BEA Jolt

Developer's Guide

Jolt 1.2 Release
Document Edition 1.2
October 1999

Copyright

Restricted Rights Legend

Trademarks or Service Marks

BEA Jolt Developer's Guide

Document Edition	Part Number	Date	Software Version
1.0		July 1999	BEA Jolt 1.2

Contents

1. Introducing BEA Jolt

2. Bulk Loading Tuxedo Services

3. Using the Jolt Repository Editor

4. Using the Jolt Class Library

5. Using JoltBeans

6. Using Servlet Connectivity for Tuxedo

7. Using Jolt ASP Connectivity for Tuxedo

1 Introducing BEA Jolt

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

Since these tasks are done in the Jolt API and the Jolt Repository Editor using the Java programming language, the Jolt documentation assumes a familiarity with BEA Tuxedo and Java programming. This documentation is intended for system administrators, network administrators and developers.

"Introducing BEA Jolt" covers the following topics:

- What is BEA Jolt?
- Key Features
- How It Works
  - Jolt Servers
  - Jolt Class Library
  - JoltBeans
  - ASP Connectivity for Tuxedo
  - Jolt Server and Jolt Client Communication
  - Jolt Repository
  - Jolt Internet Relay

What is BEA Jolt?

BEA Jolt is a Java class library and API that provides an interface to BEA Tuxedo and WLE from remote Java clients. BEA Jolt consists of several components for creating Java-based client programs that access Tuxedo services. These Jolt components are as follows:

- **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 Tuxedo-based applications running on one or more 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 can be used 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 AWT and Swing-based beans).
- **Jolt Repository—**A central Jolt Repository contains definitions of BEA Tuxedo services. These Repository definitions are used by Jolt at runtime to access 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 eliminates the need for the JSH and 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 Tuxedo services and extend your transaction environment to the corporate intranet or world-wide Internet. The key feature of the Jolt architecture is its simplicity. Using Jolt, 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 Developer's Kit (JDK) 1.2 and is fully compatible with Java threads. Jolt enables Java programmers to build graphical front-ends that use the Tuxedo application and transaction services without the need to understand detailed transactional semantics or without having to rewrite existing 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 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 Tuxedo Services via 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 Tuxedo services into the Jolt development environment. Jolt and Tuxedo simplify network and application scalability, while encouraging the reuse of application components.
- **GUI-based Maintenance and Distribution of 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

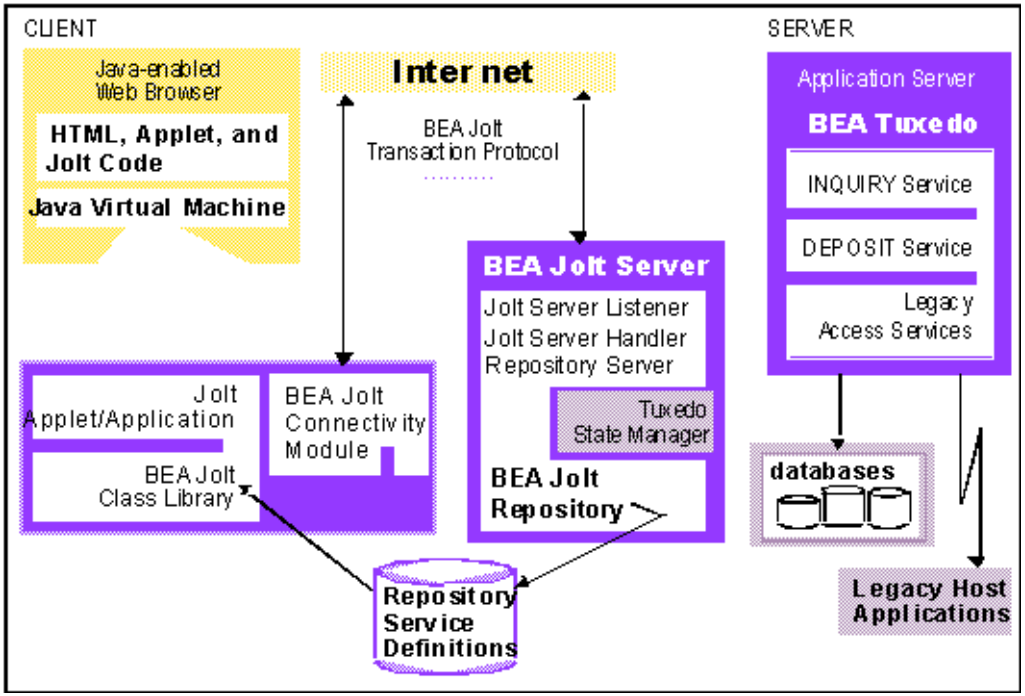
addresses the issue of security that is essential for secure Internet transaction processing.

- **Added Security via Internet Relay—**BEA Jolt features an Internet Relay component that allows network administrators to separate their Web Server and Tuxedo application server. Web servers are generally considered insecure as they often exist outside a corporate firewall. The Jolt Internet Relay gives you greater flexibility to locate your BEA Tuxedo server in a secure location or environment on your network, yet still be able to handle transactions from Jolt clients on the Internet.
- **Event Subscription Support—**Jolt Event Subscription is used to receive event notifications from either Tuxedo services or other Tuxedo clients. Jolt Event Subscription lets you handle two types of Tuxedo application events:
  - Unsolicited Event Notifications. A Jolt client can receive these notifications when a Tuxedo client or service subscribes to unsolicited events and a Tuxedo client issues a broadcast or a directly targeted message.
  - Brokered Event Notifications. The Jolt client receives these notifications via the Tuxedo Event Broker. The Jolt client receives these notifications only when it subscribes to an event and any Tuxedo client or server posts an event.

How It Works

BEA Jolt connects Java clients to applications built using the BEA Tuxedo system. The Tuxedo system provides a set of modular services, each offering specific functionality related to the application as a whole. (Figure 1-1 illustrates the end-to-end view of the BEA Jolt architecture, as well as related Tuxedo components and their interactions.) For 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 Tuxedo service on behalf of the client. The BEA Jolt Server accepts requests from the Jolt clients and maps those requests into Tuxedo service requests.

Figure 1-1 BEA Jolt Architecture



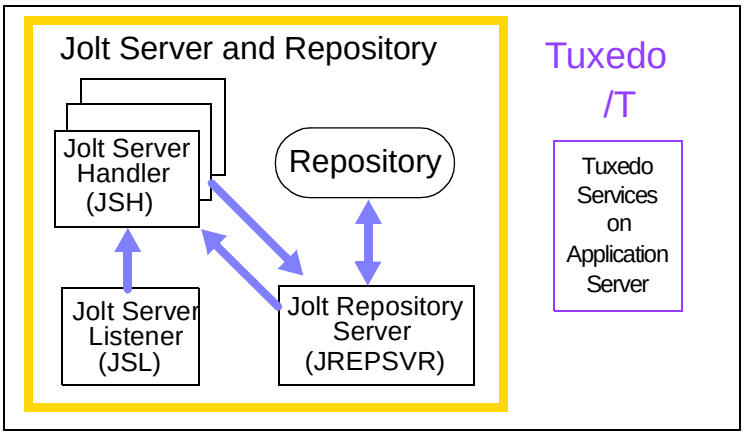
Jolt Servers

The Jolt Server has several components that act in concert to pass Jolt client transaction processing requests to the Tuxedo application. The components are as follows:

- **Jolt Server Listener (JSL)**. The JSL handles the initial Jolt client connection, and is responsible for assigning 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 Tuxedo buffer data into the Jolt buffer and vice versa.
- **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 Components



Jolt Class Library

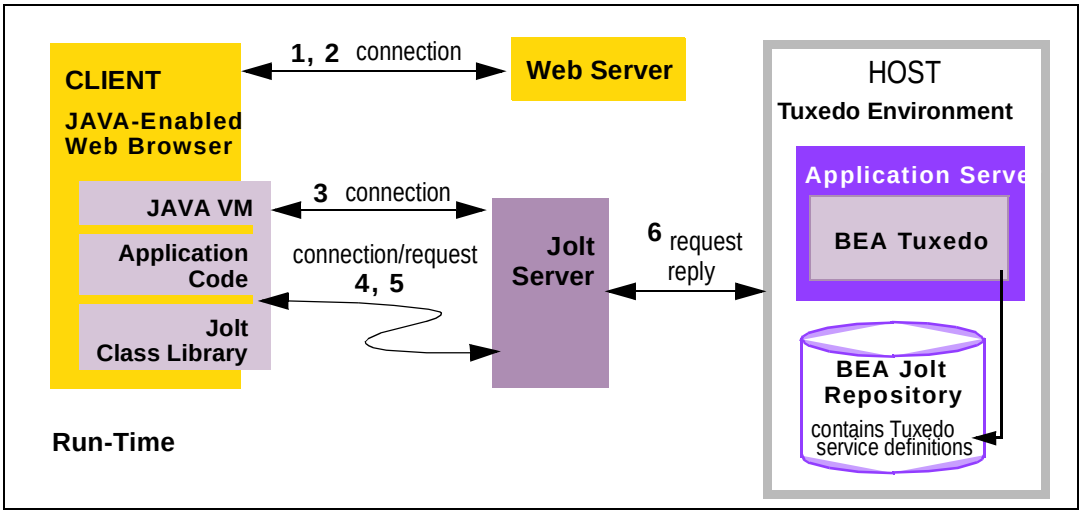
The BEA Jolt Class Library is a set of classes you can use in your Java application or applet to make service requests to the Tuxedo system from a Java enabled client. These Jolt classes allow you to access Tuxedo transaction services using objects.

When developing a Jolt client application, you only need to know about the classes that Jolt provides and the Tuxedo services that are exported by the Jolt Repository. Jolt hides the underlying application details. Using Jolt and Jolt's 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 runtime. As a result, there is no need for client installation, administration, management, or version control. If services are changed, the client application becomes aware of 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 Tuxedo system. The call-out numbers correspond to descriptions of the activity in the table that follows.

Figure 1-3 Using the Jolt Class Library to access Tuxedo services



The following table briefly describes the flow of activity involved in using the Jolt Class Library to access Tuxedo services.

Table 1-1 Using the Jolt Class Library

Process	Step	Action
Connection	**1**	A Java enabled Web browser downloads an HTML page using the HTTP protocol.
...	**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) and can build a service request object based on Jolt class definitions, and make a method call.

Table 1-1 Using the Jolt Class Library

...	**5**	The request is sent to the Jolt Server, which translates the Java based request into a Tuxedo request and forwards the request to the Tuxedo environment.
Reply	**6**	The 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 can use in a Java-enabled integrated development environment (IDE) to construct BEA Jolt clients. Using JoltBeans, you can create Jolt client applications with the ease of using typical JavaBeans. You can use popular JavaBeans-enabled development tools like Symantec Visual Café to graphically construct client applications.

BEA JoltBeans provide a JavaBeans-compliant interface to BEA Jolt. A fully functional BEA Jolt client can be developed 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 using JoltBeans is integrated with the BEA Jolt repository, providing easy access to available BEA Tuxedo functions.

ASP Connectivity for Tuxedo

The Jolt ASP Connectivity for Tuxedo Toolkit is an extension to the Jol Java class library. The Toolkit allows the Jolt client class library to be used in a Web server, such as the Microsoft Internet Information Server (IIS), to provide an interface between HTML clients or browsers, and Tuxedo services.

The Jolt ASP Connectivity for Tuxedo provides an easy-to-use interface for processing and generating dynamic HTML pages. You do not need to learn how to write Common Gateway Interface (CGI) transactional programs to access Tuxedo services.

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. 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 any data conversions necessary, 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 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.

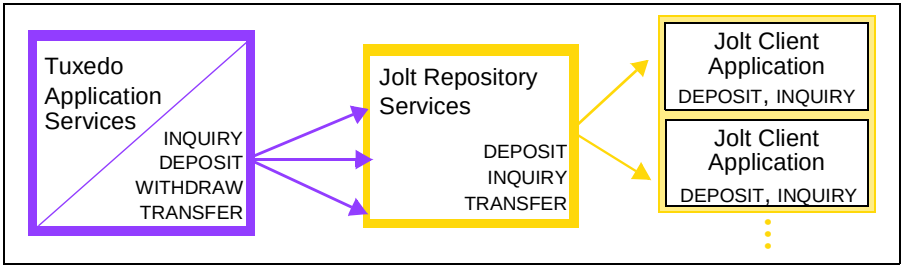
Jolt Repository

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

All Repository services that are exported to one client are exported to all clients. Tuxedo handles the cases where subsets of services may be needed for one client and not others. Figure 1-4 illustrates how the Jolt Repository brokers Tuxedo services to

multiple Jolt client applications. The diagram shows four Tuxedo services, however the WITHDRAW service is not defined in the Repository and the TRANSFER service is defined but not exported.

Figure 1-4 Distributing Tuxedo Services via 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. With the Jolt Repository Editor you can define, test, and export services to Jolt clients.

