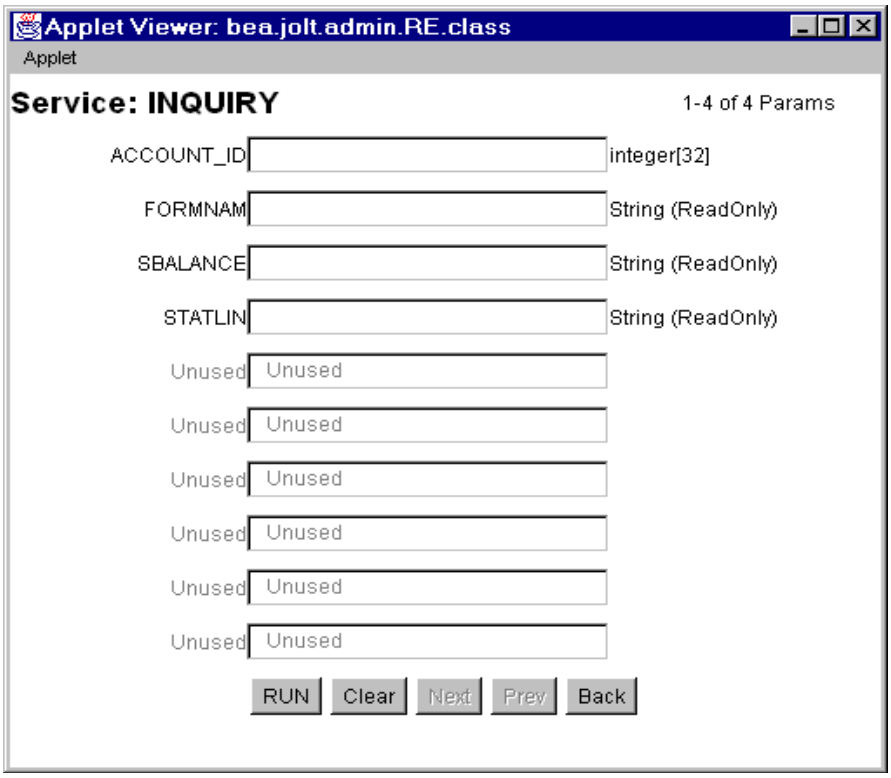
Use the Run button to test the service to ensure that the parameter information is accurate. A service can only be tested when the corresponding BEA Tuxedo server is running for the service being tested.

Although the **Test** button in the Edit Services window is enabled when parameters are not added to the service, the Service Test window displays unused in the parameter fields, and they are disabled. Refer to "Sample Service Test Window" on page 4-40 for an example of unused parameter fields.

Note: The Service Test window displays up to 20 items of any multiple-occurrence parameters. All items that follow the twentieth occurrence of a parameter cannot be tested.

The following figure is an example of a Service Test window with both writable and read-only text fields.

Figure 4-19 Sample Service Test Window



Service Test Window Description

The following table describes the Service Test window options.

Note: You can enter a two-digit hexadecimal character (0-9, a-f, A-F) for each byte in the CARRAY data field. For example, the hexadecimal value for 1234 decimal is 0422.

Option	Description
Service	Displays the name of the tested service (read-only).
Parameters displayed	Tracks the parameters displayed in the window (read-only).
Parameter text fields	The parameter information text entry field. These fields are writable or read-only. Disabled if read-only.

Option	Description
RUN	Runs the test with the data entered.
Clear	Clears the text entry field.
Next	Lists additional parameter fields, if applicable.
Prev	Lists previous parameter fields, if applicable.
Back	Returns to the Edit Services window.

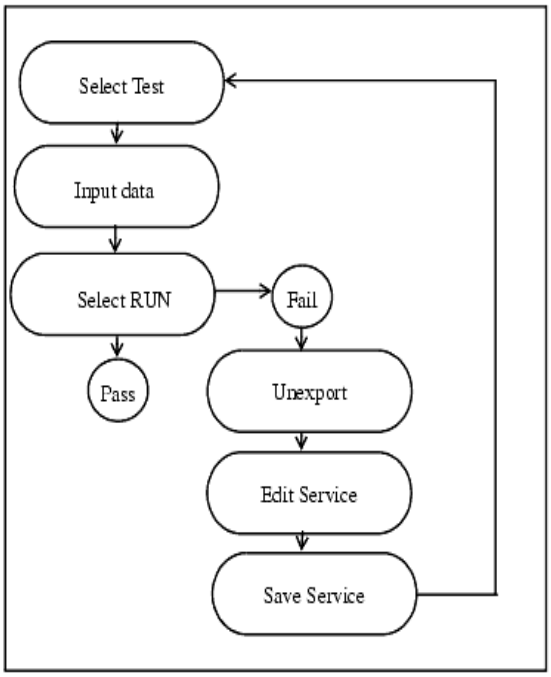
Testing a Service

You can test a service without making changes to the service or its parameters. You can also test a service after editing the service or its parameters.

Test Service Process Flow

The following figure shows a typical Repository Editor service flow test.

Figure 4-20 Test Service Flow



Instructions for Testing a Service

Follow these steps to test a service. For troubleshooting information, see the first two entries in the Repository Editor Troubleshooting table.

1. Select **Services** from the Repository Editor Logon window.

   The Services window is displayed.

2. Select the package and the service to test.
3. Click **Edit** to access the Edit Services window.
4. Click **Test** to access the Service Test window.

5. Enter data in the Service test window parameter text fields.
6. Click **RUN**.

   The status line displays the outcome as follows:

   – If the test passed: "`Run Completed OK`"

   – If the test failed: "`Call Failed`"

   See "Repository Editor Troubleshooting" on page 4-43 for additional Repository Editor troubleshooting information.

If Edits are Required After Testing

Follow these steps if editing is required to pass the test:

1. Return to the Repository Editor Logon window and click **Packages**.
2. Select the package with the services to be retested.
3. Click **Unexport**.
4. Click **Back** to return to the Repository Editor Logon window.
5. Click **Services** to display the Services window.
6. Select the package and the service that requires editing and click **Edit**.
7. In the Edit Services window, edit the service.
8. Save the service, click Test, and repeat steps 5 and 6 of the "Instructions for Testing a Service" section.

Repository Editor Troubleshooting

Consult the following table if you encounter problems while using the Repository Editor.

Table 4-5 Repository Editor Troubleshooting

If...	Then...
A parameter is incorrect	Edit the service.
The Jolt server is down	Check the server. The BEA Tuxedo service is down. You do not need to edit the service.

Table 4-5 Repository Editor Troubleshooting (Continued)

If...	Then...
You receive any error	Make sure the browser you are running is Java-enabled: • For Netscape browsers, make sure that Enable Java and Enable JavaScript are checked under Edit→Preferences→Advanced. Then select Communicator→Tools→Java Console. If the Java Console does not exist on the menu, the browser probably does not support Java. • For Internet Explorer, make sure the version is 3.0 (or later). • If running Netscape Navigator, check the Java Console for error messages. • If running `appletviewer`, check the system console (or the window where you started the `appletviewer`).
You cannot connect to the Jolt Server (after entering **Server** and **Port Number**)	Make sure that: • Your Server name is correct (and accessible from your machine). Check that the port number is the correct port. A JSL or JRLY must be configured to listen on that port. • The Jolt Server is up and running. If any authentication is enabled, check that you are entering the correct usernames and passwords. • If the applet was loaded through HTTP, make sure that the Web server, JRLY, and the Jolt server are on the same machine. (To do this, enter the server name into the Repository Editor that refers to the same machine name as the one used in the URL to download the applet).

Table 4-5 Repository Editor Troubleshooting (Continued)

If...	Then...
You cannot start the Repository Editor	If you are running the editor in a browser and downloading the Repository Editor applet through HTTP, make sure that: • The browser is Java-enabled. • The Web server is running and accessible. • The `RE.html` file is available to the Web server. • The `RE.html` file contains the correct `<codebase>` parameter. Codebase identifies where the Jolt class files are located. If running the editor in a browser (or `appletviewer`) and loading the applet from disk, make sure that: • The browser is Java-enabled. • The `RE.html` file exists and is readable. • The `RE.html` file is Java-enabled. • The `RE.html` file contains the correct `<codebase>` parameter (this is where the Jolt class files are installed on the local disk). • CLASSPATH is set and points to the Jolt class directory.
You cannot display **Packages** or **Services** even though you are sure they exist	• Make sure that the Jolt Repository Server is running (JREPSVR). • Make sure that the JREPSVR can access the repository file. • Make sure that the configuration of JREPSVR: verify CLOPT parameters and verify that `jrep.f16` (FML definition file) is installed and accessible (follow installation documentation).

Table 4-5 Repository Editor Troubleshooting (Continued)

If...	Then...
You cannot save changes in the Repository Editor	Check permissions on the repository file. The file must be writable by the user who starts JREPSVR.
You cannot test services	• Check that the service is available. • Verify the service definition matches the service. • If BEA Tuxedo authentication is enabled, check that you have the required permissions to execute the service. • Check if the application file (FML or VIEW) is specified correctly in the variables (FIELDTBLS or VIEWFILES) in the ENVFILE. All applications FML field tables or VIEW files must be specified in the FIELDTBLS and VIEWFILES environment variables in the ENVFILE. If these files are not specified, the JSH cannot process data conversion and you receive the message “ServiceException: TPEJOLT data conversion failed.” • Check the ULOG file for any additional diagnostic messages.

CHAPTER 5

Using the Jolt Class Library

The BEA Jolt Class Library provides developers with a set of object-oriented Java language classes for accessing BEA 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 BEA Tuxedo services from Java applets.

This topic includes the following sections:

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 BEA 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 5-11.

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 or 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

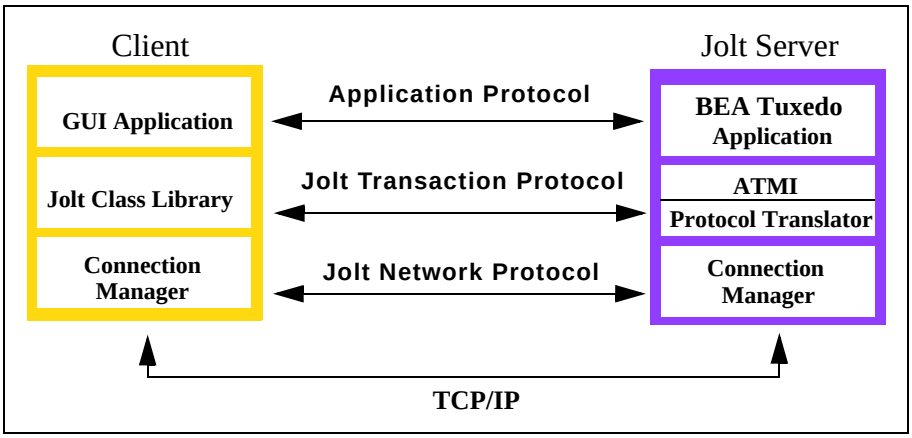
The Jolt Class Library returns both Jolt interpreter and BEA Tuxedo errors as exceptions. The Jolt Class Library Reference contains the Jolt classes and lists the errors or exceptions thrown for each class. The *BEA Jolt API Reference* contains the Error and Exception Class Reference.

Jolt Client/Server Relationship

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

The following figure illustrates the client/server relationship between a Jolt program and the Jolt Server.

Figure 5-1 Jolt Client/Server Relationship



As illustrated in the figure, the Jolt Server acts as a proxy for a native BEA Tuxedo client, implementing functionality available through the native BEA Tuxedo client. The BEA Jolt Server accepts requests from BEA Jolt clients and maps those requests into BEA Tuxedo service requests through the BEA Tuxedo ATMI interface. Requests and associated parameters are packaged into a message buffer and delivered over the network to the BEA Jolt Server. The BEA Jolt Connection Manager handles all communication between the BEA Jolt Server and the BEA Jolt applet using the BEA Jolt Transaction Protocol. The BEA 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 BEA Tuxedo as specified by the message.

Once a service request enters the BEA Tuxedo system, it is executed in exactly the same manner as any other BEA Tuxedo request. The results are returned through the ATMI interface to the BEA Jolt Server, which packages the results and any error information into a message that is sent

to the BEA Jolt client applet. The BEA Jolt client then maps the contents of the message into the various BEA 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.

The following table describes the client-side requests and Jolt Server-side actions in a simple example program.

Table 5-1 Jolt Client/Server Interaction

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

The following tasks summarize the interaction shown in the previous table, "Jolt Client/Server Interaction."

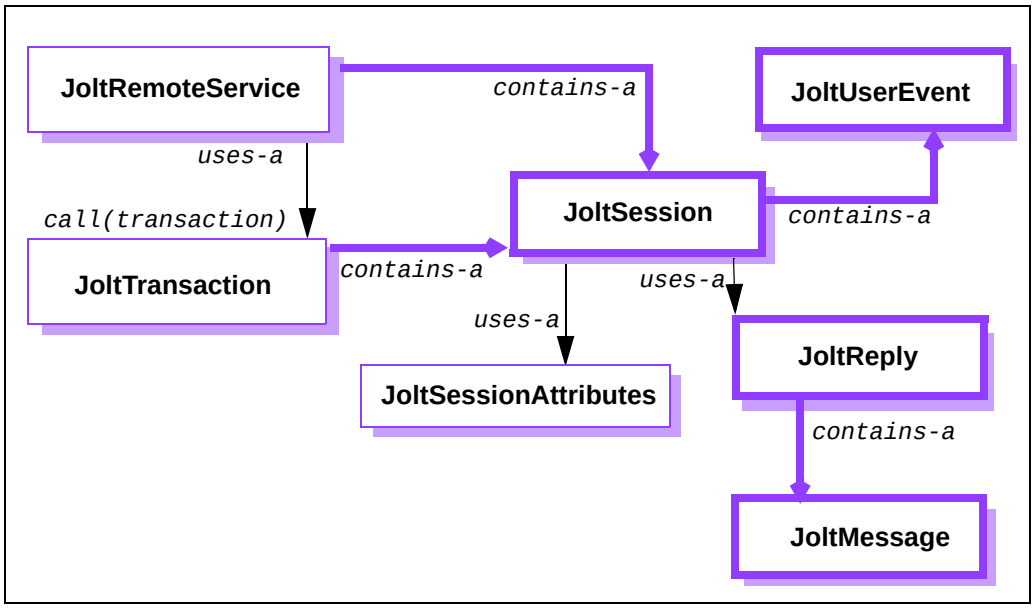
1. Bind the client to the BEA Tuxedo environment using the `JoltSessionAttributes` class.
2. Establish a session.
3. Set variables.
4. Perform the necessary transaction processing.
5. Log the client off of the BEA Tuxedo system.