Note: The Jolt Repository Editor only controls services for Jolt client applications. It cannot be used to make changes to the Tuxedo application.

The Jolt Repository Editor lets you extend and distribute Tuxedo services to Jolt clients without having to modify many lines of code. With the Jolt Repository Editor, you can modify parameters for Tuxedo services, logically group 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 allow it to work with Jolt clients, the Jolt Relay eliminates the need for the Tuxedo system to run on the same machine as the Web server.

How to Jolt your Tuxedo Applications

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

The following steps show just how quickly and easily Jolt clients can be created and deployed.

1. Begin the process with a Tuxedo system application.

   For information about installing Tuxedo and creating a Tuxedo application, refer to the Tuxedo documentation set.

2. Install the Jolt system.

   Refer to *Installing the BEA Tuxedo System*.

3. Configure and define services using the Jolt Repository Editor or the Bulk Loader.

   For information regarding configuring the Jolt Repository Editor and making Tuxedo services available to Jolt, refer to: Appendix B, “System Messages”

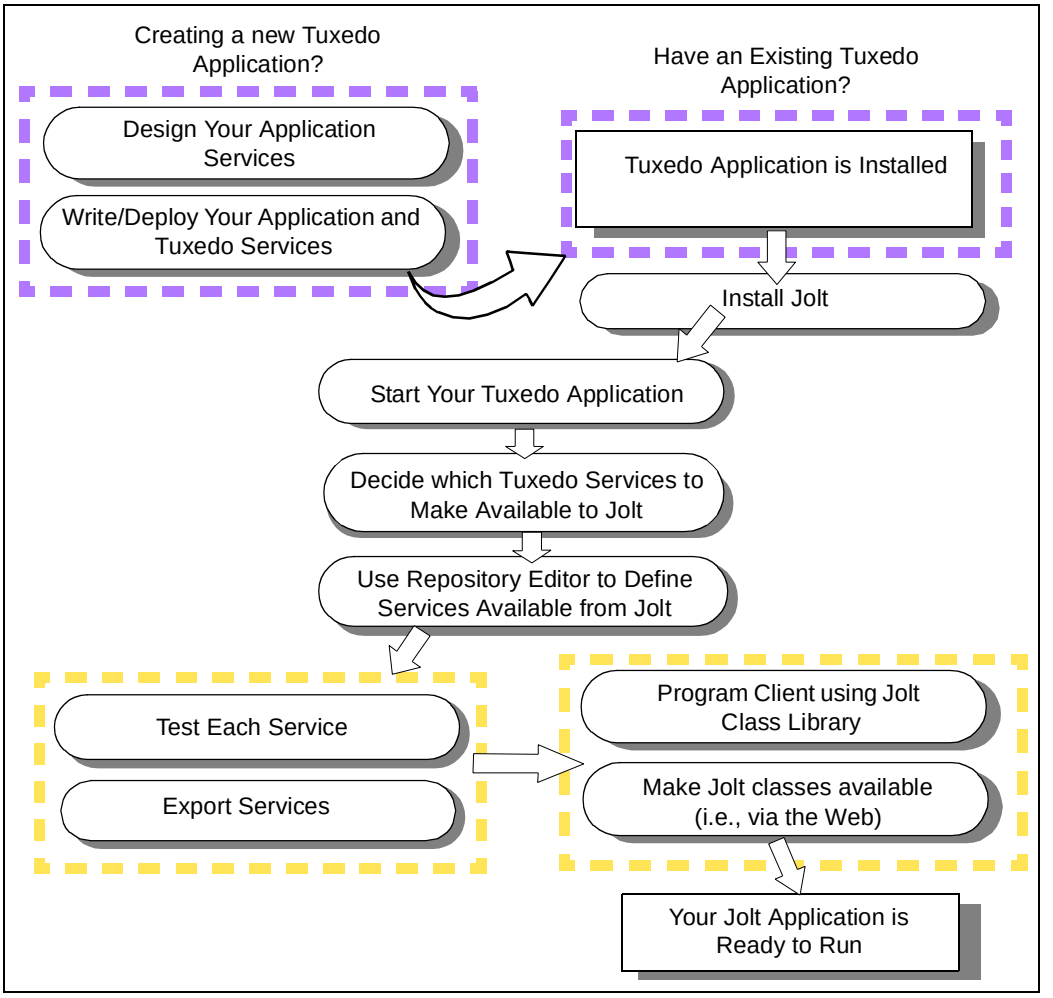
4. Create a client application using the Jolt Class Library.

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

   - "Using the Jolt Class Library"
   - API Reference in Javadoc

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

Figure 1-5 Creating a Jolt Application



2 Bulk Loading Tuxedo Services

As a systems administrator, you may have an existing Tuxedo application with multiple 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 Tuxedo services to the repository database in a single step. Using the `jbld` program, the bulk loader utility reads the 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 have populated the Jolt Repository, you can create, edit, and group services using the Jolt Repository Editor.

"Bulk Loading Tuxedo Services" covers the following topics:

- Using the Bulk Loader
- Syntax of the Bulk Loader Data Files
- Troubleshooting
- Sample Bulk Load Data

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 (i.e., `jolt.jar and joltadmin.jar`) is located. If it 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.

To Activate The Bulk Loader

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

```
java bea.jolt.admin.jbld [-n][-p package][-u name][-r role] addr
file
```

Command Line Options

The following table describes the bulk loader command-line options.

Table 2-1 Bulk Loader Command-line Options

Option	Description
`-u usrname`	Specifies the user name (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: BULKPKG)

Table 2-1 Bulk Loader Command-line Options

Option	Description
`//host:port`	Specifies the JRLY or JSL address (host name and IP port number). (Mandatory)
`filename`	Specifies the file containing the service definitions. (Mandatory)

About 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 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" 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**

The input file may 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 following listing). Following the `service=<NAME>` keyword, all of the 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 the parameters associated with the service must be specified. Following each of the `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 Correct Example of 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=” keyword. Services using CARRAY or STRING buffer types should only have one parameter in the service. The recommended parameter name for a CARRAY service is “CARRAY” with “carray” as the data type. For a STRING service, the recommended parameter name is “STRING” with “string” as the data type.

To review the service-level keywords and values, see Table 2-3.

Table 2-3 Service-Level Keywords and Values

Keyword	Value
service	Any Tuxedo service name
export	true or false (default is false)
inbuf/outbuf	Select one of these buffer types: FML FML32 VIEW VIEW32 STRING CARRAY X_OCTET X_COMMON X_C_TYPE
inview	Any view name for input parameters (optional; only if VIEW or VIEW32 or X_COMMON or X_C_TYPE buffer type is used)
outview	Any view name for output parameters (optional)

Using Parameter-Level Keywords and Values

A parameter begins with the “param=” 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” keyword.

Review the parameter-level keywords and values in the following table.

Table 2-4 Parameter-Level Keywords and Values

Keyword	Values
param	Any parameter name
type	byte short integer float double string carray
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 any problems using the bulk loader utility, see the following table. For a complete list of bulk loader utility error messages and solutions, see Appendix B, "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
The data type is invalid	Check to ensure that the parameter is using a valid data type

Sample Bulk Load Data

The following listing shows a sample data file in the correct format using the UNIX command `cat servicefile`. This example loads TRANSFER 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
```

3 Using the Jolt Repository Editor

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

"Using the Jolt Repository Editor" covers the following topics:

- Introduction to the Repository Editor
- Getting Started
- Main Components of the Repository Editor
- Setting Up Packages and Services
- Grouping Services Using the Package Organizer
- Modifying Packages/Services/Parameters
- Making a Service Available to the Jolt Client
- Testing a Service
- Troubleshooting

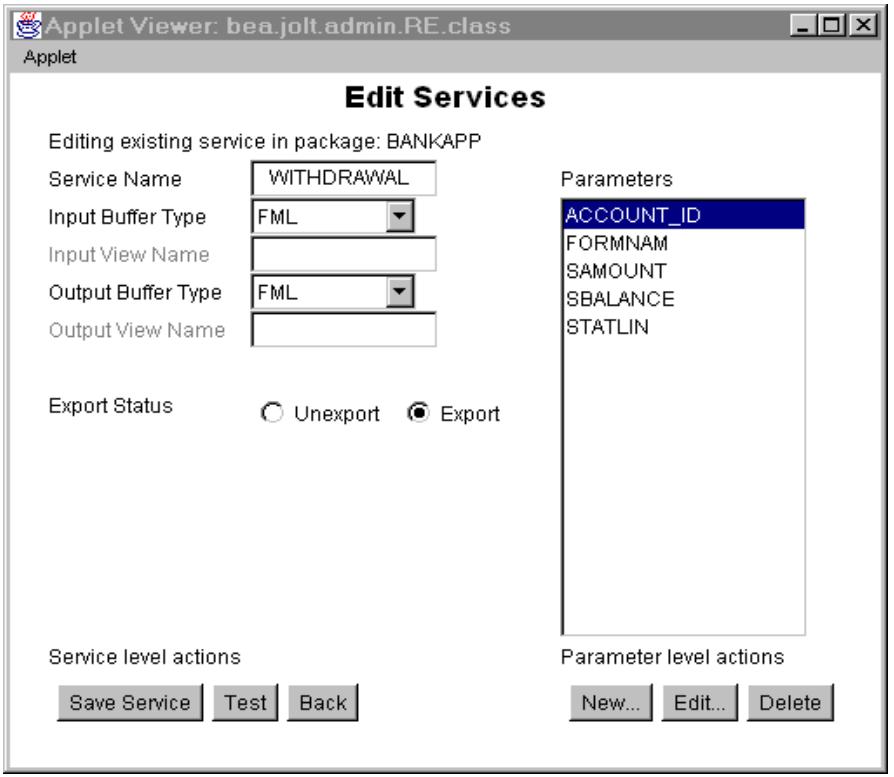
Introduction to the Repository Editor

The Jolt Repository is used internally by Jolt to translate Java parameters to a Tuxedo type buffer. The Repository Editor is available as a downloadable Java applet. When a 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 following figure illustrates the parts of a sample window. Details are explained in the “Repository Editor Window Parts” table.

Figure 3-1 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 3-1 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 display the Packages window, Services window, or Package Organizer. In the previous figure, command buttons include: "Save Service," "Test," "Back," "New," "Edit," "Delete."
5 Status	View the current status of the Repository Editor service or package. (This item does not appear in the previous figure.)
6 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 you have installed all the necessary Jolt components (at least the Jolt Server and the Jolt Client). To use the Repository Editor, you must:

- Start the Repository Editor.
- Log on to the Repository Editor.

Note: For information on exiting the Repository Editor after you have entered information, refer to "Exiting the Repository Editor" in this chapter.

Start the Repository Editor from either the JavaSoft `appletviewer` or from your Web browser.

Starting the Repository Editor Using the Java Applet Viewer

To start the 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 Repository Editor logon window displays.

Starting the Repository Editor Using 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
   ```

To start the Repository Editor 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 Repository Editor logon window displays.

Logging on to the Repository Editor

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

1. Type the name of the server machine designated as the "access point" to the Tuxedo application and select the Port Number text field.
2. Type the Port Number and press **Enter**. The system validates the server and port information.

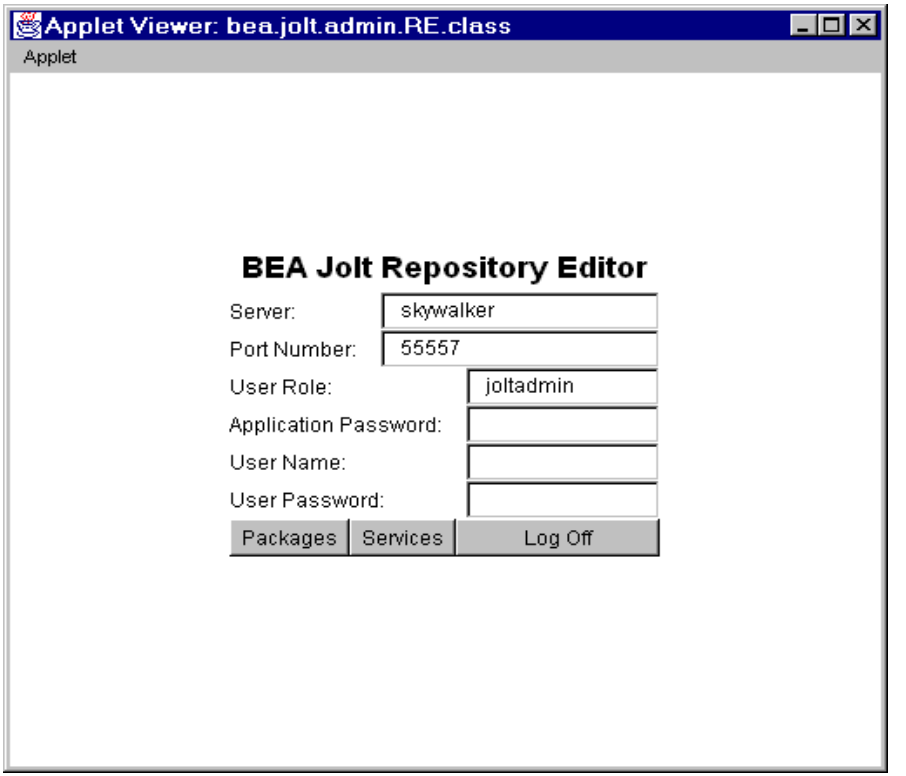
   Note: Unless you are logging on through the Jolt Relay, the same port number is used to configure the Jolt Listener. Refer to your `UBBCONFIG` file for additional information.
3. Type the Tuxedo Application Password and press **Enter**. Based on the authentication level, type the remaining information.
4. Type the Tuxedo User Name and press **Tab**.
5. Type the Tuxedo User Password and press **Enter**.