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 5-7 describes the Jolt Class Library classes in more detail.

Jolt Object Relationships

The following figure illustrates the relationship between the instantiated objects of the Jolt Class Library classes.

Figure 5-2 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 BEA Tuxedo environment prior to initiating any transaction activity. The Jolt Class Library provides the JoltSessionAttributes class and JoltSession class to establish a connection to a BEA Tuxedo system.

The JoltSessionAttributes class will contain the connection properties of Jolt and BEA 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 BEA Tuxedo object, the Jolt and BEA 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 BEA 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 5-42 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 5-11 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 BEA 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 5-11:

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 Jolt Repository, to the withdrawal and deposit objects, respectively. The services WITHDRAWAL and DEPOSIT must be defined in the Jolt 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();
   ```

The following listing shows an example of a simple application for the transfer of funds using the Jolt classes.

Listing 5-1 Jolt Transfer of Funds Example (SimXfer.java)

```
/* Copyright 1999 BEA 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 BEA Tuxedo Buffer Types with Jolt

Jolt supports the following built-in BEA 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 BEA 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 BEA 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 BEA Tuxedo typed buffers, data types, and buffer types, refer to the following documents:

- *Programming BEA Tuxedo ATMI Applications Using C*
- *BEA Tuxedo ATMI C Function Reference*
- *BEA 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 BEA 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 5-18 illustrates how Jolt works with a service whose buffer type is STRING. The `ToUpper` BEA Tuxedo Service is available in the BEA Tuxedo `simpapp` example.

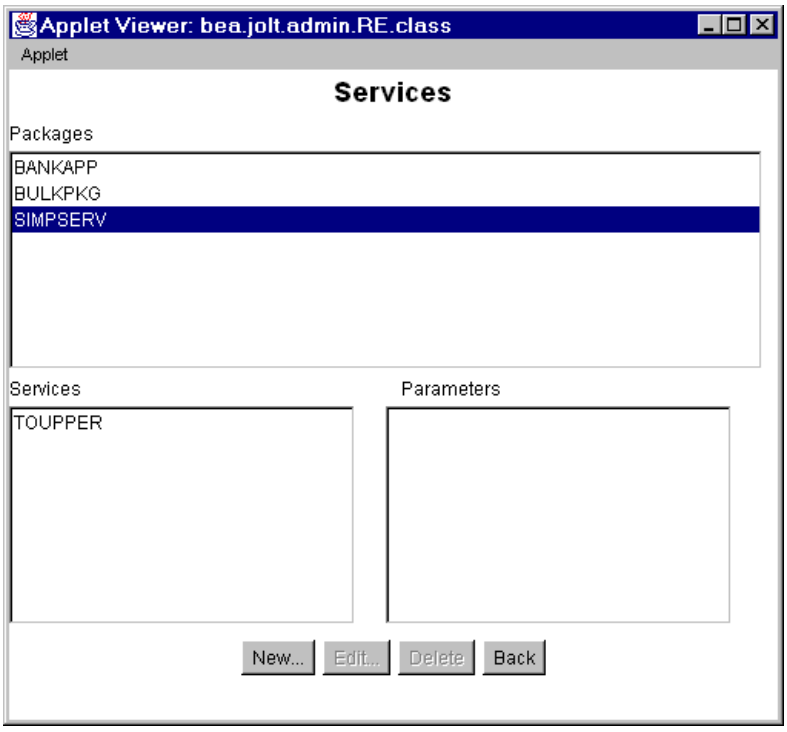
Define TOUPPER in the Repository Editor

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

Note: Refer to "Using the Jolt Repository Editor" on page 4-1 for more information about defining your services and adding new parameters.

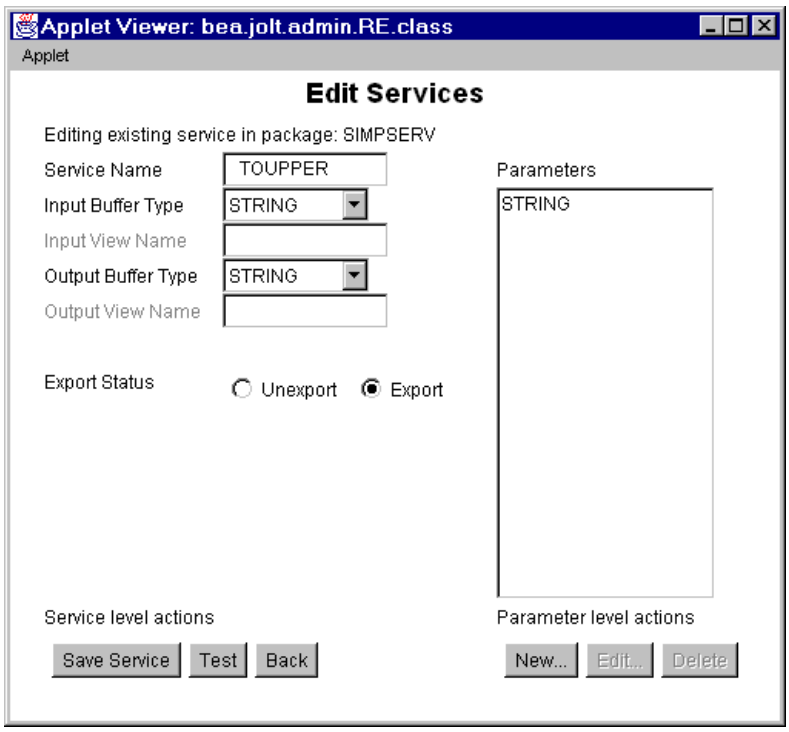
1. In the Jolt Repository Editor Logon window, click **Services**.

Figure 5-3 Add a TOUPPER Service



2. In the Services window, select the **TOUPPER** service in the SIMPSERV package.
3. Click **Edit**.

Figure 5-4 Set Input and Output Buffer Types to STRING



4. In the Edit Services window, define an input buffer type of STRING and an output buffer type of STRING. Refer to the figure "Set Input and Output Buffer Types to STRING" on page 5-17.)
5. For the TOUPPER service, define only one parameter named STRING, which is both an input and an output parameter.
6. Click **Save Service**.

ToUpper.java Client Code

The `ToUpper.java` Java code fragment in the following listing 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 BEA 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 5-2 Use of the STRING Buffer Type (ToUpper.java)

```
/* Copyright 1996 BEA 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 BEA 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 a BEA 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 BEA 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.

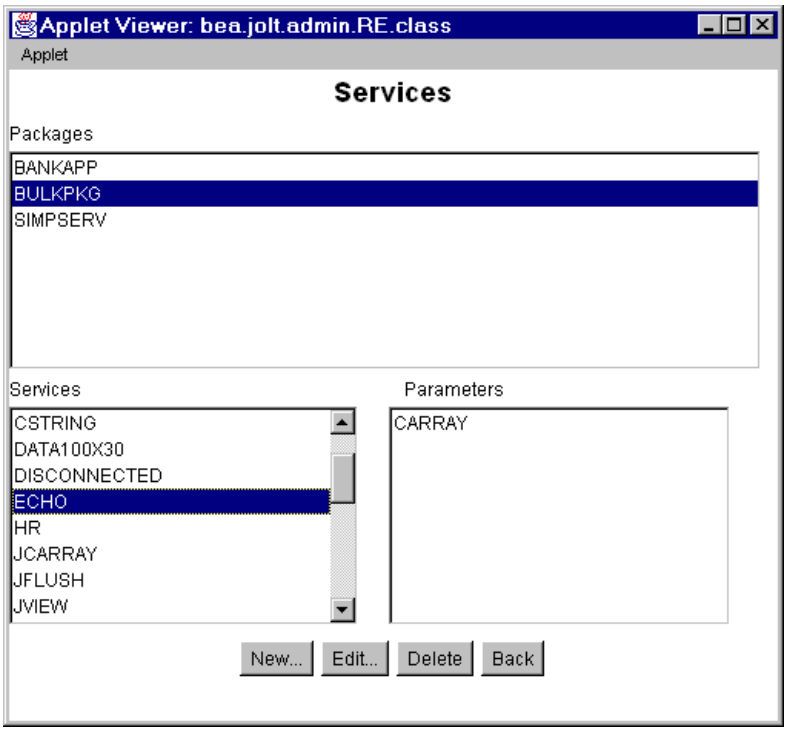
Note: X_OCTET is used identically to CARRAY.

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 Jolt Repository Editor.

Note: Refer to "Using the Jolt Repository Editor" on page 4-1 for more information about defining your services and adding new parameters.

Figure 5-5 Repository Editor: Add the ECHO Service

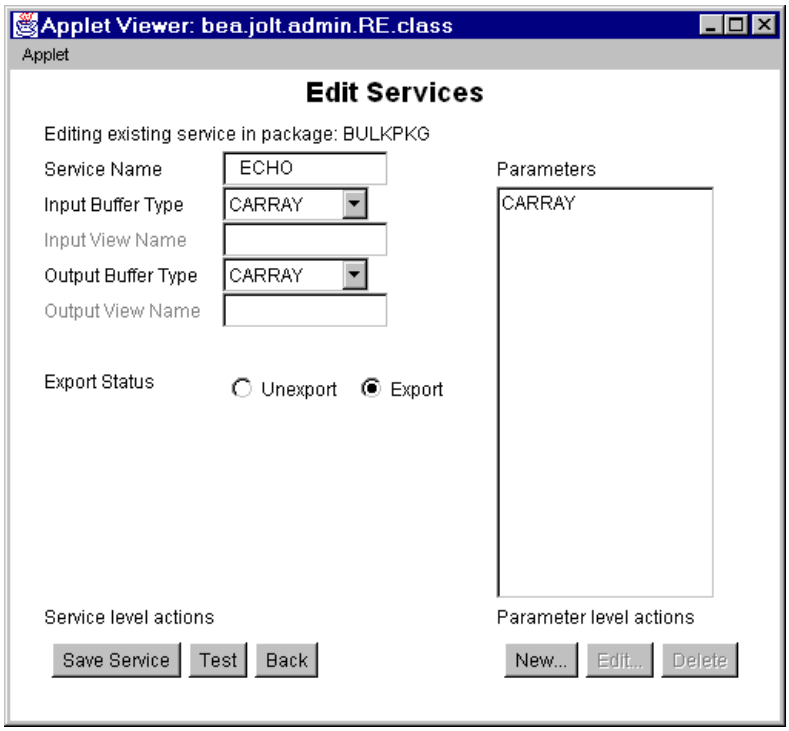


Use the Repository Editor to add the ECHO service as follows:

1. In the Repository Editor, add a service called ECHO.
2. Define the input buffer type and output buffer type as **CARRAY**.
3. Define only one parameter named CARRAY, which is both an input and output parameter.

Note: If using the X_OCTET buffer type, you must change the Input Buffer Type and Output Buffer Type fields to X_OCTET.

Figure 5-6 Repository Editor: Edit ECHO Service



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 the following listing assumes that a session object was already instantiated.

Listing 5-3 CARRAY Buffer Type Example

```
/* Copyright 1996 BEA 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 5-24 is a Jolt client that contains a PASSFML service.
- The "tryOnFml.f16 Field Definitions" on page 5-25 is a BEA Tuxedo FML field definitions table used by the PASSFML service.
- The "tryOnFml.c Code Example" on page 5-27 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 the following listing 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 5-4 tryOnFml.java Code Example

```
/* Copyright 1997 BEA 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 5-5 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

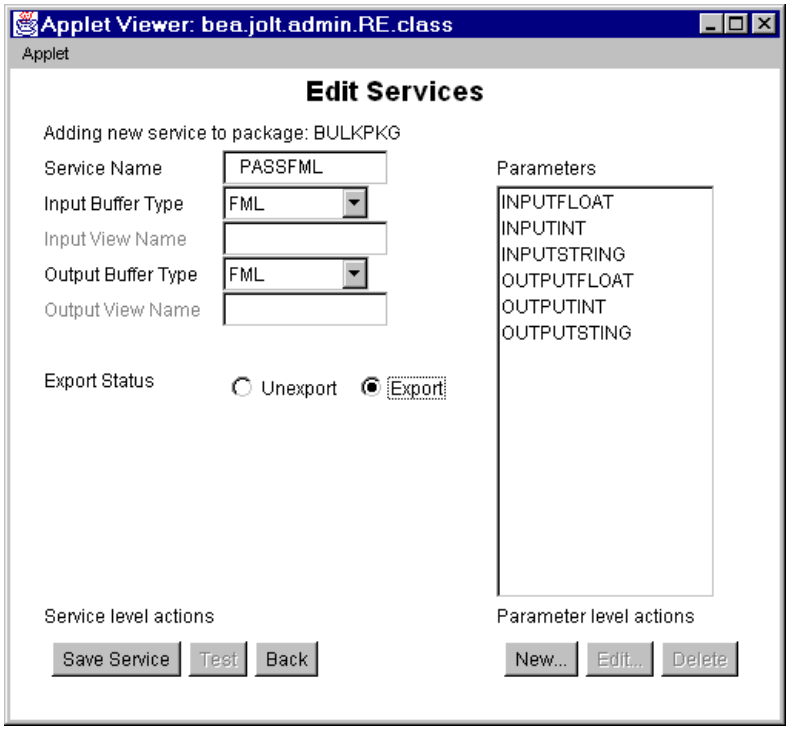
The BULKPKG package contains the PASSFML service, which is used with the `tryOnFml.java` and `tryOnFml.c` code. Before running the `tryOnFml.java` example, you need to modify the PASSFML service through the Jolt Repository Editor.

Note: Refer to "Using the Jolt Repository Editor" on page 4-1 for more information about defining a service.

1. In the Edit Services window of the Jolt Repository Editor, define the PASSFML service with an input buffer type of FML and an output buffer type of FML.

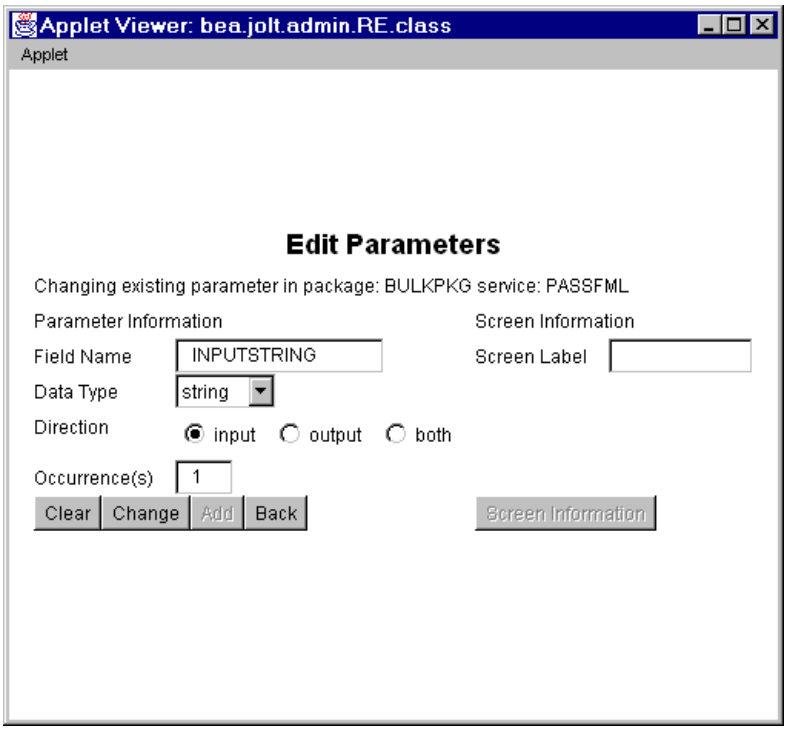
   The figure "Repository Editor Window: Edit Services (PASSFML)" on page 5-26 illustrates the PASSFML service, and Input Buffer and Output Buffer of FML.

Figure 5-7 Repository Editor Window: Edit Services (PASSFML)



2. Select the input buffer type and output buffer type as **FML** for the PASSFML service.
3. Click **Edit** to display the Edit Parameters window as shown in the following figure.

Figure 5-8 Edit the PASSFML Parameters



4. Define the parameter for the PASSFML service.
5. Repeat steps 2-4 for each parameter in the PASSFML service.

tryOnFml.c Server Code

The following listing 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 5-6 tryOnFml.c Code Example

```
/*

 * tryOnFml.c
 *
```

```
 * Copyright (c) 1997 BEA Systems, Inc. All rights reserved
 *
 * Contains the PASSFML BEA 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 )
{
       FLDLEN len;
       FBFR   *svcinfo = (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(Ferror));
              tpreturn(TPFAIL, 0, rqst->data, 0L, 0);
       }
       if (Fget(svcinfo, INPUTINT, 0, (char *) &inputInt, &len) < 0) {
              (void)userlog("Fget of INPUTINT failed %s",Fstrerror(Ferror));
              tpreturn(TPFAIL, 0, rqst->data, 0L, 0);
       }
       if (Fget(svcinfo, INPUTFLOAT, 0, (char *) &inputFloat, &len) < 0) {
              (void)userlog("Fget of INPUTFLOAT failed %s",
                     Fstrerror(Ferror));
              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 *)tpalloc("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, (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, 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 BEA Tuxedo typed buffer. The VIEW buffer provides a way to use C structures and COBOL records with the BEA Tuxedo system. The VIEW typed buffer enables the BEA 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 BEA 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 5-33 is the Jolt client that contains the code used to connect to BEA Tuxedo and uses the VIEW buffer type.
- The listing “simpview.v16 Field Definitions” on page 5-34 contains the BEA Tuxedo VIEW field definitions.
- The “simpview.c Code Example” on page 5-34 contains the server side C code for handling the input from the Jolt client.

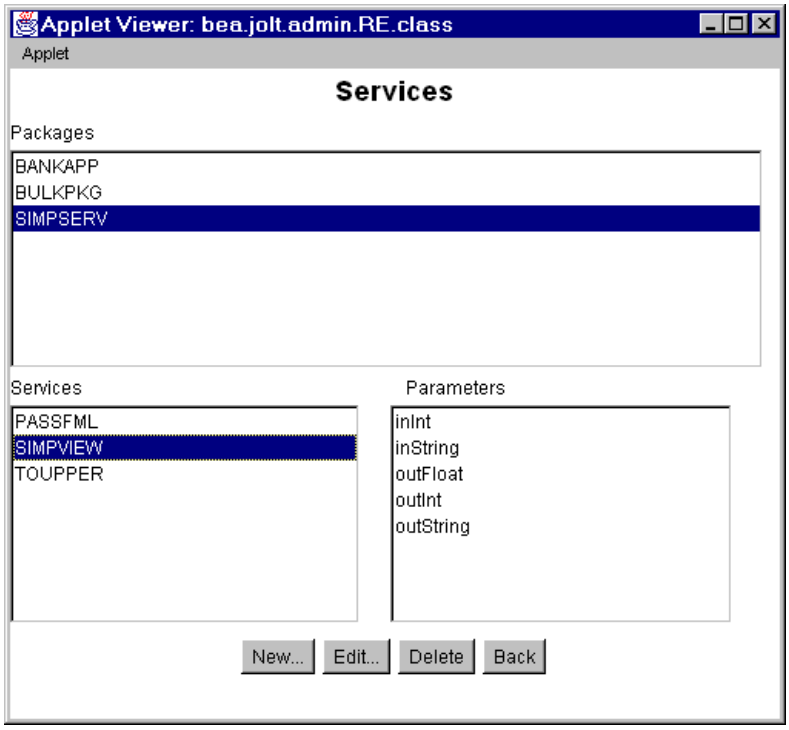
The Jolt client treats a null character in a VIEW buffer string format as an end-of-line character and truncates any part of the string that follows the null.

Define VIEW in the Repository Editor

Before running the `simpview.java` and `simpview.c` examples, you need to define the SIMPVIEW service through the Jolt Repository Editor.

Note: Refer to “Using the Jolt Repository Editor” on page 4-1 for more information about defining a service.

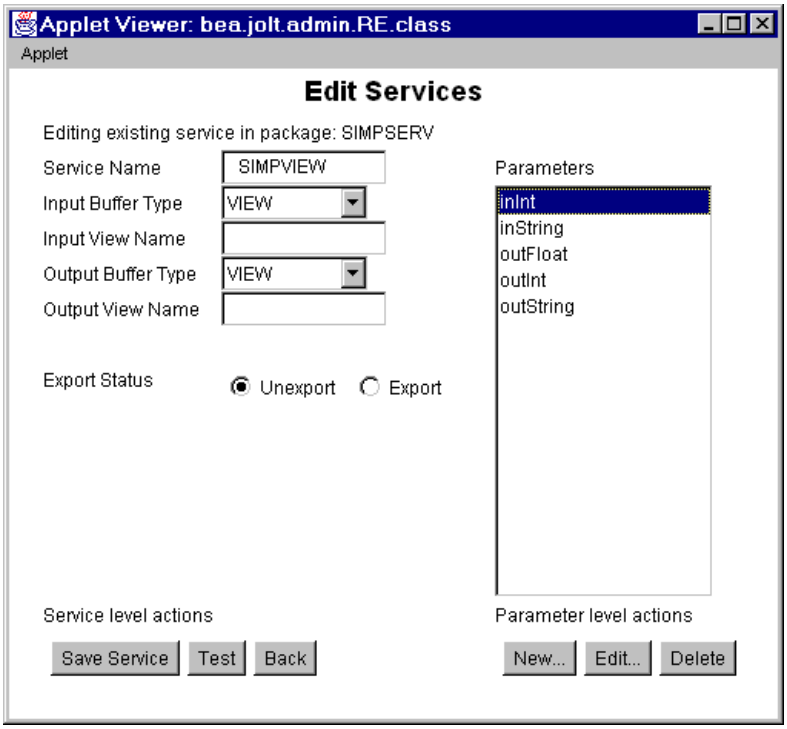
Figure 5-9 Repository Editor: Add SIMPVIEW Service



In the Repository Editor add the VIEW service as follows:

1. Add a SIMPVIEW service for the SIMPSERV package.
2. Define the SIMPVIEW service with an input buffer type of VIEW and an output buffer type of VIEW.

Figure 5-10 Repository Editor: Edit SIMPVIEW Service



3. Define the parameters for the VIEW service. In this example the parameters are: `inInt`, `inString`, `outFloat`, `outInt`, `outString`.

Note: If using the X_COMMON or X_C_TYPE buffer types, you must put the correct buffer type in the Input Buffer Type and Output Buffer Type fields. Additionally, you must choose the corresponding Input View Name and Output View Name fields.

simpview.java Client Code

The listing "simpview.java Code Example" on page 5-33 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 Jolt Repository Editor and write the `simpview.c` BEA 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 5-7 simpview.java Code Example

```
/* Copyright 1997 BEA 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 BEA 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 BEA Tuxedo VIEW field definitions for the `simpview.java` example that were shown in the previous listing.

Listing 5-8 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         fbname 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 the following listing, the input and output buffers are VIEW. The code accepts the VIEW buffer data as input and outputs the same data as VIEW.

Listing 5-9 simpview.c Code Example

```
/*
 * SIMPVIEW.c
 *
 * Copyright (c) 1997 BEA Systems, Inc.  All rights reserved
 *
 * Contains the SIMPVIEW BEA 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 (VIEWSVC) from the TPSVCINFO structure
        */
       struct SimpView*svcinfo = (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 TPFAIL if the input
        * is not correct.
        */
       (void)userlog("SIMPVIEW: InString=%s,InInt=%d,InFloat=%f",
              svcinfo->inString, svcinfo->inInt, svcinfo->inFloat);
       /*
```

```
         * Populate the output fields and send them back to the caller
         */

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

Using the XML Buffer Type

The XML buffer type enables BEA Tuxedo applications to use XML documents for exchanging data within and between applications. BEA 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.

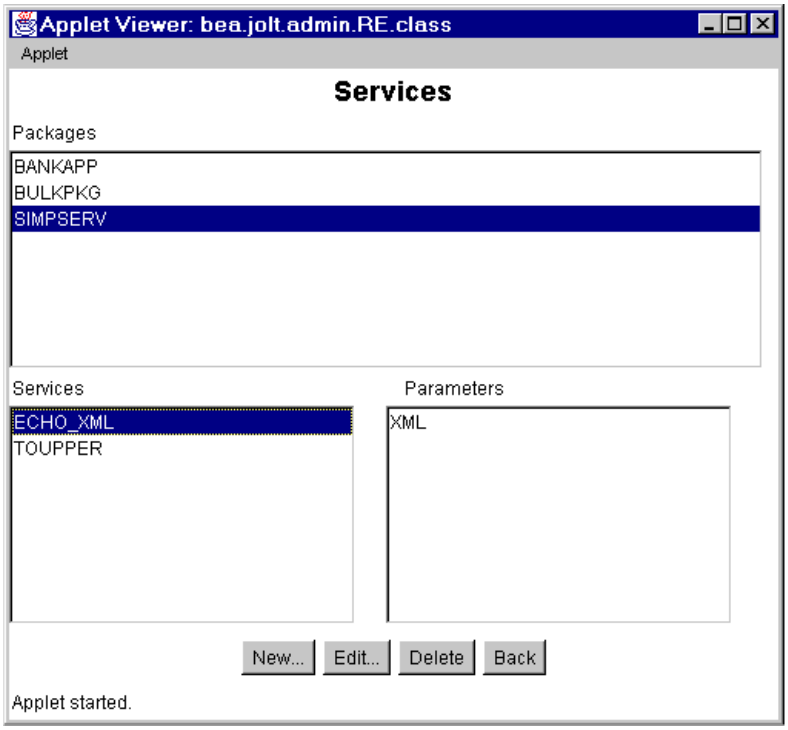
Note: Similar to CARRAY, the XML buffer type is treated as a byte arrary, not a STRING. Therefore, no data conversion takes place between a Jolt client and a BEA Tuxedo service.

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 Jolt Repository Editor.

Note: Refer to "Using the Jolt Repository Editor" on page 4-1 for more information about defining your services and adding new parameters.