Note: See the `JoltSessionClass` for additional information.

The **Packages** and **Services** options are activated.

The following figure is an example of the Repository Editor logon window.

Figure 3-2 Repository Editor Logon Window



Repository Editor Logon Window Description

The following listing details the Repository Editor logon window.

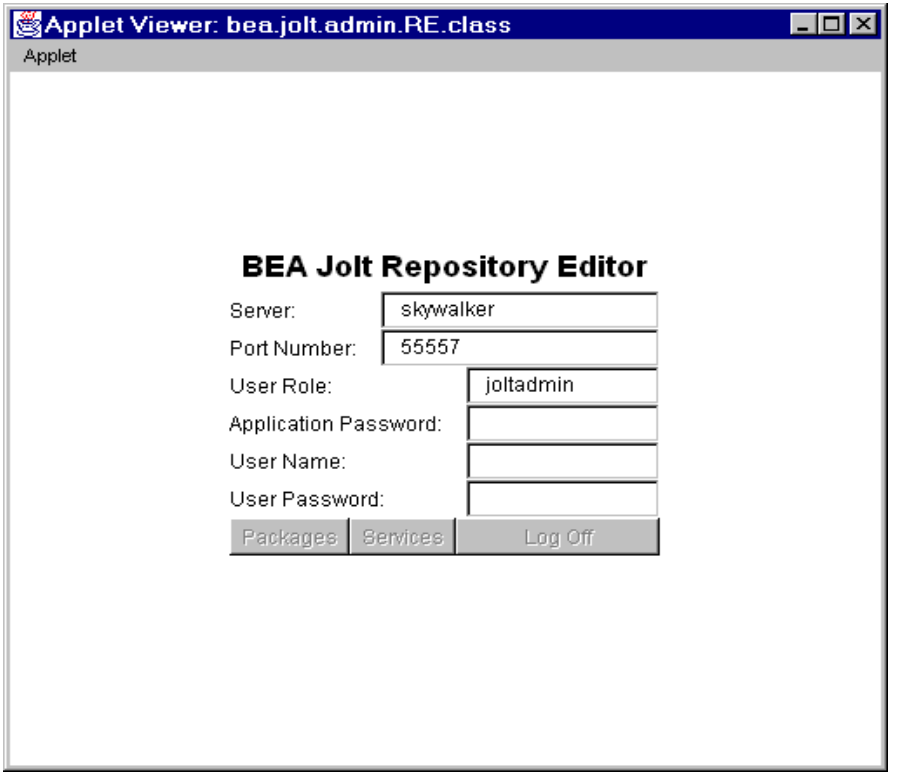
Option	Description
Server	The server name.

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 Tuxedo application.
User Role	
Application Password	Tuxedo administrative password text entry.
User Name	Tuxedo user identification text entry. The first character must be an alpha character.
User Password	Tuxedo password text entry.
Packages	This button accesses the Packages window. (Enabled after the logon.)
Services	This button accesses the Services window. (Enabled after the logon.)
Log Off	This button terminates the connection with the server.

Exiting the Repository Editor

Exit the Repository Editor when you are finished adding, editing, testing, or deleting packages, services, and parameters. The following figure is an example of the Repository Editor window before exiting. Only **Packages**, **Services**, and **Log Off** are enabled. All text entry fields are disabled.

Figure 3-3 Example of the Repository Editor Logon Window Before Exiting



To exit the Repository Editor:

1. Select **Back** from a previous window to return to the Logon window.
2. Select **Log Off** to terminate the connection with the server. The Repository Editor Logon window continues to display with disabled fields.
3. Select **Close** from your browser menu to remove the window from your screen.

Main Components of the Repository Editor

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

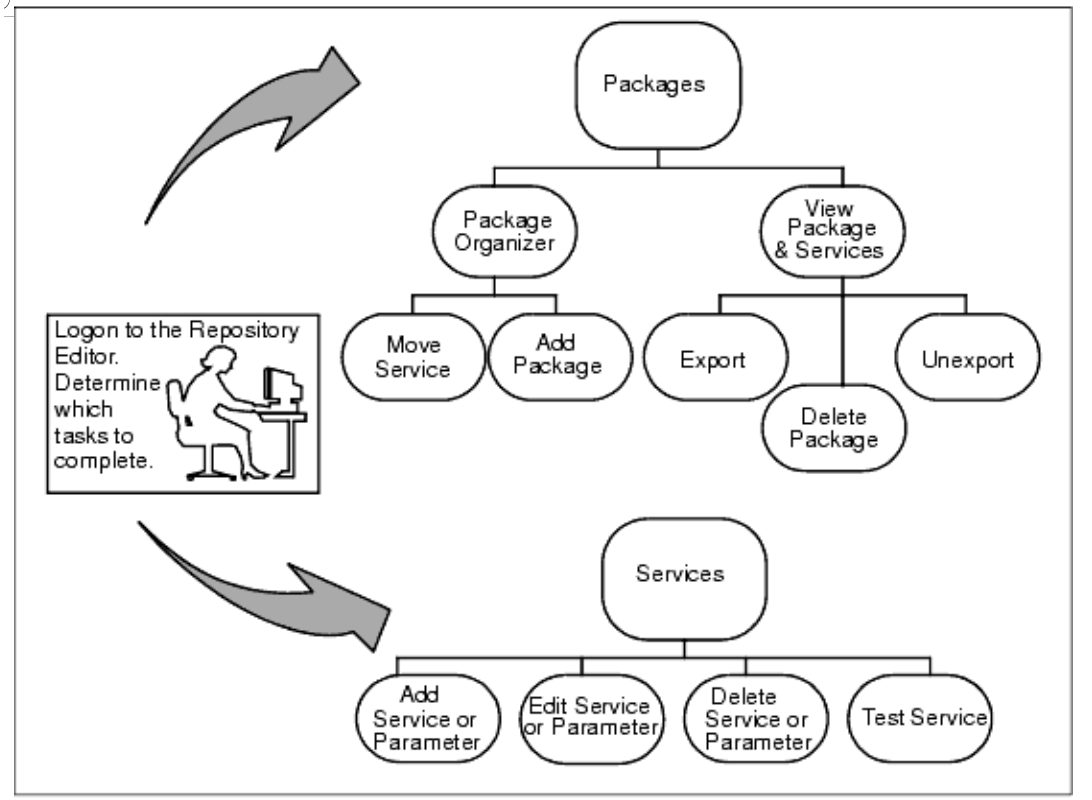
- Packages
- Services
- Parameters

In addition, you can test and group Services.

Repository Editor Flow

After logging on to the Repository Editor, two options are enabled, **Packages** and **Services**. The following figure illustrates the Repository Editor flow to help you determine which button to select.

Figure 3-4 Repository Editor Flow Diagram



Select **Packages** to 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

Select **Services** to access 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

What is a Package?

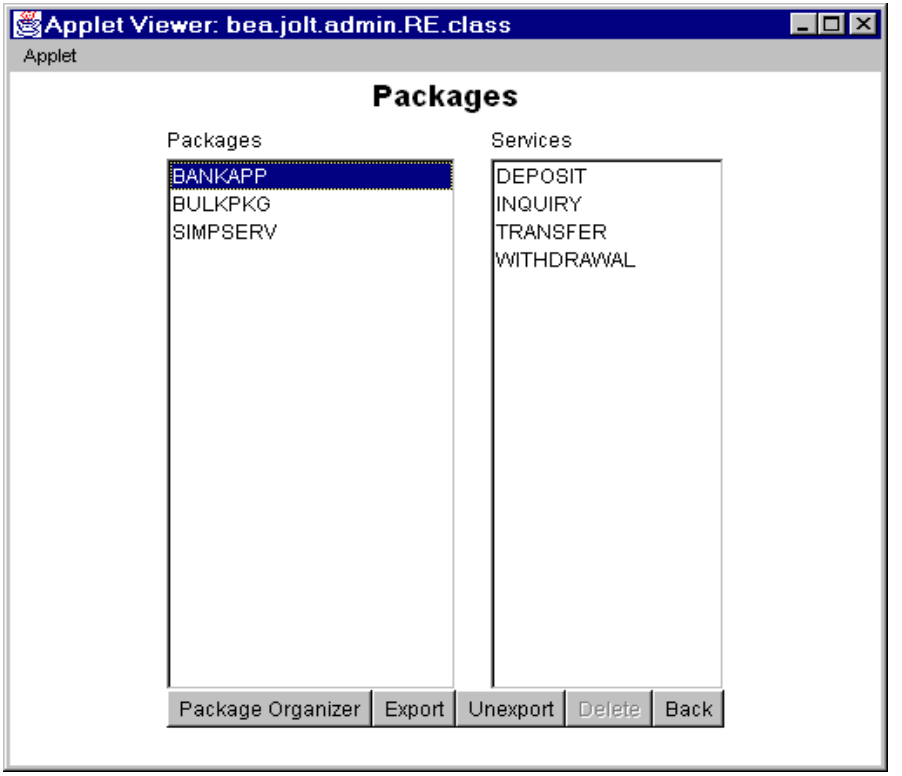
Packages provide a convenient method for grouping services for Jolt administration. A service is comprised of parameters, including pin number, account number, payment, rate, term, age, or Social Security number. The **Packages** button can be used to:

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

The available packages are displayed. When a package is selected, the services contained within a package display.

The following figure is an example of a Packages window.

Figure 3-5 Highlighted Package with Services



Packages Window Description

The following listing describes the Packages window options.

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. Moves the services among the packages or adds a new package.

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).

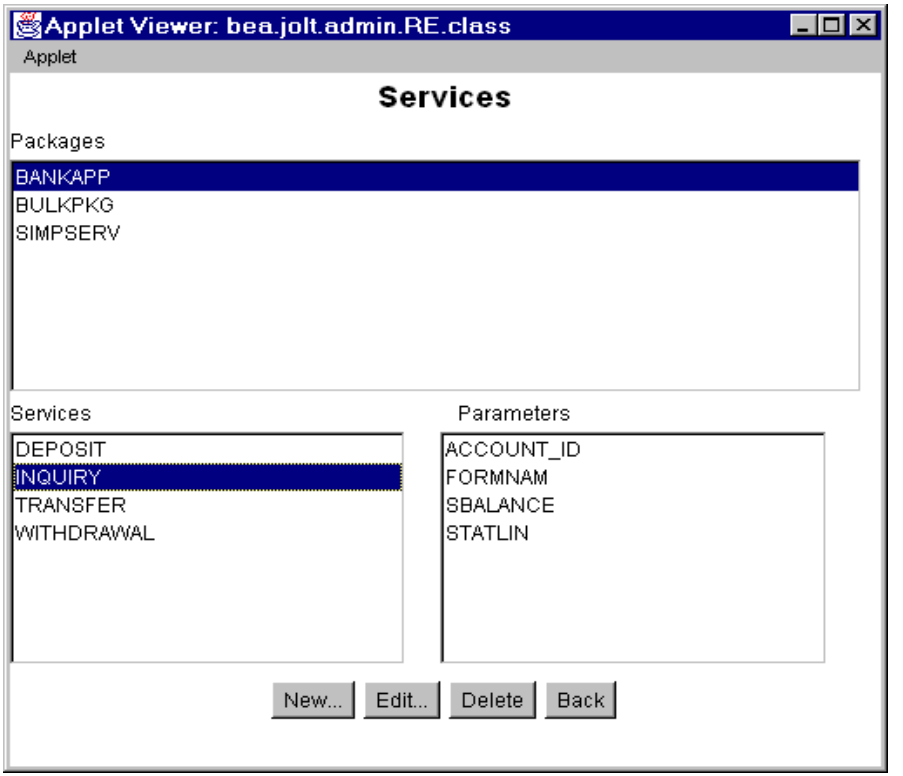
Instructions for Viewing a Package

1. To view the packages, select **Packages** from the Logon window. The Packages window displays.
2. The packages are displayed in the Packages display list. In Figure 3-5, BANKAPP, BULKPKG, and SIMPSERV are the available packages.

What is a Service?

A service is a definition of an available 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 Tuxedo service. Use the Services window to add, edit, or delete services. The following figure is an example of a Services window showing the available services for the package selected.