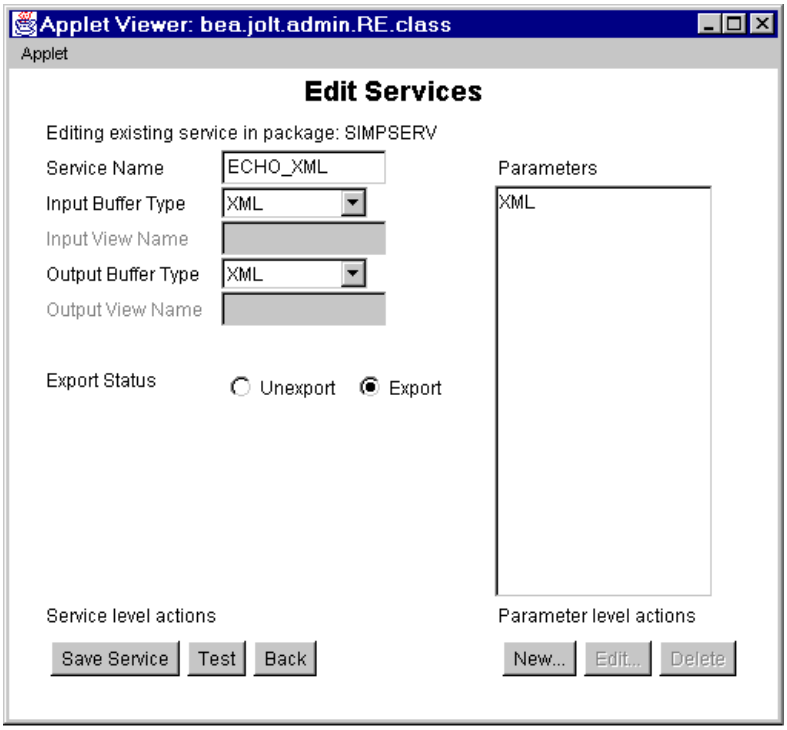
Figure 5-11 Repository Editor: Add the XML Service



Use the Repository Editor to add the XML service as follows:

1. In the Repository Editor, add a service called ECHO_XML.
2. For the ECHO_XML service, define the input buffer type and output buffer type as XML.
3. Define the ECHO_XML service with only one parameter named **XML**, which is both an input and output parameter.

Figure 5-12 Repository Editor: Edit the XML Service



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 the following listing assumes that a session object was already instantiated.

Listing 5-10 XML Buffer Type Example

```
/* Copyright 2001 BEA 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 Jolt 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 BEA 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 BEA 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 a BEA Tuxedo service function and then

immediately start another thread that sends another request to a BEA 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

The following listing illustrates the use of two threads in a Jolt application.

Listing 5-11 Using Multiple Threads with Jolt (ThreadBank.java)

```
/* Copyright 1996 BEA 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 (dattr.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;
}
        dattr.setInt(dattr.IDLETIMEOUT, 60);
        session = new JoltSession (dattr, 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 BEA Tuxedo Services or other BEA Tuxedo clients. The Jolt Class Library contains classes that support the following types of BEA Tuxedo notifications for handling event-based communication:

- **Unsolicited Event Notifications**—these are notifications that a Jolt client receives as a result of a BEA 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 BEA Tuxedo Event Broker. The notifications are only received when the Jolt client subscribes to an event *and* any BEA 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 BEA 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 BEA 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 BEA 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 BEA 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 BEA 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 Jolt 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 BEA 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.

BEA Tuxedo Event Subscription

BEA Tuxedo brokered event notification allows BEA 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 BEA Tuxedo events that are broadcast or posted using the BEA Tuxedo `tpnotify()` or `tpbroadcast()` calls.

Jolt clients can only subscribe to events and notifications that are generated by other components in BEA Tuxedo (such as a BEA 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 BEA Tuxedo. In BEA 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 BEA 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 5-12 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 5-13 Jolt Object Reuse (reuseSample.java)

```
/* Copyright 1999 BEA 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 BEA Tuxedo BANKAPP funds TRANSFER service.

The following listing 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 5-14 Extending Jolt Remote Service (extendSample.java)

```
/* Copyright 1999 BEA 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 BEA 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 BEA 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 BEA Tuxedo server and Jolt server to work with Jolt is available in *Installing the BEA 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 BEA 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 5.0
- Sun JSSE implementation bundled in HP JRE 5.0
- IBM JSSE implementation bundled in IBM JRE 5.0

The following listing is the sample Jolt client code to make it possible to use SSL when communicate with `JSL`/`JSH`.

Listing 5-15 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 optionly 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;
    }
}
```

CHAPTER 6

Using JoltBeans

Formerly available as an add on, JoltBeans are included in BEA 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 BEA Jolt. You can develop a fully functional BEA Jolt client without writing any code.

This topic includes the following sections:

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 BEA Jolt Repository, providing easy access to available BEA Tuxedo services.

Note: Currently, Symantec Visual Café 3.0 is the only IDE that is certified by BEA 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 BEA 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 BEA Tuxedo Events

JavaBeans communicate through events. An event in a BEA Tuxedo system is different from an event in a JavaBeans environment. In a BEA 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 BEA Tuxedo Event Subscription and Notification with JoltBeans

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

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

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

1. Use the `setEventName()` and `setFilter()` methods of the JoltUserEventBean to specify the BEA 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 BEA Tuxedo event notification arrives, JoltSessionBean sends a JoltOutputEvent to its listeners by calling `serviceReturned()` on them. The JoltOutputEvent object contains the data of the BEA 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.

How JoltBeans Use JavaBeans Events

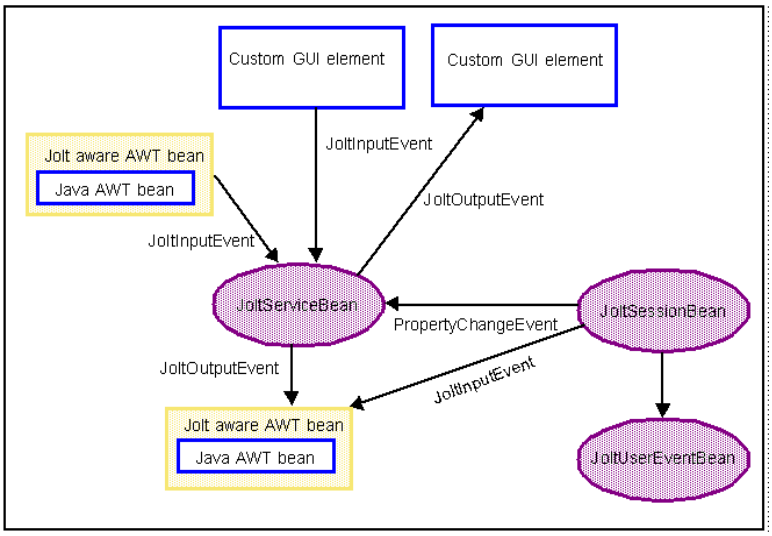
A Jolt client applet or application that is built using JoltBeans typically consists of Jolt-aware GUI beans, such as JoltTextField or JoltList, and JoltBeans, such as JoltServiceBean and JoltSessionBean. The main mode of communication between Beans is by JavaBeans events.

Jolt-aware beans are sources of JoltInputEvents or listeners of JoltOutputEvents or both. JoltServiceBeans are sources of JoltOutputEvents and listeners of JoltInputEvents.

The Jolt-aware GUI Beans expose properties and methods so you can link the beans directly to the parameters of a BEA Tuxedo service (represented by a JoltServiceBean). Jolt-aware beans notify the JoltServiceBean via a JoltInputEvent when their content changes. The JoltServiceBean sends a JoltOutputEvent to all registered Jolt-aware beans when the reply data is available after the service call. The Jolt-aware GUI Beans contain logic that updates their contents with the corresponding output parameter of the service.

The following figure represents the possible relationships among the JoltBeans.

Figure 6-1 Possible Interrelationships Among JoltBeans



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 BEA 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 BEA 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 a BEA Tuxedo username, as well as methods to open and close a BEA Tuxedo session.

The JoltSessionBean sends a PropertyChange event when the BEA 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()`.

The following table shows the JoltSessionBean properties and descriptions.

Table 6-1 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 `//host:port number` (for example, `myhost:7000`).
AppPassword	Set the BEA Tuxedo application password used at logon, if required.
IdleTimeOut	Set the IDLETIMEOUT value.
inTransaction	Indicate `true` or `false` depending if a transaction has been started and not committed or aborted.
LoggedOn	Indicate `true` or `false` if a BEA 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 BEA Tuxedo username, if required.
UserPassword	Indicate the BEA Tuxedo user password, if required.
UserRole	Indicate the BEA Tuxedo user role, if required.

JoltServiceBean

The JoltServiceBean represents a remote BEA 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(…)`).

The following table shows the JoltServiceBean properties and descriptions.

Table 6-2 JoltServiceBean Properties and Descriptions

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

JoltUserEventBean

The JoltUserEventBean provides access to BEA Tuxedo events. You define the BEA 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.

The following table shows the JoltUserEventBean properties and descriptions.

Table 6-3 JoltUserEventBean Properties and Descriptions

Property	Description
EventName	Set the name of the user event represented by the bean.
Filter	Set the event filter.
Session	The JoltSessionBean associated with the bean that allows access to the BEA 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 BEA Jolt API Reference.

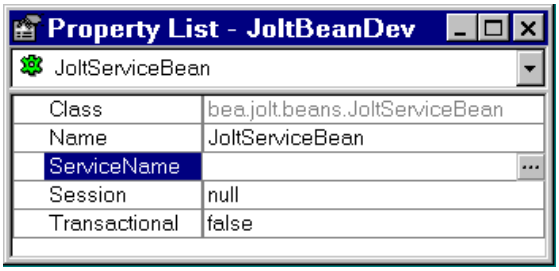
Using the Property List and the Property Editor to Modify the JoltBeans Properties

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 6-2 Property List: Ellipsis Button



When you click the ellipsis button, the Property Editor shown in the following figure is displayed.

Figure 6-3 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 Jolt 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 Jolt Repository and Setting the Property Values" on page 6-41.

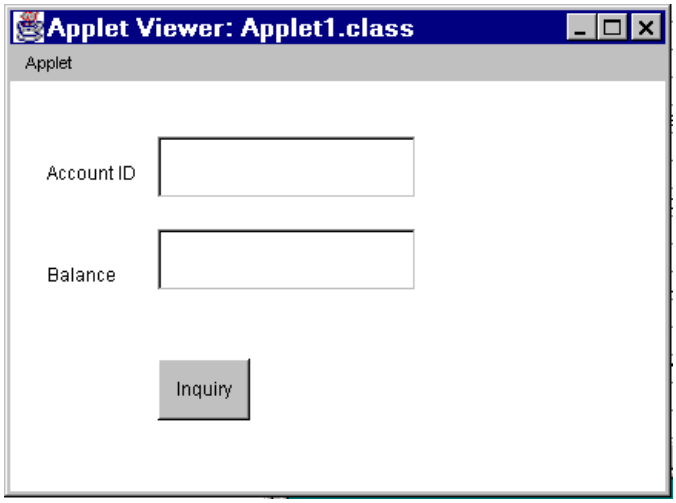
JoltBeans Class Library Walkthrough

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)

The following figure 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 BEA Tuxedo. To run this sample, the BEA Tuxedo server must be running.

Figure 6-4 Sample Inquiry Applet



Refer to the figure "Visual Café 3.0 Form Designer" on page 6-18 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 6-4 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 a BEA Tuxedo session.
JoltTextField	Gets input from the user (in this case, ACCOUNT_ID).
JoltTextField	Displays the result (in this case, SBALANCE).
JoltServiceBean	Accesses a BEA 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 a BEA 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.

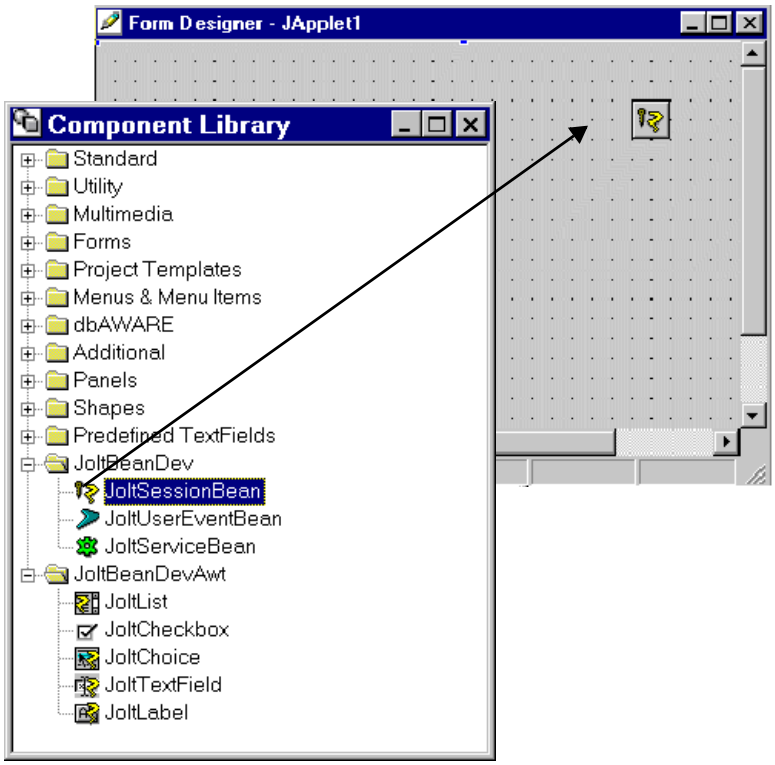
These steps are described in detail in the following sections.

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

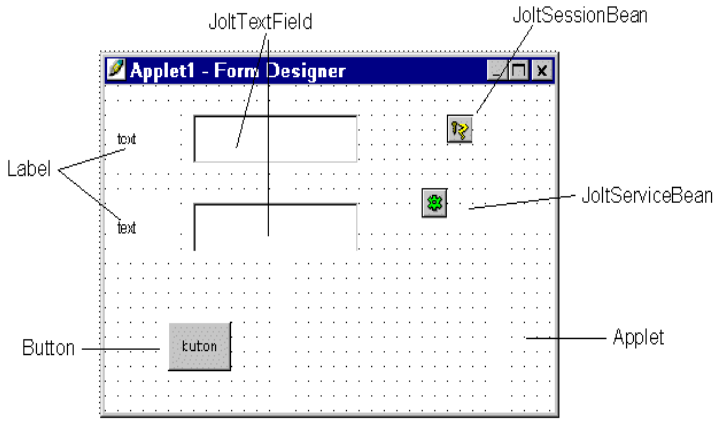
1. Choose File→New Project, and choose JFC Applet.
2. Drag and drop the beans from the Component Library (shown in the following figure) onto the palette of the form designer.

Figure 6-5 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 6-6 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 Jolt Repository and Setting the Property Values" on page 6-41.

The following table 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 6-5 Required and Recommended Property Values

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

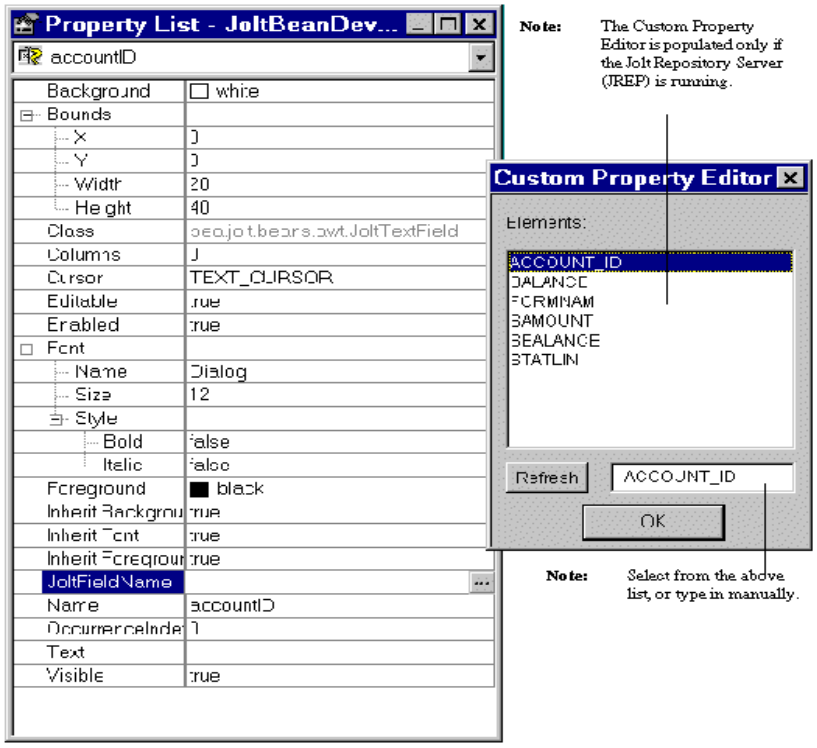
Table 6-5 Required and Recommended Property Values (Continued)

Bean	Property	Value
JoltTextField1	JoltFieldName	**ACCOUNT_ID**
JoltTextField2	Name	balance
JoltTextField2	JoltFieldName	**SBALANCE**
JoltSessionBean1	AppAddress	*//tuxserv:2010*
JoltServiceBean1	Name	inquiry
JoltServiceBean1	ServiceName	**INQUIRY**