Figure 3-6 Services Window



Services Window Description

The following table describes the Services window options:

Option	Description
Packages	Lists the services and parameters for the select package. Select the package to add a new service, edit, or delete a service.
Services	Lists a service in the package to edit or delete. Selecting a service displays the parameters within the service.
Parameters	Displays selected service parameters.
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. To view the services, select **Services** from the Logon window.

   The Services window displays.

   The available packages are displayed in the Packages display list.

2. Select a package.

   In the previous figure, BANKAPP is the selected package.

   The available services for the selected package are displayed in the Services display list. In the previous figure, DEPOSIT, INQUIRY, TRANSFER and WITHDRAWAL are the available services for BANKAPP.

Working With Parameters

A service contains parameters, which may include pin number, account number, payment, rate, term, age, or Social Security number. Adding or editing a parameter does not modify or change an existing Tuxedo service. The following figure shows a Services window displaying a selected service and its parameters.

Instructions for Viewing a Parameter

1. Select **Services** from the Logon window to view its parameters.

   The Services window displays.

   View packages in the Packages display list.

2. Select a package to view its available services.

   In the following figure, BANKAPP is the selected package.

   View services in the Services display list.

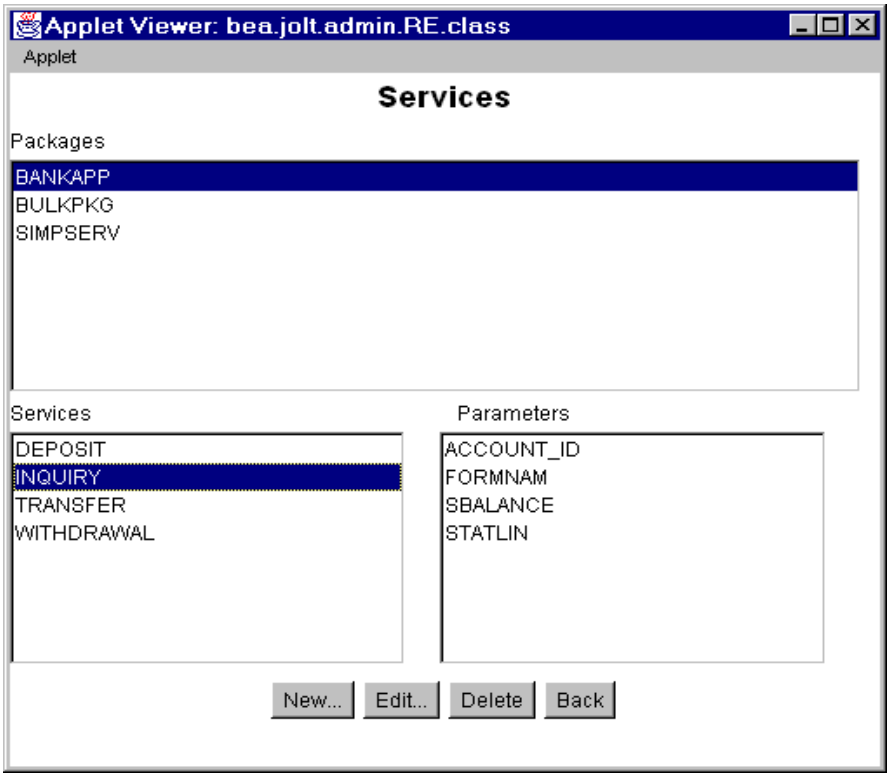
3. Select a service to view its available parameters.

   In the following figure, INQUIRY is the selected service.

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

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

Figure 3-7 Services Window with Parameters



Setting Up Packages and Services

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

- Adding a package
- Adding a service
- Adding a parameter

Saving Your Work

As you are creating and editing services and parameters, it is important to regularly save information to ensure that you do not inadvertently lose any input. Selecting **Save Service** can prevent the need to re-enter information in the event of a system failure.

Caution: Be sure to exercise caution when you are adding or editing 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 at 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 briefly displays on the status line at the bottom of the window.

Adding a Package

If you need to add a new group of services, you must create a new package before adding the services. Figure 3-8 shows how to add a new package, BALANCE, to the Packages listing.

Instructions for Adding a Package

Follow these instructions to add a package:

1. Select **Packages** from the Logon window.

   The Packages window displays.

2. Select **Package Organizer**.

   The Package Organizer window displays. (For a description of the Package Organizer window, see "Package Organizer Description" in this chapter in this chapter.)

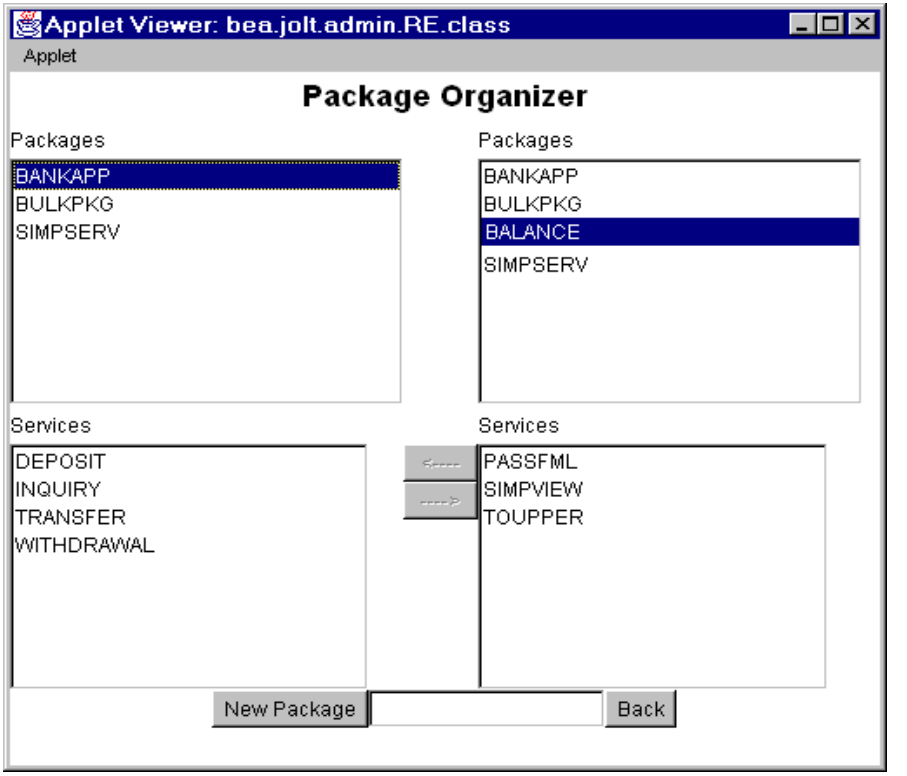
3. Select **New Package** from the Package Organizer window.

   The text field is activated.

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

   The new name (in the following figure, BALANCE) is displayed in the Packages display list in random order.

Figure 3-8 Adding a New Package

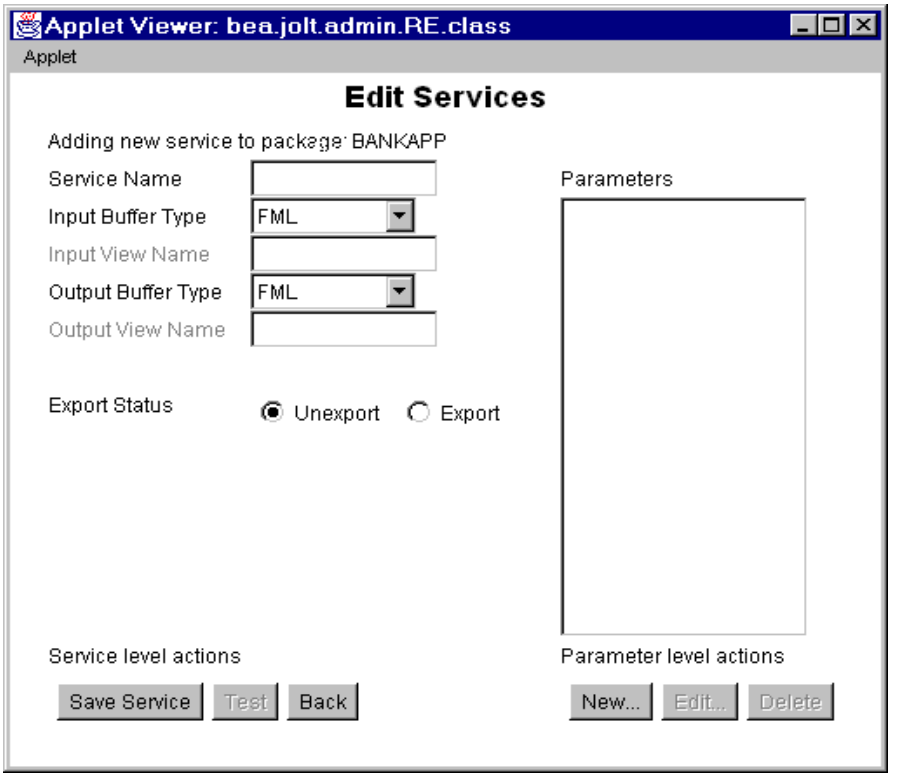


Adding a Service

Services are definitions of available Tuxedo services and can only be a part of a Jolt package. You are not required to create a new package before creating a new service; however, you must create the service as a part of a package, even if it is moved to a different package at a later date.

The Repository Editor accepts the new service name exactly as it is typed (that is, all capital letters, abbreviations, misspellings are accepted). Service names must not exceed 30 characters. The following figure shows an Adding New Service window.

Figure 3-9 Edit Services: Adding New Service Window



Adding a Service Window Description

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

	Option	Description
Edit Services Selections	Service Name	The name of the new service to be added to the Repository.
	Input Buffer Type/Output Buffer Type	VIEW - a 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 - an array of uninterrupted binary data that is neither encoded nor decoded during transmission; it may contain null characters. X_OCTET is equivalent. FML - a 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 - a character array terminated by a null character that is encoded or decoded.
	Input View Name/Output View Name	A 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	Lists current status of the service. EXPORT or UNEXPORT status is displayed. UNEXPORT is the default.
Service Level Actions	Save Service	This command button saves 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	This command button returns you to the previous window.
Parameters	Parameters	List of service parameters to edit or delete.

Parameter Level Actions	New	This command button adds new parameters to the service.
	Edit	An existing parameter can be edited. This command button is disabled until a new parameter is selected.
	Delete	This command button deletes a parameter. This option is disabled until a parameter is selected.

Instructions for Adding a Service

To add a service, follow these instructions:

1. From the Logon window, select **Services**.
2. Select the package where the service is going to be added.

   If you are uncertain which package should contain the new service, select a package and use the Package Organizer to move the service to a different package. (See "Grouping Services Using the Package Organizer" for additional information.)
3. Select **New** from the Services window.

   The Edit Services window is displayed.
4. Select the **Service Name** text field to activate it.
5. Type the service name.
6. Select the buffer type.

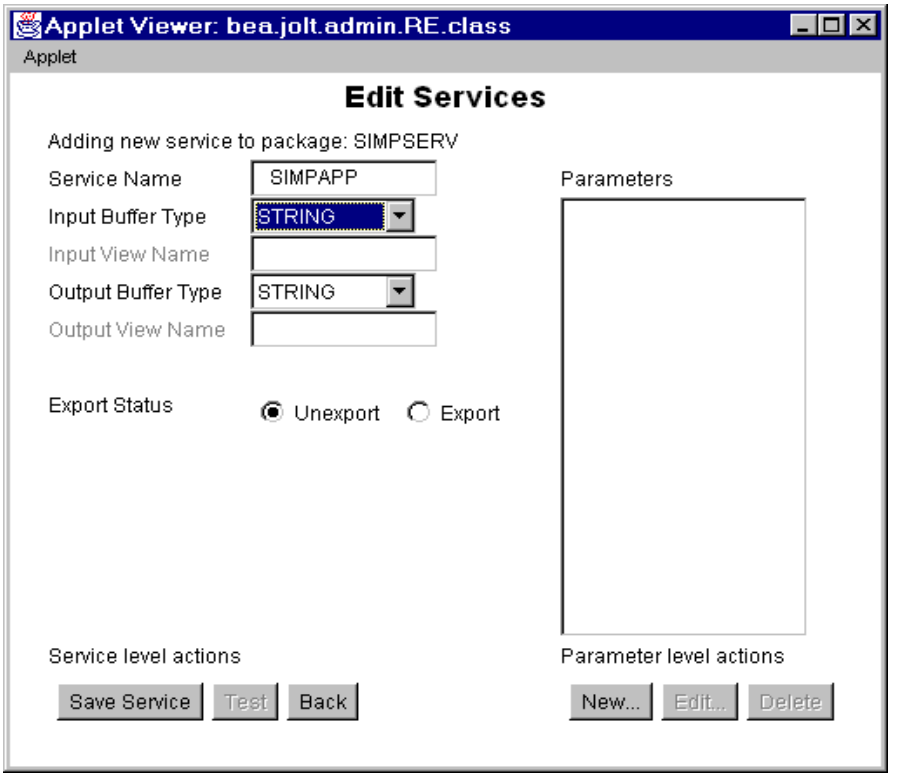
   Although the same buffer type selected for the Input Buffer is automatically selected for the Output Buffer, you can change the Output Buffer type to a different buffer type.
7. If VIEW or VIEW32 is selected, type the Input View Name and Output View Name in the accompanying text field.
8. If another buffer type is selected, the Input View Name and Output View Name text fields are disabled.
9. If CARRAY or STRING is selected, refer to "Selecting CARRAY or STRING as a Service Buffer Type" in this chapter for additional instructions.
10. 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" and "Selecting CARRAY or STRING as a Parameter Data Type" in this chapter. For further information, refer to "Using the Jolt Class Library"

The following figure is an example of the Edit Services window with STRING as the selected buffer type for the service.