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 Jolt Repository and Setting the Property Values" on page 6-41 for general guidelines about JoltBean properties.

Figure 6-7 Example of JoltTextField Property List and Custom Property Editor



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

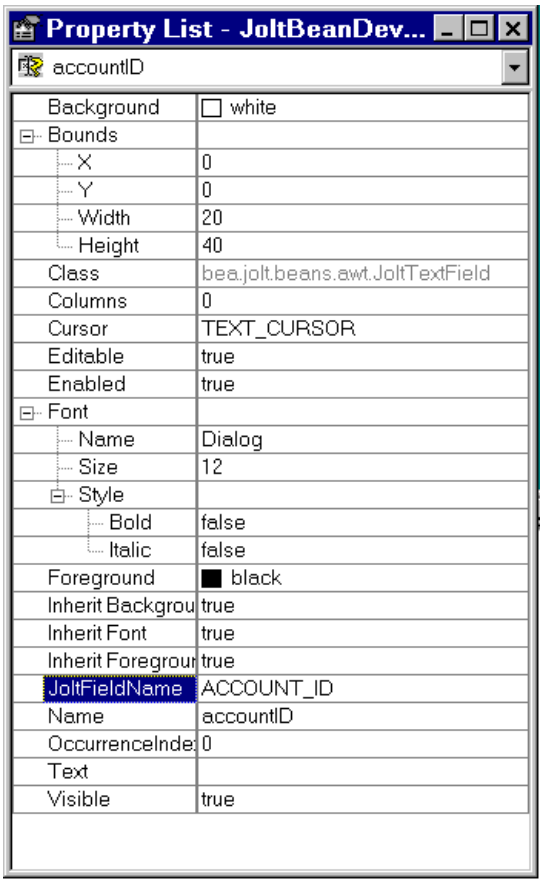
   The Custom Property Editor is displayed.

6. 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 6-21.

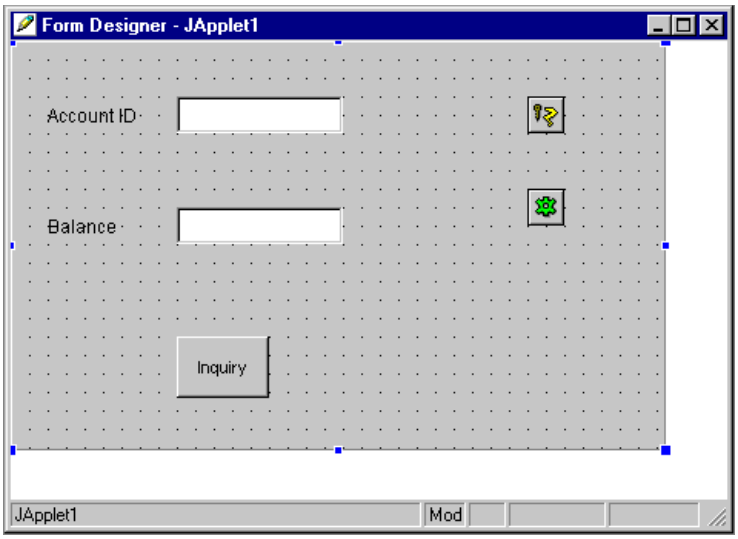
Note: The properties that are visible in the Custom Property Editor are cached locally; therefore, if the source database is modified you must use the Refresh button to see the current, available properties.

Figure 6-8 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 6-9 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 6-18 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.

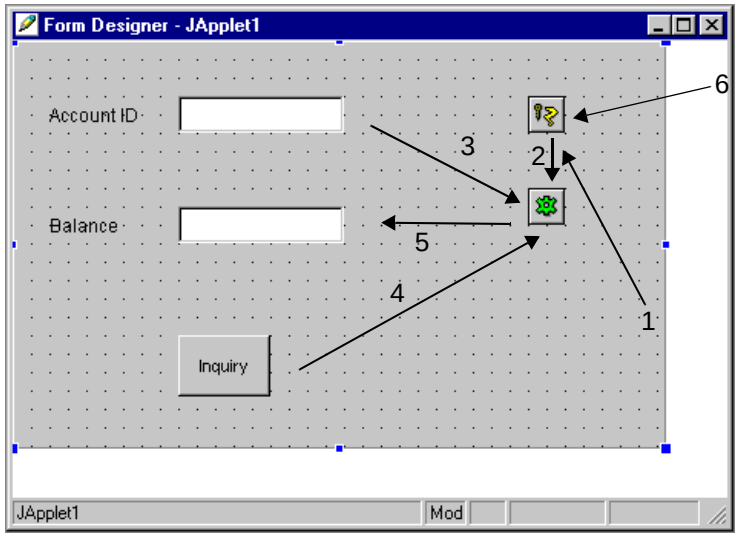
Wiring the JoltBeans Together

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.

The following figure 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 6-10 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 6-23. 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

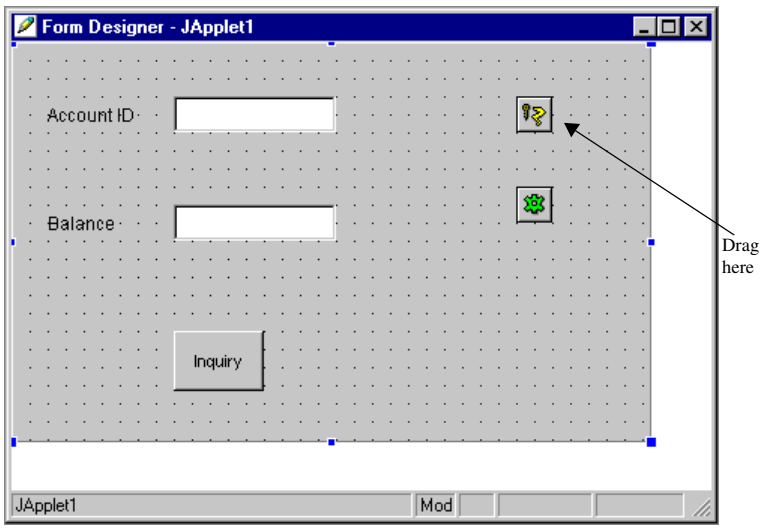
Step 6: Wire the JoltSessionBean Logoff

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 the following figure.

Figure 6-11 Wire the Applet to the Jolt Session Bean

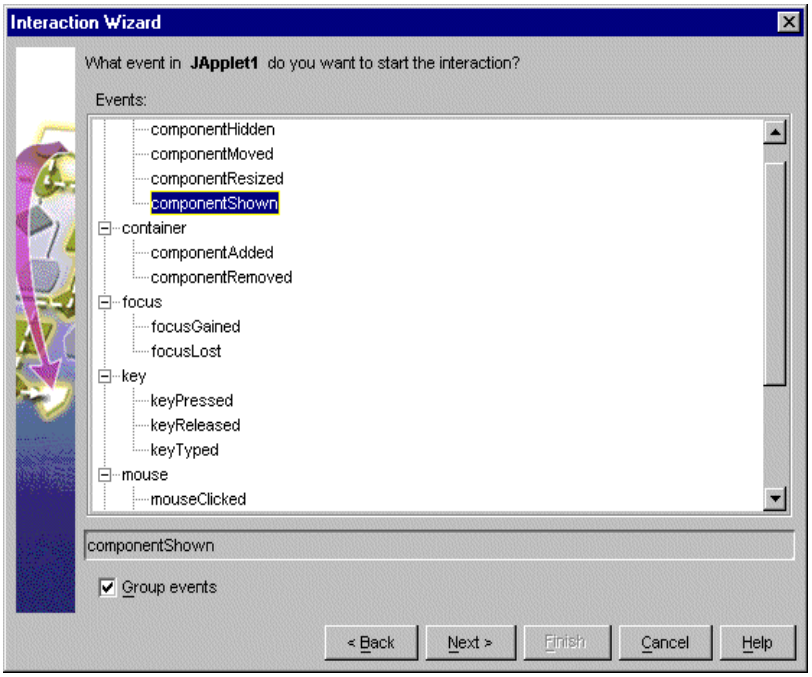


The Interaction Wizard window is displayed as shown in the figure "Select ComponentShown Event" on page 6-25, with the prompt:

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 the following figure.

Figure 6-12 Select ComponentShown Event



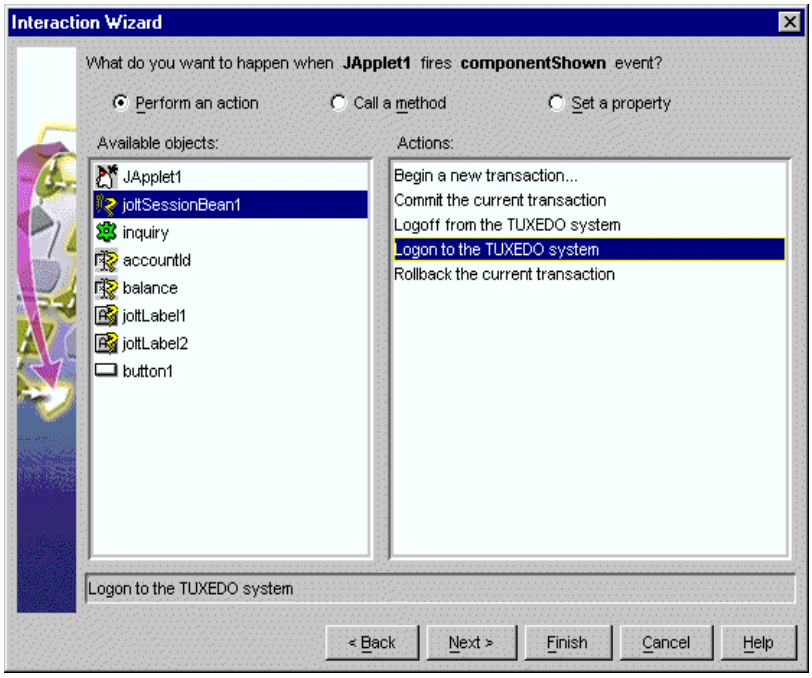
4. Click Next.

   The Interaction Wizard window is displayed, as shown in the figure "Select Logon to the Tuxedo System Action" on page 6-26, 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 the following figure.

Figure 6-13 Select Logon to the Tuxedo System Action



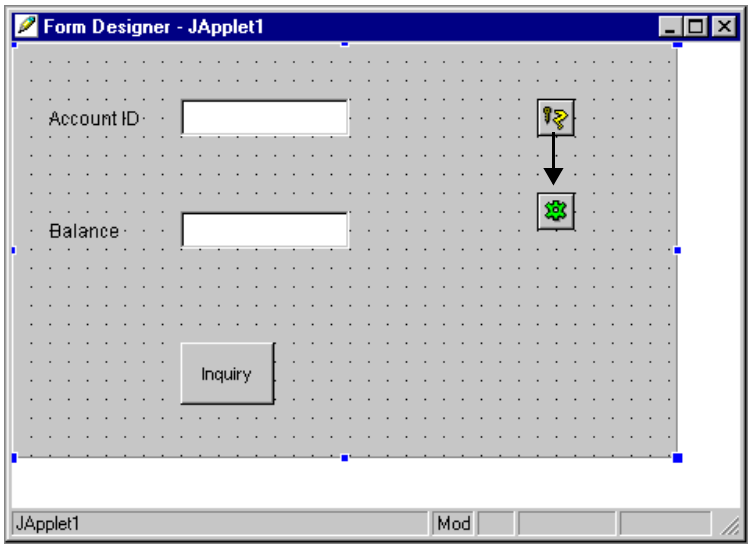
6. Click **Finish**.

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 the following figure.

Figure 6-14 Wire the JoltSessionBean to the JoltServiceBean

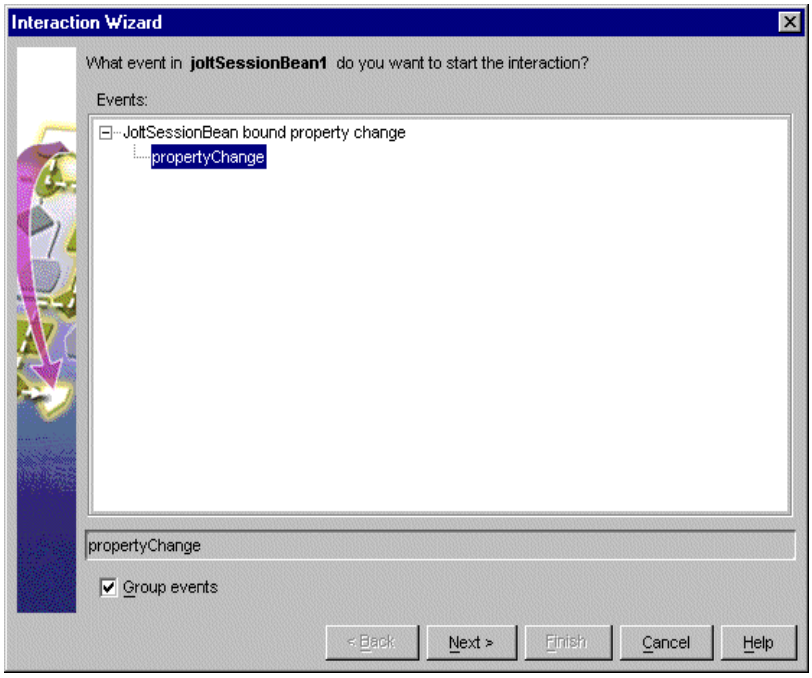


The Interaction Wizard window is displayed as shown in the figure "Select propertyChange Event" on page 6-28, with the prompt:

What event in joltSessionBean1 do you want to start the interaction?

3. Select **propertyChange** as the event that starts the interaction, as shown in the following figure.

Figure 6-15 Select propertyChange Event



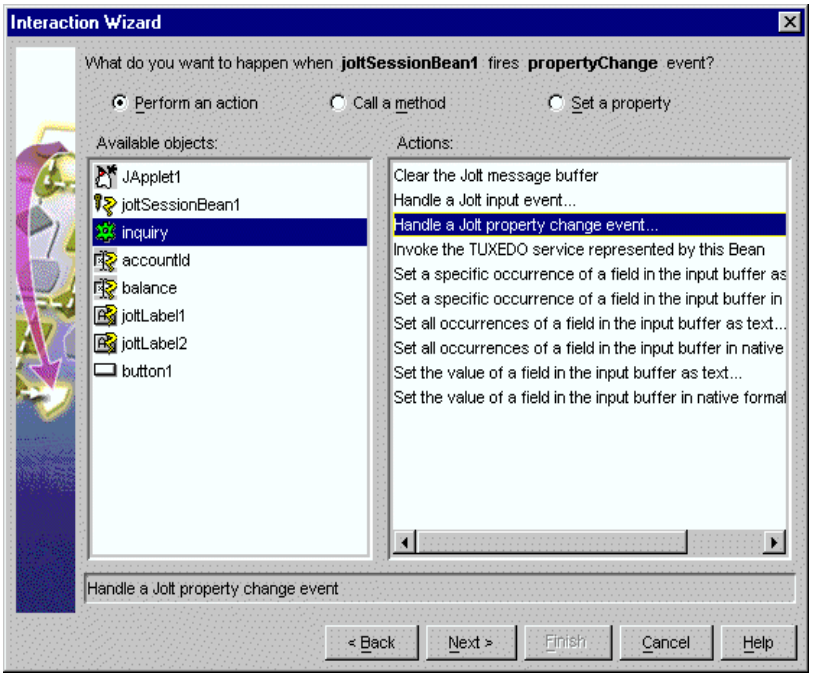
4. Click Next**.**

   The Interaction Wizard window is displayed as shown in the figure "Select Handle a Jolt property change event..." on page 6-29, 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 the following figure.

Figure 6-16 Select Handle a Jolt property change event...



6. Click **Next**.

   The Interaction Wizard window is displayed as shown in the figure "Select joltSesssionBean1" on page 6-30, with the prompt:

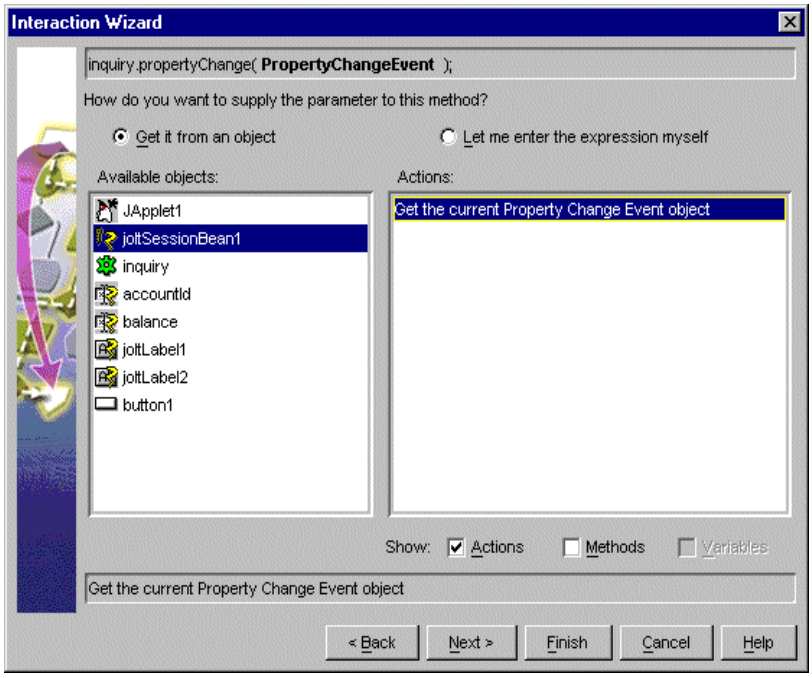
   How do you want to supply the parameter to this method?

   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 the following figure.

Figure 6-17 Select joltSesssionBean1



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

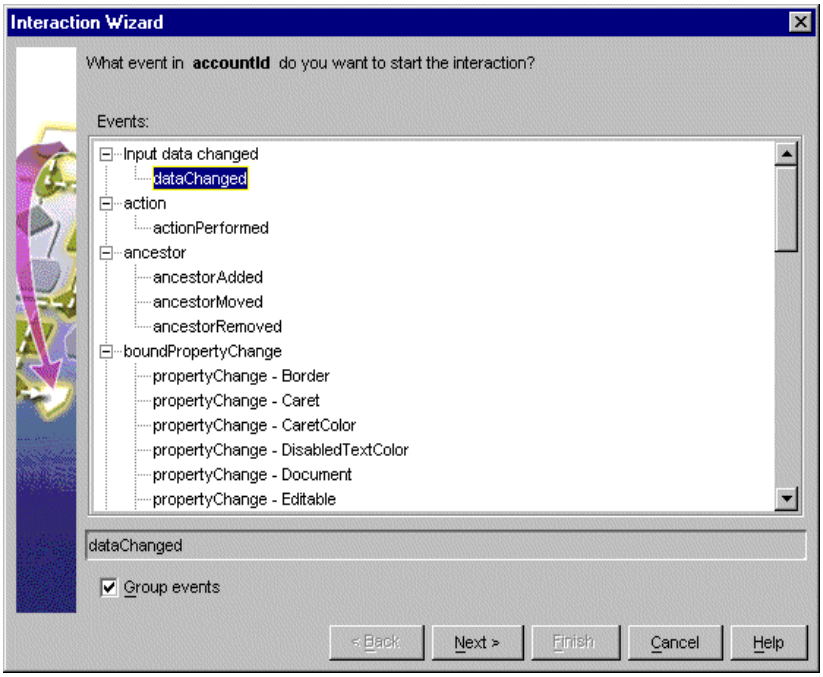
1. Click the Interaction Wizard icon in the Visual Café Form Designer window.
2. Select the **accountID JoltTextField** bean and drag a line to the JoltServiceBean.

   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 the following figure.

Figure 6-18 Select dataChanged Event



4. Click **Next**.

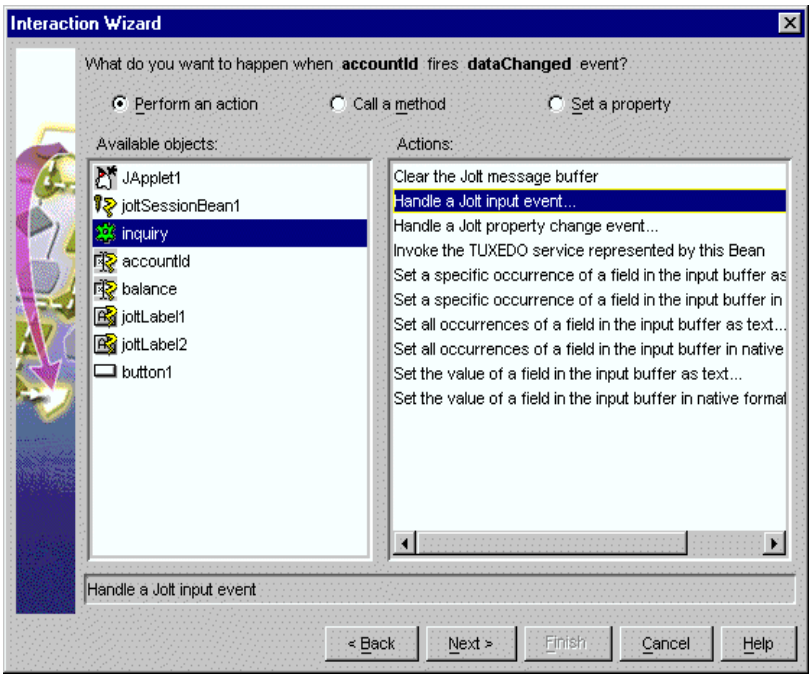
   The Interaction Wizard window is displayed as shown in the figure "Select inquiry Object and Handle a Jolt input event Action" on page 6-32, with the prompt:

   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 the following figure.

Figure 6-19 Select inquiry Object and Handle a Jolt input event Action



7. Click **Next**.

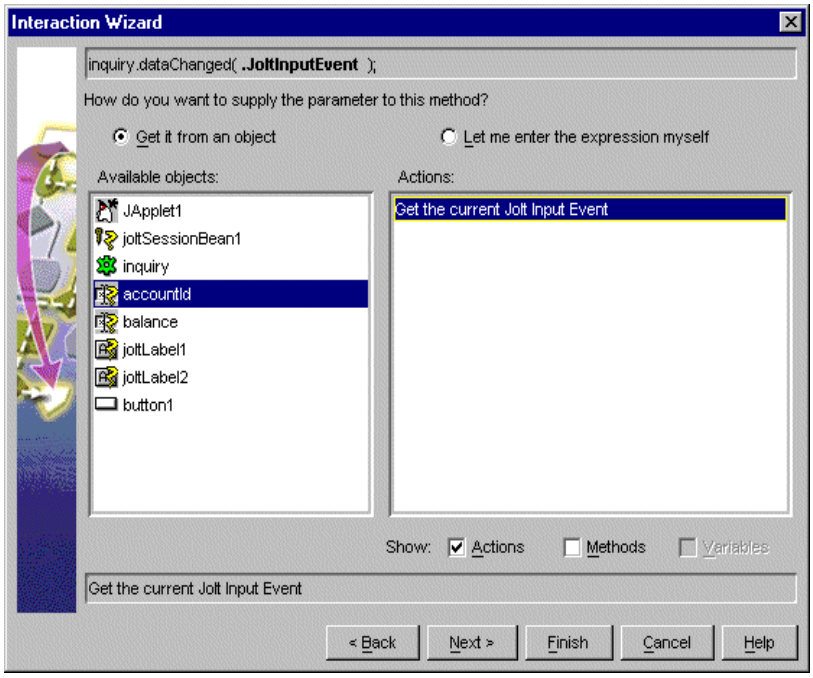
   The Interaction Wizard window is displayed as shown in "Select accountId Object and Get the current Jolt Input Event Action" on page 6-33, with the prompt:

   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 the following figure.

Figure 6-20 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

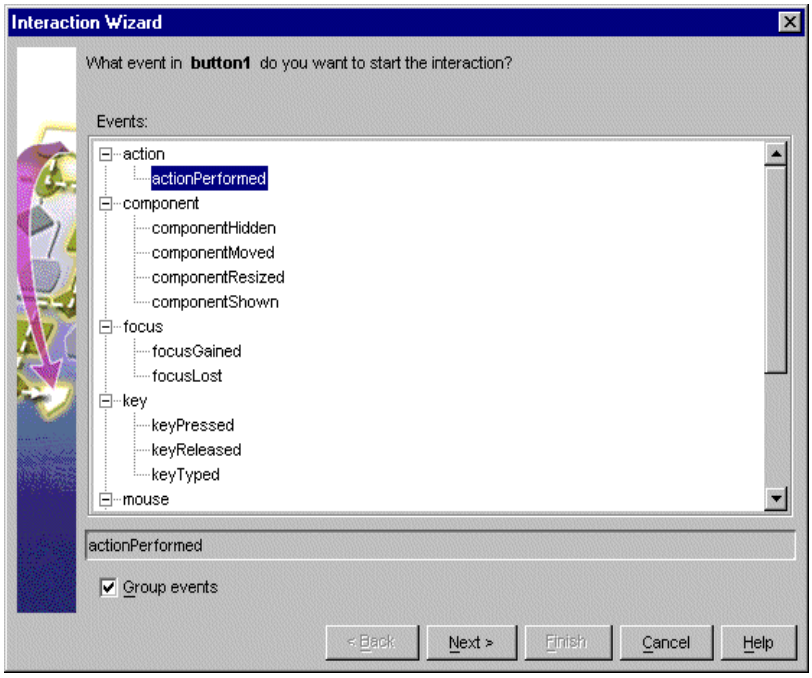
1. Click the Interaction Wizard icon in the Visual Café Form Designer window.
2. Click the Inquiry Button and drag a line to the JoltServiceBean.

   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 the following figure.

Figure 6-21 Select action Performed Event



4. Click Next.

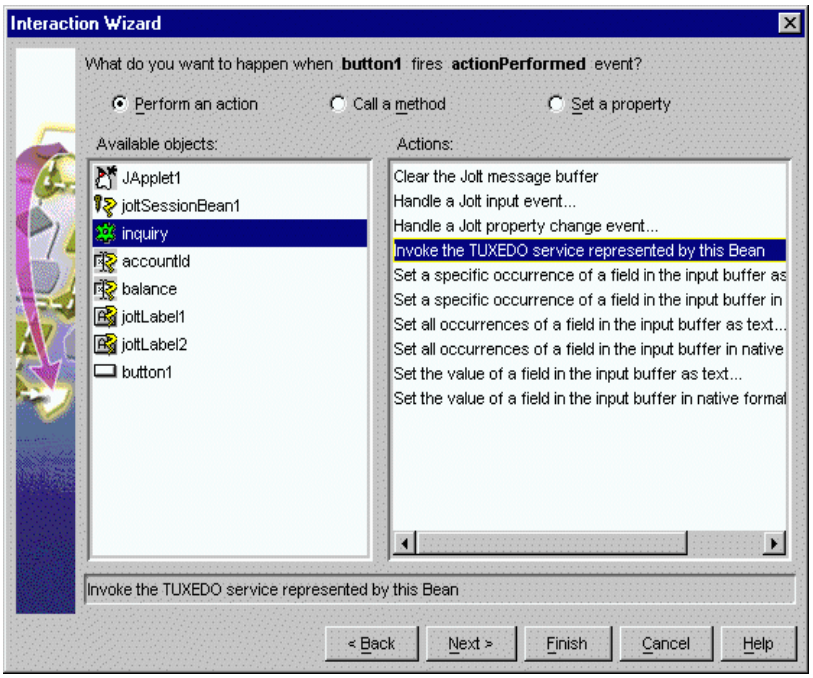
   The Interaction Wizard window is displayed, as shown in the figure "Select inquiry Object and Invoke the TUXEDO Service... Action" on page 6-35, with the prompt:

   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 the following figure.

Figure 6-22 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.

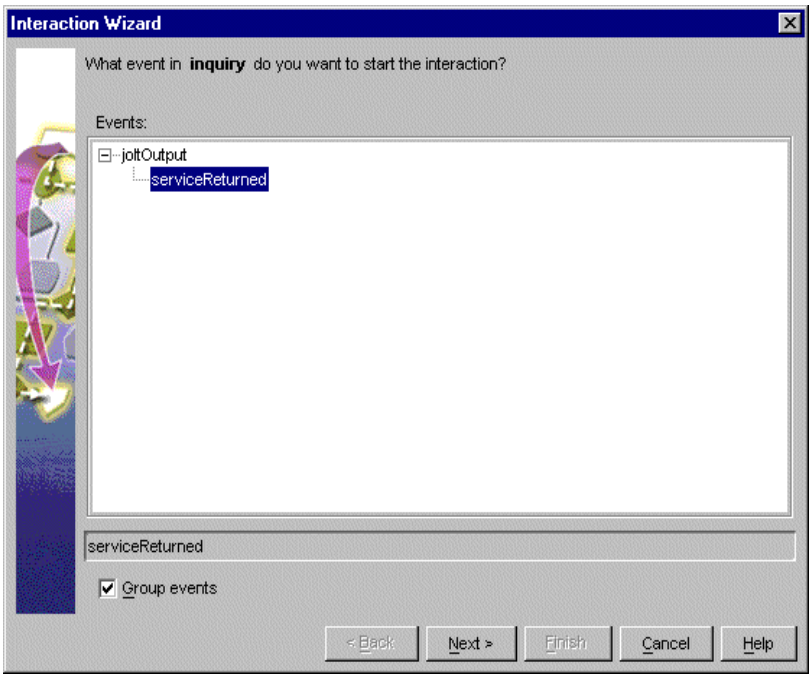
Step 5: Wire JoltServiceBean to the Balance JoltTextField Using JoltOutputEvent

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 the following figure.

Figure 6-23 Select ServiceReturned Event



4. Click Next.

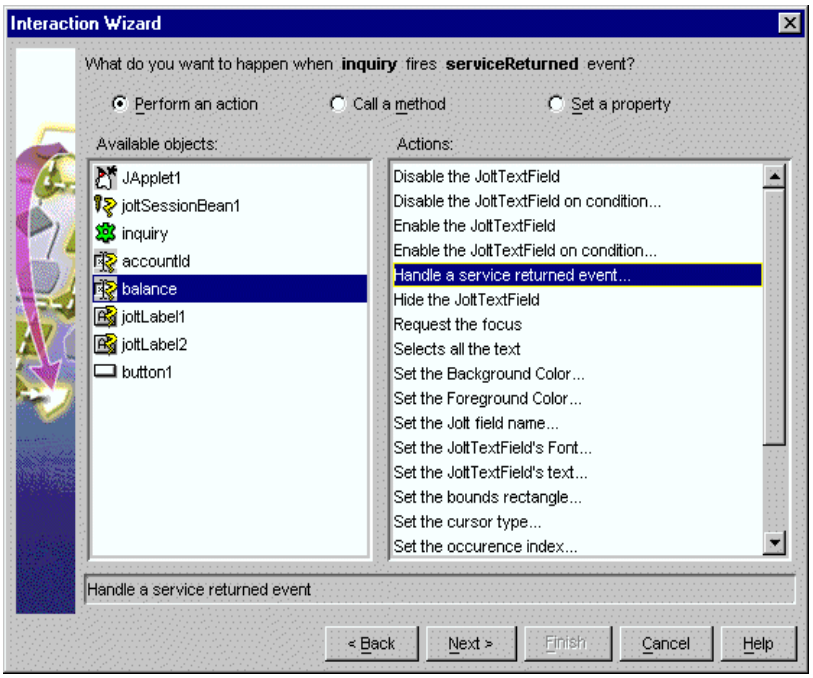
   The Interaction Wizard window is displayed, as shown in the figure "Select balance Object and Handle a service returned event Action" on page 6-37, with the prompt:

   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 6-24 Select balance Object and Handle a service returned event Action



7. Click **Next**.

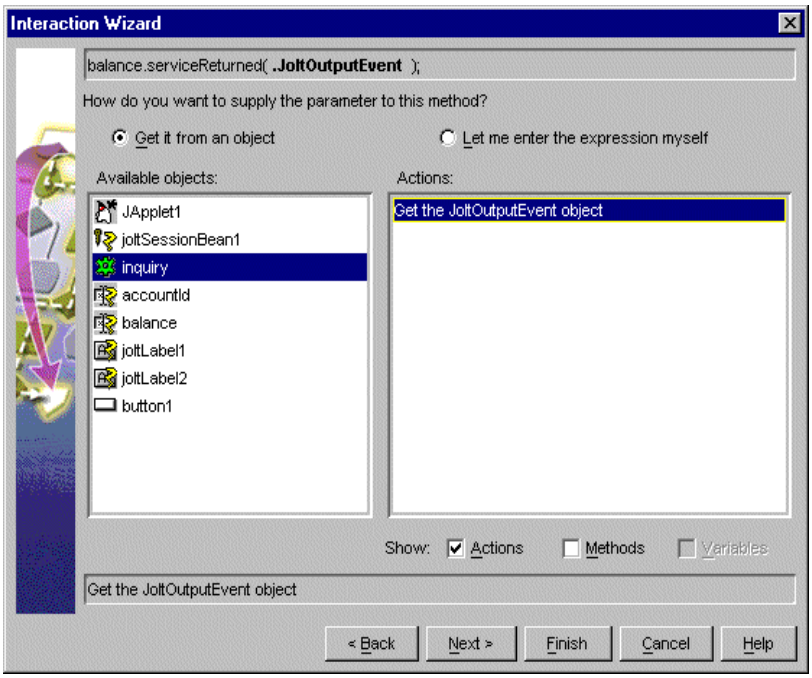
   The Interaction Wizard window is displayed, as shown in the figure "Select inquiry Object and Get the JoltOutputEvent object Action" on page 6-38, with the prompt:

   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 the following figure.

Figure 6-25 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.

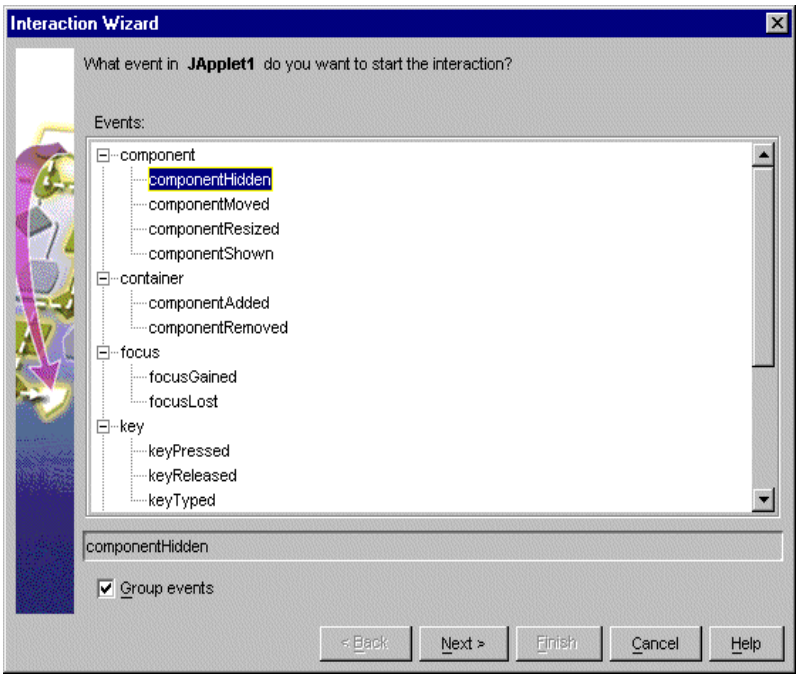
Step 6: Wire the JoltSessionBean Logoff

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 the following figure.

Figure 6-26 Select componentHidden Event



4. Click Next.

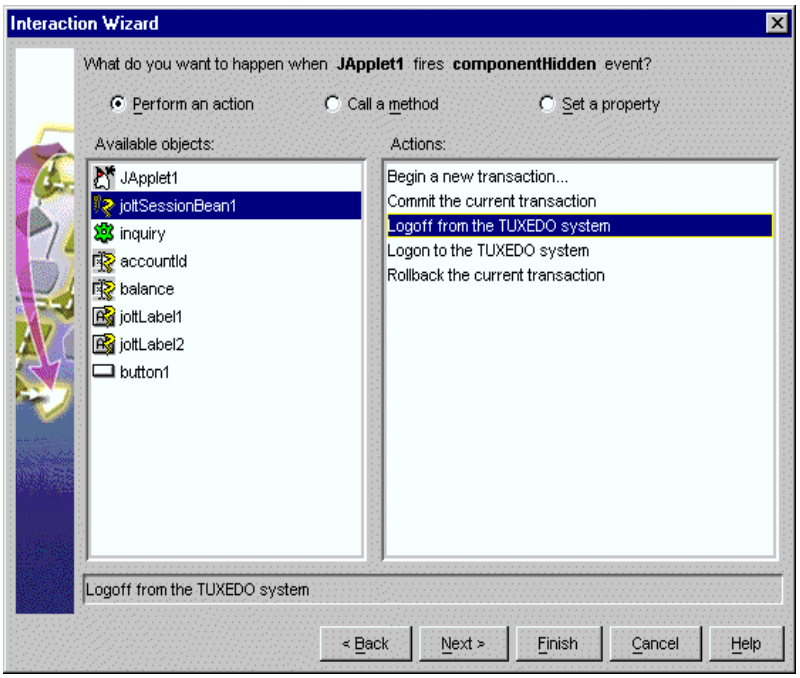
   The Interaction Wizard window is displayed, as shown in the figure "Select joltSessionBean1 Object and Logoff from the Tuxedo System Action" on page 6-40, with the prompt:

   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 the following figure.

Figure 6-27 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 Jolt Repository and Setting the Property Values." Also refer to the table "JoltBean Specific Properties" on page 6-41 and the figure "JoltServiceBean Property Editor" on page 6-42.

Running the Sample Application

To run the sample application, you must have the BEA 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 Jolt 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 the following table.