Figure 3-10 Edit Services Window with STRING as the Selected Buffer Type

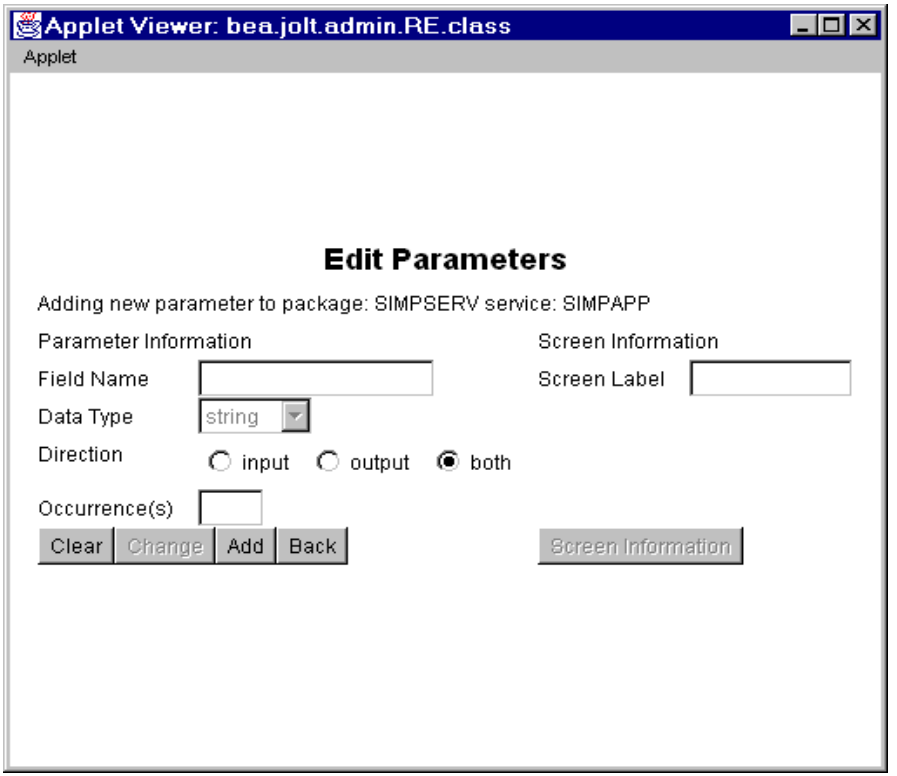


Adding a Parameter

Selecting **New Parameter** from the Edit Services window brings up the Edit Parameters window. Review the features in the following figure. Use this window to enter the parameter and window information for a service.

The following figure is an example of the Edit Parameters window used to add a new parameter.

Figure 3-11 Adding a Parameter Window



Parameters Window Description

The following listing describes the Edit Parameters window options.

Option	Description
Field Name	Adds the field name (for example, asset, inventory).
Type	List 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	Lists 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 Tuxedo.
Clear	This command button clears the fields of the window.
Change	This command button is disabled while new parameters are added.
Add	This command button adds new parameters to the service. The parameters are saved when the service is saved.
Back	This command button 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. Select the **Occurrences** text field to activate it, and then enter the number of occurrences.
4. Specify a direction by selecting the **input**, **output**, or **both** radio buttons.
5. Select **Add** to append the information. **Add** does not save the parameter.
6. Select **Save Service** to save the parameter as a part of the service.

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

7. Select **Back** to return to the previous window.

Selecting CARRAY or STRING as a Parameter Data Type

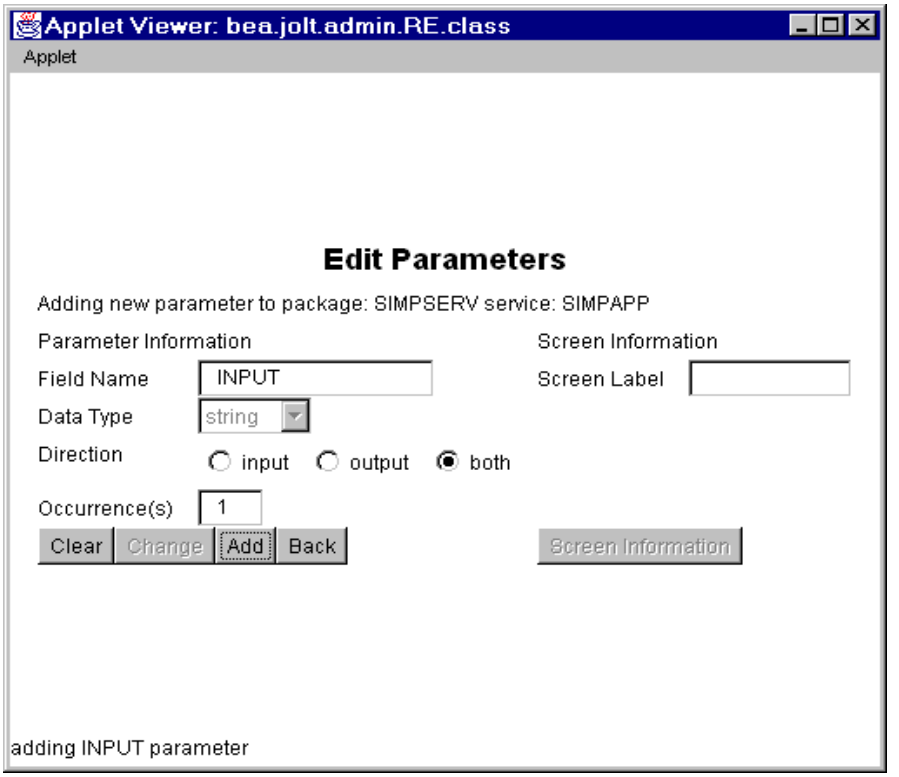
If CARRAY or STRING is the selected buffer type for a new service, only carray or string can be added as the data type for the accompanying parameters.

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” and “Selecting CARRAY or STRING as a Service Buffer Type” in this chapter. For further information, refer to "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 **Type** defaults to string and does not allow you to modify that particular data type. The **Field Name** can be any name.

Figure 3-12 Edit Parameters Window with string as the Data Type



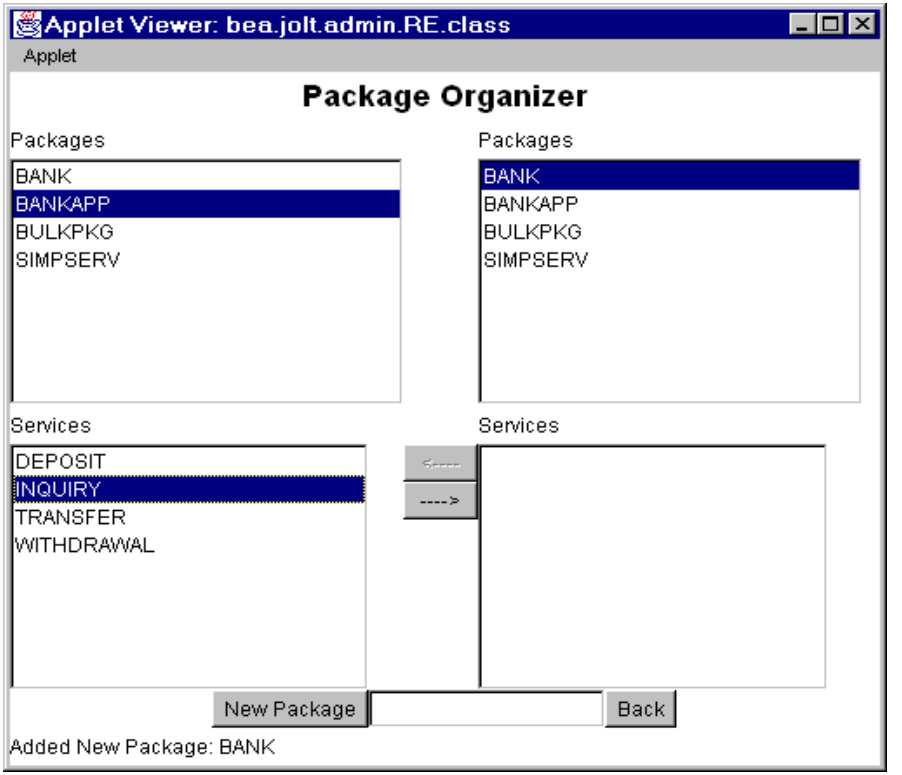
Grouping Services Using the Package Organizer

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).

The Package Organizer arrow buttons allow you 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 a of Package Organizer window with a service selected for transfer to another package.

Figure 3-13 Example of a Selected Service



Package Organizer Description

The following table describes the options for the Package Organizer window:

Option	Description
Available Packages (left display list)	Lists packages available where the service to be moved currently resides.
Available Packages (right display list)	Lists packages available as destinations for the service you are moving.
Services (left display list)	Lists available services for the highlighted package that can be moved.

Services (right display list)	Lists available services of the highlighted package that have been 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. Select the package containing the services to be moved from the Packages left display window to the right display window.

   In the previous figure, BANKAPP is the selected package.

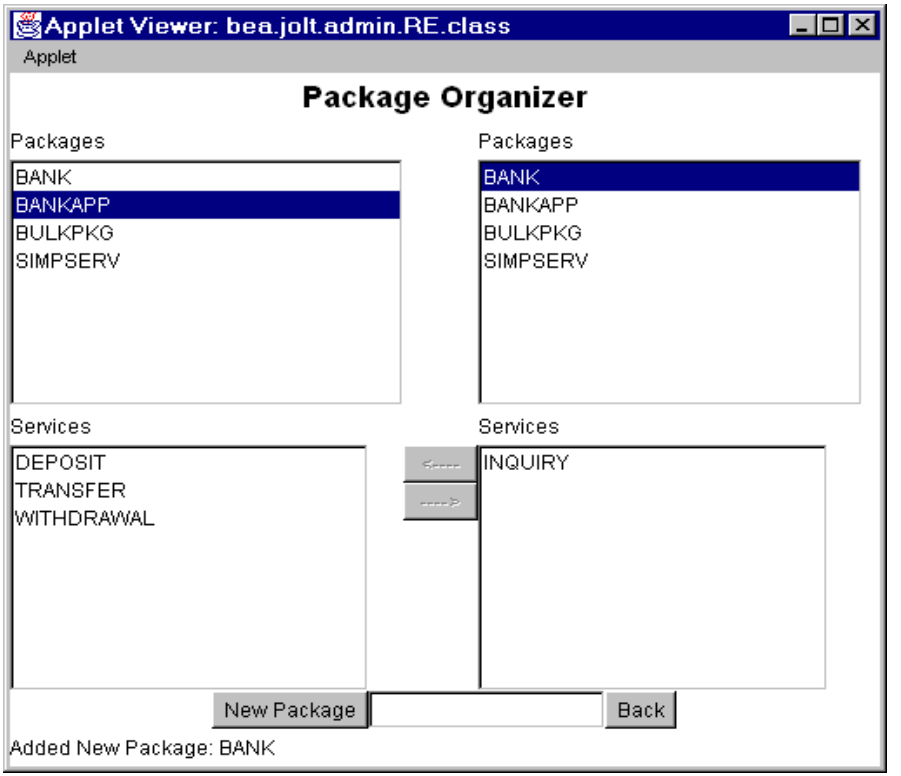
2. Select the service to be moved from the Services left display window to the right display window.

   In the previous figure, INQUIRY is the selected service in the BANKAPP package.