Table 6-6 JoltBean Specific Properties

JoltBean	Property	Input Description
JoltSessionBean	appAddress	e.g., `//host:port`
	userName, Password or AppPassword	Type your BEA Tuxedo username and passwords.
JoltServiceBean	serviceName	INQUIRY, for example.
	isTransactional	Set to `true` if the service needs to be executed within a transaction. Set isTransactional to `false` if the service does not require a transaction.
JoltUserEventBean	eventName	Refer to the BEA Tuxedo `tpsubscribe` calls.
	filter	

Table 6-6 JoltBean Specific Properties (Continued)

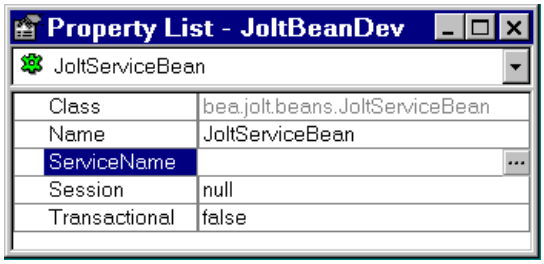
JoltBean	Property	Input Description
All Jolt-aware GUI beans	joltFieldName	ACCOUNT_ID, for example
	occurrenceIndex	Multiple fields of the same name. Index starts at 0.
JoltCheckbox	TrueValue and FalseValue	The field value corresponding to the state of the checkbox.

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 the following figure.

Figure 6-28 JoltServiceBean Property Editor



The Custom Property Editor for ServiceName shown in the following figure is displayed.

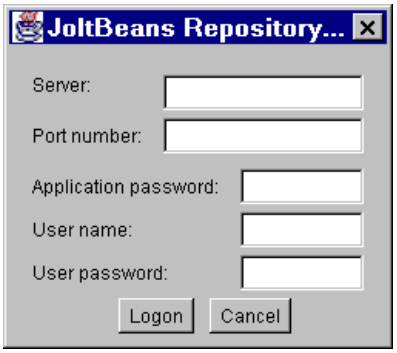
Figure 6-29 Custom Property Editor for ServiceName

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 the following figure is displayed.

Figure 6-30 JoltBeans Repository Logon



3. Type the BEA 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."

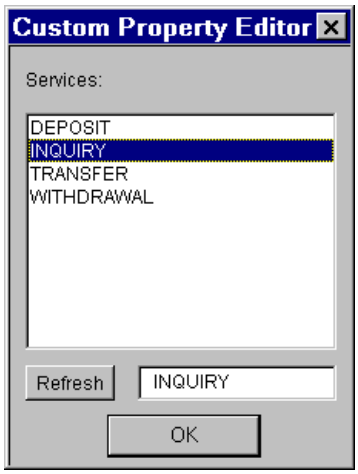
5. Select the appropriate service name from the list box, as shown in the following figure.
6. Enter the property value (service or field name) directly.

   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 6-31 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 BEA 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 BEA 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.

CHAPTER 7

Using Servlet Connectivity for BEA Tuxedo

With BEA 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 BEA Tuxedo services without directly connecting to BEA Tuxedo. HTML clients can instead connect to a Web server, through HTTP, where the BEA Tuxedo service request is executed by a generic servlet. Using a Jolt session, the servlet on the Web server administers the BEA Tuxedo service request by connecting to the BEA Tuxedo Server through the Jolt Server Handler (JSH) or the Jolt Server Listener (JSL), which then makes the BEA Tuxedo service request.

This capability allows many types of HTML clients to make remote BEA 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:

- 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.Http Servlet 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 a BEA 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 a BEA Tuxedo system. This class provides call methods that accept input parameters for a BEA 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 BEA 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 a BEA Tuxedo service
- Processing the result set

This example demonstrates how a servlet can connect to BEA 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 BEA 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
- BEA Tuxedo 8.0 or later with SimpApp sample running
- BEA Jolt

Installing the SimpApp Sample

1. Install the Jolt class library (`jolt.jar`) and Servlet Connectivity for BEA 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 BEA 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 BEA 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 BEA 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 BEA 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 a BEA 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
- BEA Tuxedo 8.0 or later with BankApp sample running
- BEA Jolt

Installation Instructions

1. Install the Jolt class library (`jolt.jar`) and Servlet Connectivity for BEA 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 BEA 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 BEA 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 BEA 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
- BEA Jolt

Installation Instructions

1. Install the Jolt class library and Servlet Connectivity for BEA 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:

BEA 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`

Servlet Interest Group

`http://servlet-interest@java.sun.com`

APPENDIX A

BEA Jolt Exceptions

This appendix describes all the BEA Jolt exceptions that you may encounter. Keep in mind that the Jolt Class Library returns both BEA Jolt and BEA Tuxedo exceptions.

For details about BEA Tuxedo exceptions, refer to the appropriate document in the following list:

- *BEA Tuxedo Command Reference*
- *BEA Tuxedo ATMI C Function Reference*
- *BEA Tuxedo ATMI COBOL Function Reference*
- *BEA 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 *BEA Jolt API Reference*.

The following table lists the BEA Jolt and BEA Tuxedo exceptions that you may encounter while running BEA Jolt. Each exception includes a possible cause (or causes) and a recommended action (wherever possible) to help resolve the situation

1. TPEABORT	**A transaction could not commit.**	
	Cause	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 BEA Tuxedo, you can get this exception if `tpcommit()` is called with outstanding replies or open conversation connections.
	Action	Check transaction failures on the server side. BEA Jolt clients should resend the request after the transaction problem has been fixed on the server side.
2. TPEBADDESC	**This exception should not occur in BEA Jolt.**	
	Cause	In BEA Tuxedo, this exception usually occurs when an invalid caller descriptor is given to `tpgetrply()` or `tpsend()`.
	Action	None.
3. TPEBLOCK	**A blocking condition has occurred and the `TPNOBLOCK` flag is specified in BEA Tuxedo.**	
	Cause	This exception occurs because the server is backed up.
	Action	You may need to re-examine and re-architect the application to handle extreme load cases.
4. TPEINVAL	**Invalid arguments were given by the application.**	

	Cause	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 `openConnection()` method. The `TPEINVAL` exception can also occur if you specified a hexadecimal address for the JSL `-H` option without a leading "`0x`" , or if you entered a wrong address in `UBBCONFIG` file. In addition, the `GETREC()`, `DELREC()` and `GETSVC()` services in `JREPSVR` can return `TPEINVAL` if the `REPNAME` is missing. Also, the `ADDREC()` service in `JREPSVR` can return `TPEINVAL` if the `REPVAL` is not specified.
	Action	This type of exception should have been handled during the application development cycle. You should not receive this exception in a production environment.
5. TPELIMIT	**The maximum number of outstanding requests or subscriptions has been reached.**	
	Cause	The maximum number of outstanding requests has been reached. This exception could also mean that the BEA Tuxedo System Event Broker's maximum number of subscriptions (50 internally defined for now) has been reached.
	Action	You may need to re-examine and re-architect the application to handle load extreme cases.