3. 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 3-14 Example of a Moved Service



4. To move the services between the packages, select the left arrow **(<—)** or right arrow **(—>)**.

   These keys are activated only when both packages 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 shows how the Repository Editor moves the INQUIRY service 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/Parameters

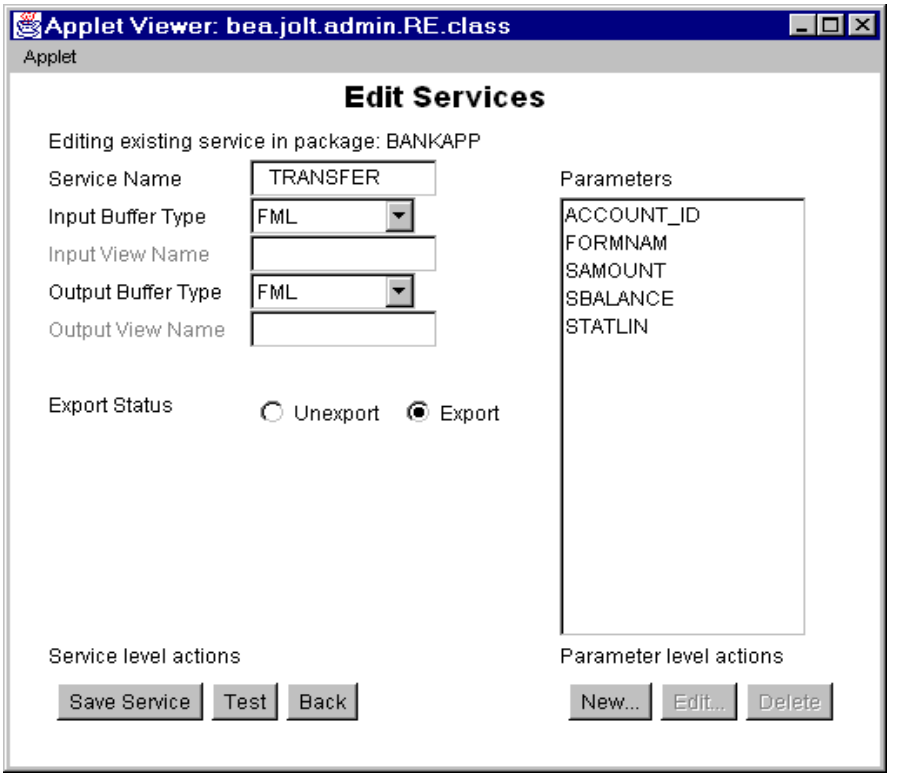
If a package, service, or parameter requires any modifications, you can make the following changes:

- Edit a service
- Edit a parameter
- Delete a parameter/service/package

Editing a Service

Edit an existing service name, 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" in this chapter. The following figure is an example of the Edit Services window.

Figure 3-15 Edit Services Window



Instructions for Editing a Service

Follow these instructions to edit a service:

1. Select the package containing the service that requires editing from the Services window.
2. Select the service to edit.

   The parameters are displayed in the parameters display list.
3. Select **Edit**.

   The Edit Services window displays.

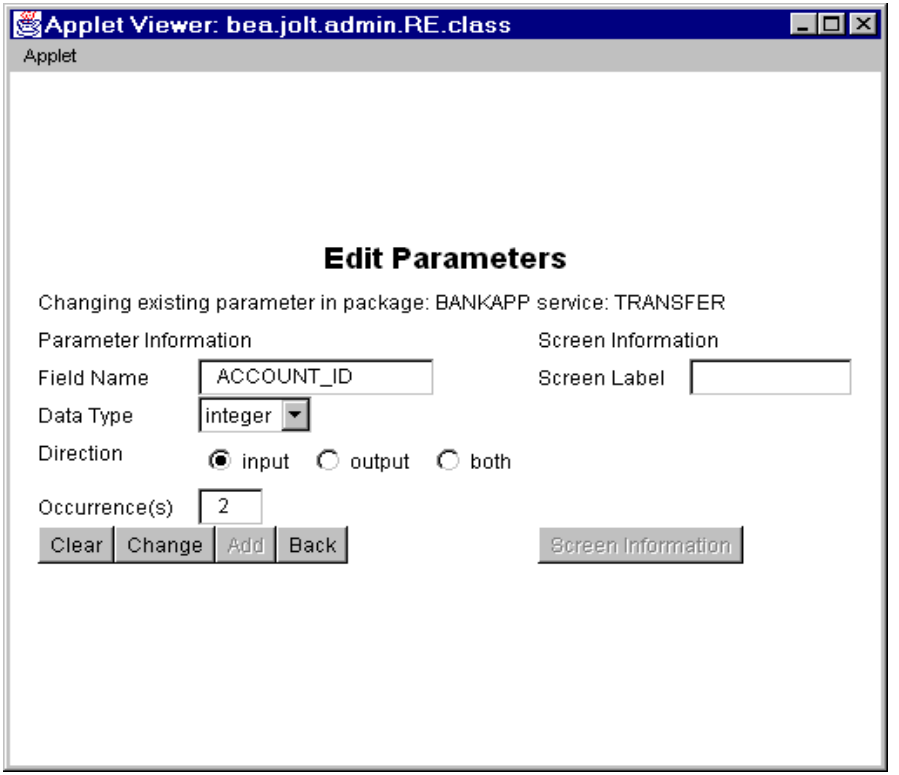
4. Type or select the new information and select **Save Service**.

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 3-16 Edit Parameters Window



Editing a Parameter

To change a parameter, follow these instructions:

1. Select the parameter in the Parameters window and select **Edit Parameters**.

   The "Edit Parameters: Changing Existing Parameter" window displays.

2. Type the new information and select **Change**.
3. Select **Back** to return to the previous window.

Deleting Parameters/Services/Packages

This section details the necessary steps 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 instructions.

1. Highlight the parameter you want to delete in the Parameters display list and select **Delete Parameter**.
2. Select **Back** to return to the previous window.

Deleting a Service

Determine which services to delete and follow these instructions. Make sure that all parameters within this service are deleted before selecting this option.

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

   The Packages window displays.

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

3. Select the service you want to delete.

   Delete is enabled.

4. Select **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. To delete packages, select **Packages** from the Logon window. The Packages window displays.
2. Select a package.
3. Select **Delete**.

   The package is deleted.

Making a Service Available to the Jolt Client

To make a service available to a Jolt client, you must 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 section discusses:

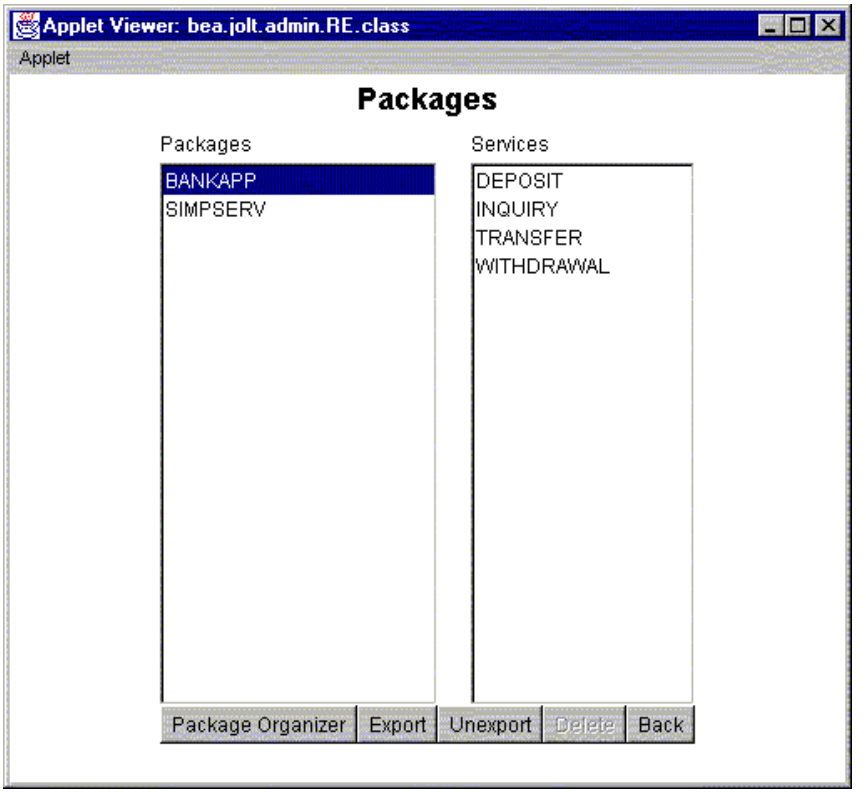
- Exporting/Unexporting services
- Reviewing the Export/Unexport status

Exporting/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. From there you can **Export** and **Unexport** services.

Figure 3-17 Export and Unexport Buttons



Exporting/Unexporting a Service

Follow the instructions below to export or unexport a service.

1. Select **Packages** from the Logon window.

   The Packages window displays.

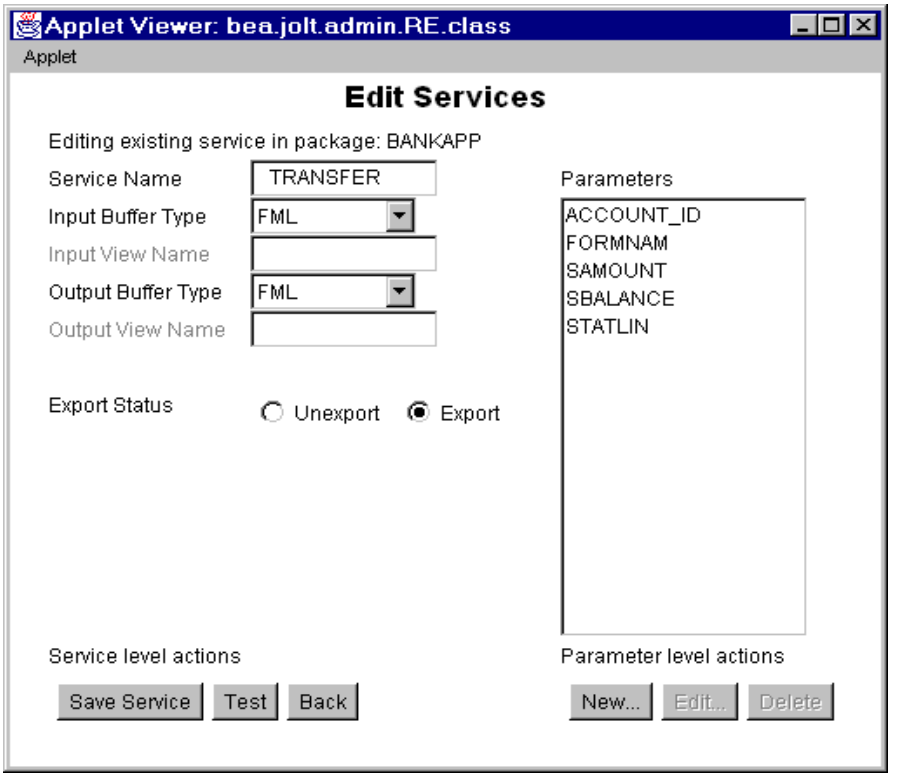
2. Select a package. **Export** and **Unexport** are enabled.
3. To make services available, select **Export**.
4. To make services unavailable, select **Unexport**.

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

Reviewing the Exported/Unexported Status

When a service is exported or unexported, you can review its status from the Edit Services window. The following figure shows the current status as EXPORTED.

Figure 3-18 Exported/Unexported Status



Reviewing the Exported/Unexported Status

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

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

   The Services window displays.

2. To find out if a service has been exported or unexported, check its status by selecting a package from the Package display list.

   The Services display list is enabled with a listing of services for the selected package.

3. Select the desired service.

4. Select **Edit**. The Edit Services window displays with the **Current Status** of the service as EXPORTED or UNEXPORTED.

Testing a Service

Before they are made available to Jolt clients, a service and its parameters should be tested to ensure that they are functioning properly. Services that are currently available can be tested without making changes to the services and parameters.

Note: The Jolt Repository Editor allows you to test an existing 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 section explains the following:

- Jolt Repository Editor Service Test Window
- Testing a Service Instructions

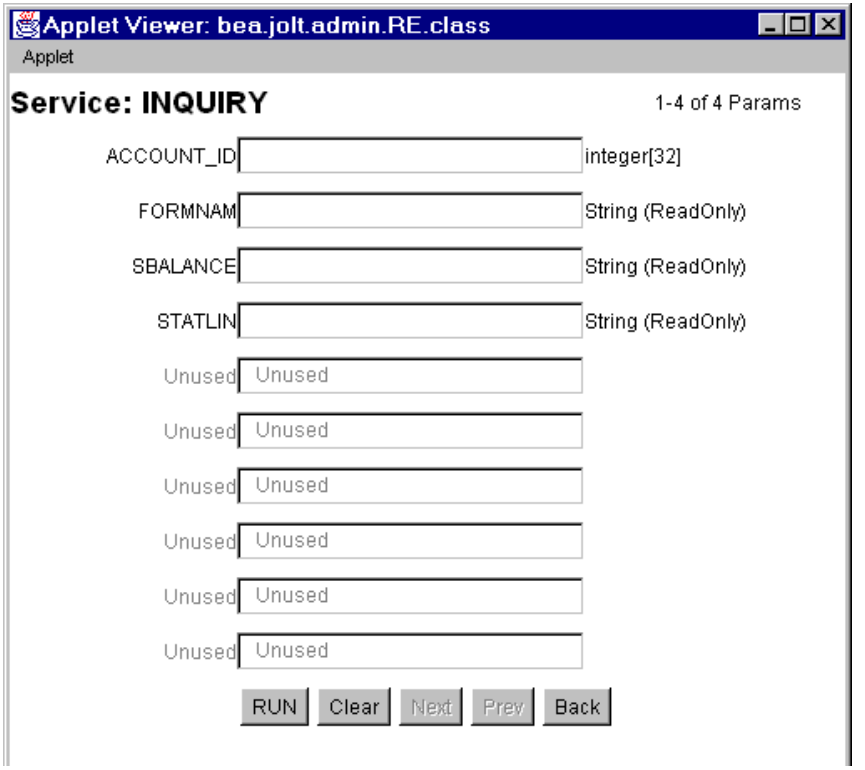
Repository Editor Service Test Window

Test the service to ensure that the parameter information is accurate. Although **Test** is enabled when parameters are not added to the service, the Service Test window (the following figure) displays the parameter fields as “unused” and they are disabled. A service can only be tested when the corresponding Tuxedo server is running for the service being tested.

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 shows an example of a Service Test window with writable and read-only text fields.

Figure 3-19 Sample Service Test Window



Service Test Window Description

The following table details the Service Test window.

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.

Table 3-2

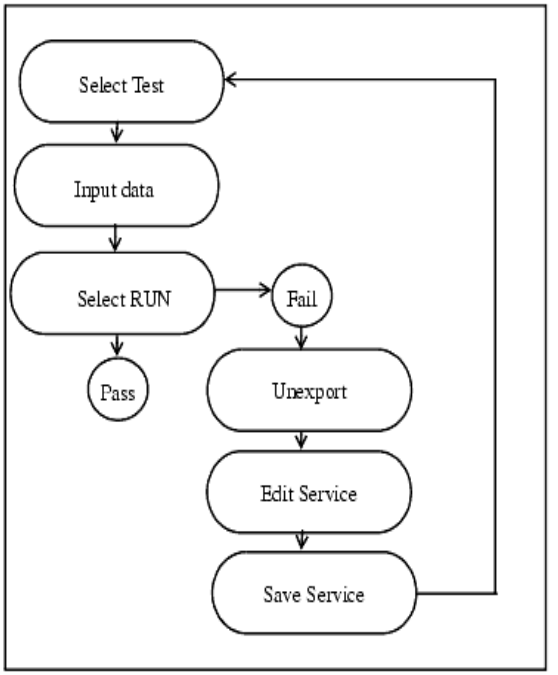
Option	Description
Service	Displays the name of the tested service (read-only).
Parameters displayed	Tracks the parameters displayed in the window (read-only).
Parameter text fields	The parameter information text entry field. These fields are writable or read-only. Disabled if read-only.
RUN	Runs the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

Test a service to ensure that all service and parameter information is correct. 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.

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

Figure 3-20 Test Service Flow



Testing a Service

Follow these instructions to test a service.

1. Select **Services** from the Logon window to display the Services window.
2. Select the package and the service to test.

3. Select **Edit** to access the Edit Services window.
4. Select **Test** to access the Service test window.
5. Input data in the Service test window parameter text field.
6. Select **RUN**.

   The status line displays the message, "Run Completed OK," if the test passes, or "Call Failed," if the test fails. See "Some Reasons Why a Test Might Fail" or the "Repository Editor Troubleshooting Table" for additional Repository Editor troubleshooting information.

Follow these instructions if editing is required to pass the test.

1. Return to the Repository Editor logon window and select **Packages**.
2. Select the package with the services to be retested.
3. Select **Unexport**.
4. Select **Back** to return to the Logon window.
5. Select **Services** to display the Services window.
6. Select the package and the service that requires editing and select **Edit**.
7. Edit the service.
8. Save the service, select **Test**, and repeat steps 5 and 6 from previous list.

Some Reasons Why a Test Might Fail

Here are some reasons why a service test might fail and possible solutions.

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

Troubleshooting

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

Table 3-3 Repository Editor Troubleshooting Table

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**)	Check and make sure that: ♦ Your Server name is correct (and accessible from your machine). Check that the port number is the correct port. There must be a JSL or JRLY 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 user names and passwords. ♦ If the applet was loaded through http, the Web server, JRLY and the Jolt server must be on the same machine (i.e., the Server name entered into the Repository Editor must be the same machine as the one used in the URL to download the applet).

Table 3-3 Repository Editor Troubleshooting Table

If . . .	Then . . .
You cannot start the Repository Editor	If you are running the editor in a browser and downloading the 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 (this is 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)
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.

Table 3-3 Repository Editor Troubleshooting Table

If . . .	Then . . .
You cannot test services	♦ Check that the service is available. ♦ Verify the service definition matches the service. ♦ If 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 is unable to process data conversion and you will receive the message "ServiceException: TPEJOLT data conversion failed." ♦ Check the ULOG file for any additional diagnostic messages.

Repository Enhancements for Jolt

The Jolt Repository uses the FML32 buffer type, which increases the internal buffer size beyond 64K bytes.

Additionally, the JREPSVR and the Jolt Server (JSH) support the following XATMI buffer types:

- X_OCTET
- X_C_TYPE
- X_COMMON

4 Using the Jolt Class Library

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

“Usingthe Jolt Class Library” covers the following topics:

- Class Library Functionality Overview
- Jolt Object Relationships
- Jolt Class Functionality
- Jolt Class Library Walk-through
- Using Tuxedo Buffer Types with Jolt
- Multithreaded Applications
- Event Subscription and Notifications
- Clearing Parameter Values
- Reusing Objects
- Application Deployment and Localization

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 provides the Tuxedo application developer with 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 Listing 4-1.

Java Applications vs. Java Applets

Java programs that run in a browser are called "applets." Applets are intended to be small, easily downloaded parts of an overall application that perform specific functions. Many popular browsers impose limitations on the capabilities of Java applets for the purpose of providing a high degree of security for the users of the browser. The following are some of the restrictions imposed on applets:

- 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 where it originated; it cannot make other network connections, not even to the client machine.

Programming workarounds exist for most of the 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 time-outs for continuous and intermittent client network connections.
- Features methods that allow a Jolt client to subscribe to and receive event-based notifications.

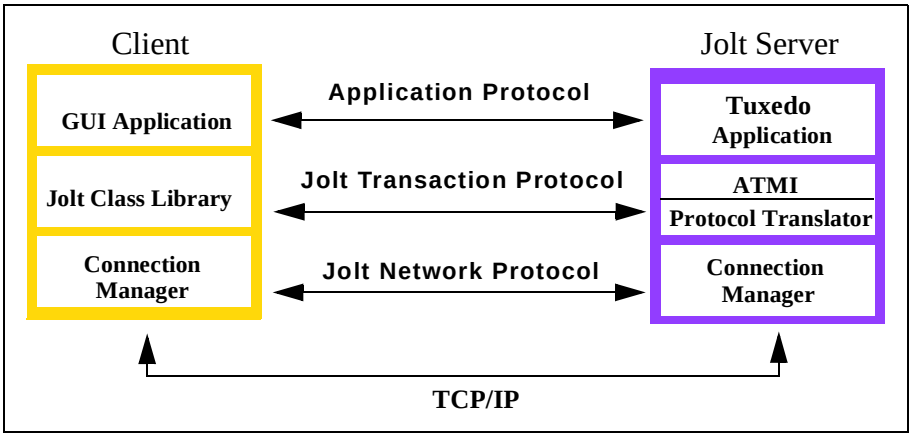
Error and Exception Handling

The Jolt Class Library returns both Jolt interpreter and 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 1.2 Online API Javadoc 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. Figure 4-1 illustrates the client/server relationship between a Jolt program and the Jolt Server.

Figure 4-1 Jolt Client/Server Relationship



As illustrated in the diagram, 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 4-1 Jolt Client/Server Interaction

	Jolt Client	Jolt Server
1	`attr=new JoltSessionAttributes();` `attr.setString(attr.APPADDRESS, "//myhost:8000");`	Binds the client to the Tuxedo environment
2	`session=new JoltSession(attr, username, userRole, userPassword, appPassword);`	Logs the client onto 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 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 Tuxedo

The following tasks, which summarize the interaction shown in Table 4-1,are the steps involved in beginning a transaction:

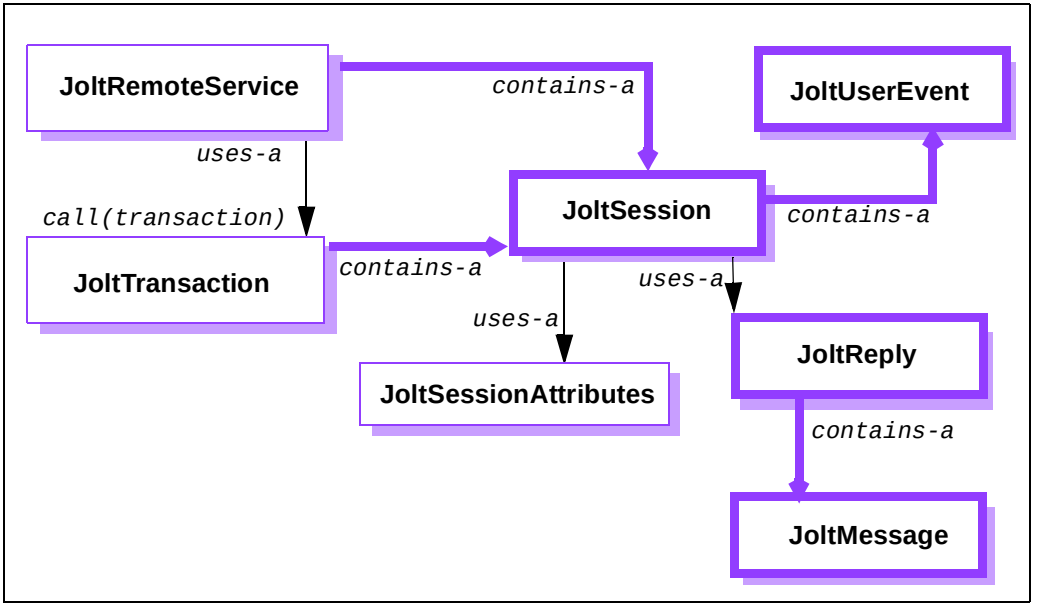
1. Bind the client to the Tuxedo environment using the `JoltSessionAttributes` class.
2. Establish a session.
3. Set variables.
4. Perform the necessary transaction processing.
5. Log the client off of the Tuxedo system.

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. The next section 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 4-2 Jolt Object Relationships



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

Contains-a relationship. At the class level an object can contain other objects. For example, a JoltTransaction stores (or contains) a JoltSession object.

Is-a relationship. The is-a relationship usually occurs at the class instance or sub-object level and denotes that the object is an instance of a particular object.

Uses-a relationship. An object can use another object without containing it. For example, a JoltSession can use the JoltSessionAttributes object to obtain the host and port information.

Jolt Class Functionality

Jolt classes are used to perform the basic functions of transaction processing: log on/log off, synchronous service calling, transaction begin, commit, rollback and subscribe to events or unsolicited messages. The following sections describe how the Jolt classes are used to perform these functions.

Logon/Logoff

The client application must log on to the 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 Tuxedo System.

The JoltSessionAttributes class is used to contain the connection properties to a Jolt/Tuxedo system and contains various properties about the Jolt/Tuxedo System. To establish a connection, the client application must create an instance of the JoltSession class. This instance is the JoltSession object. By instantiating a JoltSession object, users log on to Jolt/Tuxedo or log off by calling the endSession method.

Synchronous Service Calling

Transaction activities such as requests and replies are handled through the use of a JoltRemoteService object (an instance of the JoltRemoteService class). Each JoltRemoteService object refers to an exported 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, simply:

- Set the input parameters
- Invoke the service
- 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. Since 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.

Since Jolt is designed for a multithreaded environment, you can invoke multiple JoltRemoteService objects simultaneously by using the Java multithreading capability. Refer to "Multithreaded Applications" in this chapter 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 time out 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.

Jolt Class Library Walk-through

The example code provided in Listing 4-1 shows how to use the Jolt Class Library and includes the use of the JoltSessionAttributes, JoltSession, JoltRemoteService, and JoltTransaction classes.

The example combines two user-defined 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 basic steps of the transaction process shown in the example are as follows:

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 will be 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 has been 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 4-1 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 Tuxedo Buffer Types with Jolt

Jolt supports the following built-in Tuxedo buffer types:

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

Note: X_OCTET is used identically to CARRAY.

X_COMMON and X_C_TYPE are used identically to VIEW.

For information about all the Tuxedo typed buffers, data types, and buffer types, refer to the *Tuxedo System Programmer's Guide, Volume 1* and the *Tuxedo System Reference Manual*.

Of the Tuxedo built-in buffer types, the Jolt application programmer should be particularly aware of how Jolt handles the CARRAY (character array) and STRING built-in buffer types. The CARRAY type is used to handle data opaquely, (for example, the characters of a CARRAY data type are not interpreted in any way). No data conversion is performed between a Jolt client and Tuxedo service.

For example, if a 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 Tuxedo service. If the Tuxedo service is run on a machine whose processor uses little-endian byte-ordering (for example, Intel processors), the Tuxedo service must convert the data properly before the data can be used.