6. TPENOENT	**The requested service is not available.**	
	Cause	Usually, the requested service is not booted or advertised on the BEA Tuxedo server side. It is also possible that the requested service is not defined in the Jolt Repository. This exception could also indicate that you could not access the BEA Tuxedo System Event Broker.
	Action	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 Jolt Repository. After the service is available on the server side, Jolt clients should resend the request.
7. TPEOS	**An operating system exception has occurred.**	

	Cause	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.
	Action	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.**	
	Cause	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 `openConnection()` method. If the Jolt Repository is set to read-only, the `ADDREC()` and `DELREC()` services, or the `GARBAGECOLLECT()` service in `JREPSVR`, also return the `TPEPERM` exception.
	Action	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.**	
	Cause	For this exception, an improper context could include a `rollback()` or `commit()` 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.
	Action	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 BEA Tuxedo.**	

	Cause	The service routine is returning application-level failures, which may include any of the following: an application calls `tpreturn()` or `tpforward()` with invalid flags, the caller descriptor is no longer valid, or there are invalid return values.
	Action	This type of exception should have been handled during the application development cycle. You should not receive this exception in a production environment.
11. TPESVCFAIL	**The service routine sending the caller's reply called `tpreturn()` with `TPFAIL`.**	
	Cause	The service routine is returning application-level failures.
	Action	This type of exception should have been handled during the application development cycle. You should not receive this exception in a production environment.
12. TPESYSTEM	**A BEA Tuxedo system exception has occurred.**	
	Cause	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.
	Action	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.
13. TPETIME	**A transaction timeout has occurred.**	
	Cause	There is a transaction timeout on the server side.
	Action	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.
14. TPETRAN	**The requested service belongs to a server that does not support transactions and `TPNOTRAN` is not set.**	

	Cause	A transaction is not supported for the requested service.
	Action	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.
15. TPGOTSIG	**An unexpected signal was received.**	
	Cause	A signal was received and the `TPSIGSTRT` flag was not specified.
	Action	None.
16. TPERMERR	**A resource manager failed to open or close correctly on the server side.**	
	Cause	The resource manager might not be available; or all the resource might not be released or committed before close.
	Action	Check the ULOG file for reasons why the resource manager failed to open or close on the server side.
17. TPEITYPE	**For the JoltRemoteService class, the requested BEA Tuxedo service does not recognize the type and subtype of the input data.**	
	Cause	The type and subtype of input data is not defined in the Jolt Repository.
	Action	The type and subtype of input data should be defined in the Jolt Repository. This type of exception should have been handled during the application development cycle. You should not receive this exception in a production environment.
18. TPEOTYPE	**For the JoltRemoteService class, the BEA Tuxedo caller does not recognize the type and the subtype of the reply data.**	
	Cause	The type and subtype of output data is not defined in the Jolt Repository.
	Action	The type and subtype of output data should be defined in the Jolt Repository. This type of exception should have been handled during the application development cycle. You should not receive this exception in a production environment.
19. TPERELEASE	**This exception should not occur in BEA Jolt.**	

	Cause	Usually, this exception occurs when an unsolicited notification message is sent from a server with the TPACK flag set, and the target is a Jolt client from an older release of BEA Jolt that does not support the acknowledgment protocol.
	Action	Verify that the correct version of BEA 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 an production environment.
20. TPEHAZARD	**Due to some failure, the work done on behalf of the transaction may have been heuristically completed.**	
	Cause	Check the ULOG file on the server side for details.
	Action	None.
21. TPEHEURISTIC	**Due to a heuristic decision, the work done on behalf of the transaction was partially committed and partially aborted.**	
	Cause	Check the ULOG file on the server side for details.
	Action	None.
22. TPEEVENT	**This exception should not occur in BEA Jolt.**	
	Cause	Usually, this exception means that an event has occurred when sending or receiving a message in a conversational connection in BEA Tuxedo. However, conversational server connections are not available in BEA Jolt.
	Action	None.
23. TPEMATCH	**The JoltUserEvent class has implemented a subscription to an asynchronous notification event, but the subscription has failed because it matches an existing subscription.**	
	Cause	The subscription failed because it matched one already listed with the BEA Tuxedo System Event Broker.
	Action	None.
24. TPEDIAGNOSTIC	**This exception should not occur in BEA Jolt.**	

	Cause	Usually, this exception occurs when enqueuing or dequeuing a message from the specified queue fails in BEA Tuxedo. However, enqueing and dequeing of messages is not available in BEA Jolt.
	Action	None.
25. TPEMIB	**This exception should not occur in BEA Jolt.**	
	Cause	Usually, this exception occurs when an administrative request via `tpadmcall()` has failed in BEA Tuxedo. However, TMIB calls are not available in BEA Jolt.
	Action	None.
26. TPEJOLT	**This exception indicates there is a problem in BEA Jolt.**	
	Cause	The `TPEJOLT` exception could occur for any of the following reasons: • JoltSession class—the `send()`, `recv()` or `cancel()` methods throw `TPEJOLT` if the session object or message ID is invalid. • JoltSession class—throws `TPEJOLT` when `TPINIT` data conversion fails. • bea.jolt.pool.connection class—throws `TPEJOLT` when a run-time exception occurs. • JoltRemoteService—the `call()` method throws `TPEJOLT` when the buffer conversion between BEA Jolt and BEA Tuxedo fails, the requested service is not defined in the Jolt Repository, the requested service does not the right version, or the reply data conversion fails. • JoltUserEvent class—throws `TPEJOLT` when event name conversion fails, an invalid message ID is encountered, or unsolicited message data conversion fails.
	Action	This type of exception should have been handled during the application development cycle. You should not receive this exception in a production environment.

Index

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X