Using the STRING Buffer Type

The STRING buffer type is a collection of characters. STRING consists of non-null characters and is terminated by a null character. 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.

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.

The following `ToUpper` code fragment, Listing 4-2, illustrates how Jolt works with a service whose buffer type is STRING. The TOUPPER Tuxedo Service is available in the Tuxedo `simpapp` example.

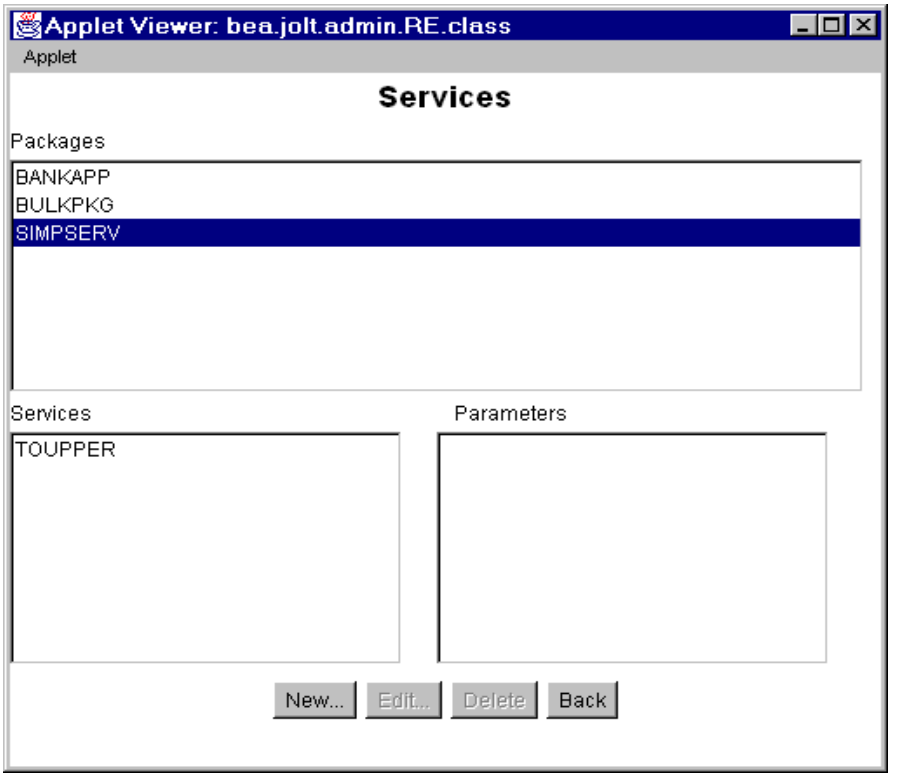
Define TOUPPER in the Repository Editor

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

Note: If you are not familiar with using the Jolt Repository Editor, refer to "Using the Jolt Repository Editor" for more information about defining your services and adding new parameters.

1. Using the Jolt Repository Editor, define the TOUPPER service for the SIMPSERV package.

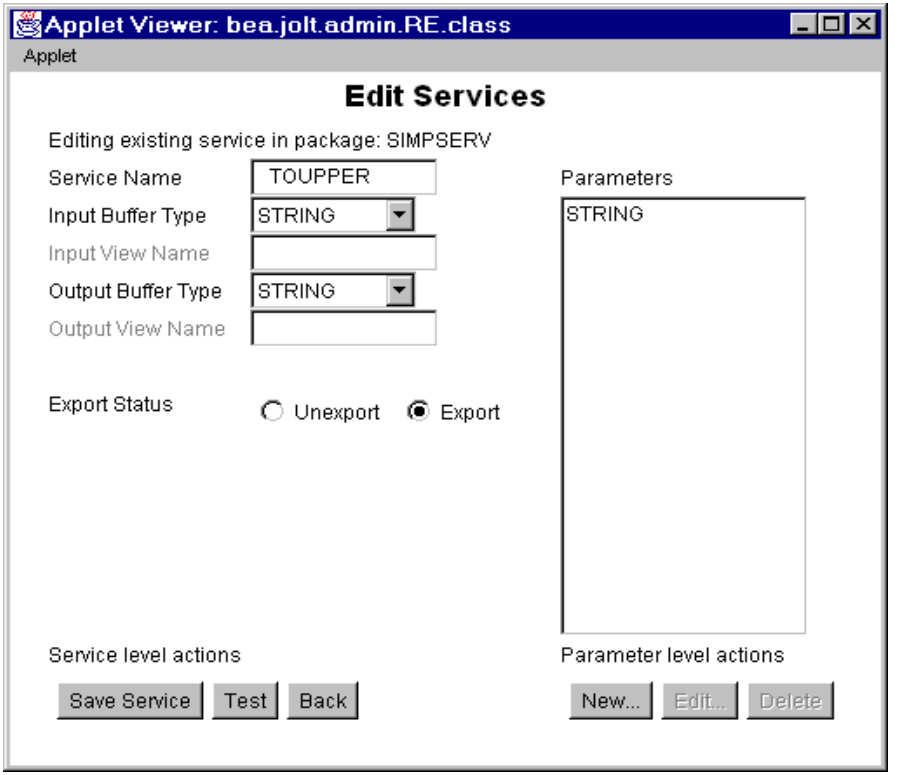
Figure 4-3 Add a TOUPPER Service



2. For the TOUPPER service, define an input buffer type of STRING and an output buffer type of STRING. (See the following figure.)

3. Define only one parameter for the `TOUPPER` service named STRING that is both an input and an output parameter.

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



ToUpper.java Client Code

The `ToUpper.java` Java code fragment in the following listing illustrates how Jolt works with a service whose buffer type is STRING. The example shows a Jolt client using a STRING buffer to pass data to a server. The Tuxedo server would take the buffer, convert the string to all uppercase letters and pass the string back to the client. The following example assumes that a session object was already instantiated.

Listing 4-2 Use of the STRING buffer type (ToUpper.java)

```
/* Copyright 1996 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

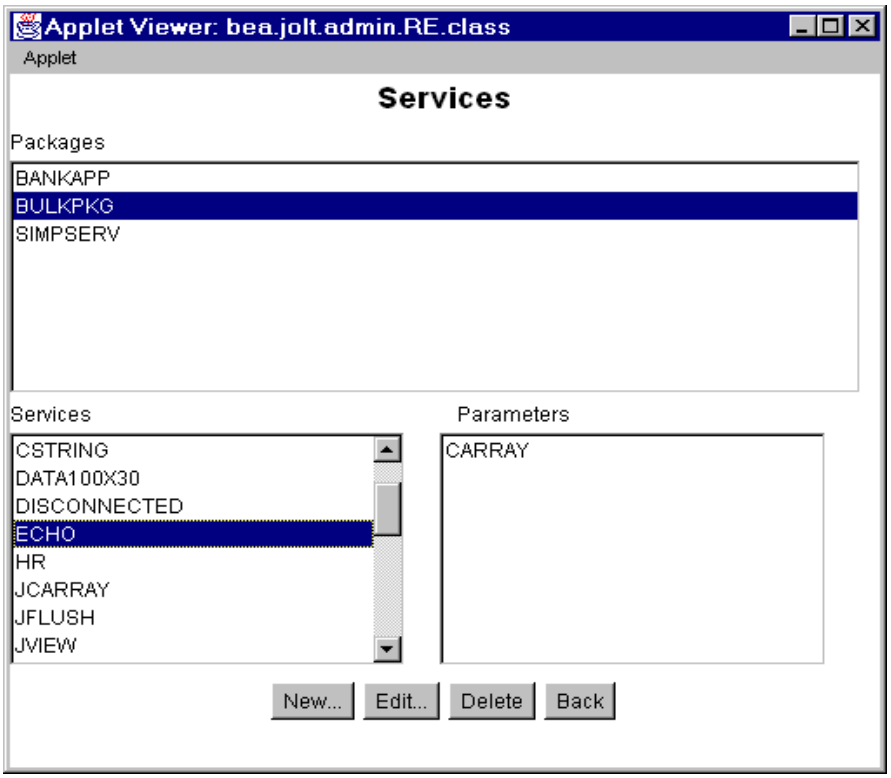
The CARRAY buffer type is a simple character array buffer type that is built into the Tuxedo system. With the CARRAY buffer type, because the system does not interpret the data (although the data type is known) there is no way of determining how much data to transmit during an operation. The application is always required to specify a length when passing this buffer type.

Note: X_OCTET is used identically to CARRAY.

Define ECHO in the Repository Editor

Before running the example in the "Add a TOUPPER Service" figure, you must write and boot an ECHO Tuxedo service. The ECHO service takes a buffer and passes it back to the Jolt client. You also need to use the Jolt Repository Editor to define the ECHO service.

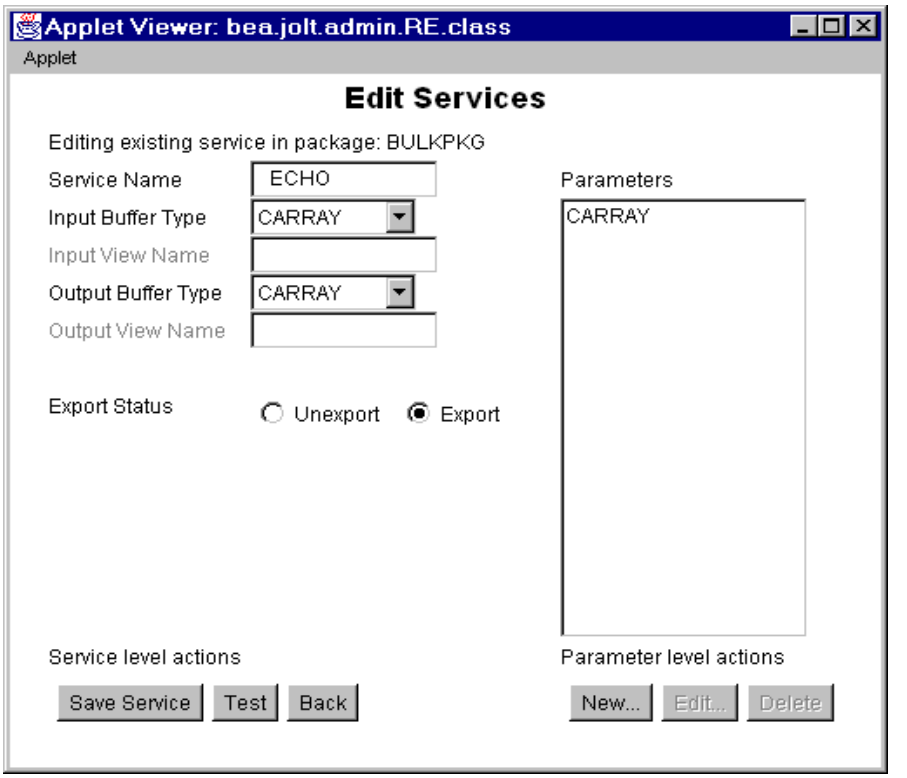
Figure 4-5 Add ECHO Service



In the Repository Editor add the ECHO service as follows:

1. Add a service named ECHO whose buffer type is CARRAY.
2. Define the input buffer type and output buffer type as CARRAY for the ECHO service.

Figure 4-6 Edit ECHO Service



3. Define the ECHO service with only one parameter named CARRAY that 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.

tryOnCARRAY.java Client Code

The code in the following listing illustrates how Jolt works with a service whose buffer type is CARRAY. Since Jolt does not look into the CARRAY data stream, it is the programmer's responsibility to have the matching data format between the Jolt client and the CARRAY service. The following example assumes that a session object was already instantiated.

Listing 4-3 CARRAY Buffer Type

```
/* 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 runtime determines the field location and data type.

FML is especially suited for use with Jolt clients as the client and server code may be in two languages (for example, Java and C), the client/server platforms may have different data type specifications, or the interface between the client and the server changes 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 is a Jolt client that contains a PASSFML service.
- The “tryOnFml.f16 Field Definitions” code example is a Tuxedo FML field definitions table used by the PASSFML service.
- The “tryOnFml.c” code example 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. The following example assumes that a session object was already instantiated.

Listing 4-4 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`, show FML field definitions for the `tryOnFml.java` example.

Listing 4-5 tryOnFml.f16 Field Definitions

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

Define PASSFML in the Repository Editor

The"Using the STRING Buffer Type" figure illustrated the SIMPAPP package with two services. The TOUPPER service was used to illustrate the STRING typed buffer. The other service in SIMPAPP package is the PASSFML service. This service 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: If you are not familiar with using the Jolt Repository Editor, refer to "Using the Jolt Repository Editor" for more information about defining a service.

1. Using the Jolt Repository Editor, define the PASSFML service with an input buffer type of FML and an output buffer type of FML.

   The following figure illustrates the Jolt Repository Edit Services window with the PASSFML service.

2. Define the input buffer type and output buffer type as FML for the PASSFML service.

3. Define the parameters for the PASSFML service. In this example, the parameters are: INPUTSTRING, OUTPUTINT, INPUTINT, OUTPUTSTRING, OUTPUTFLOAT, INPUTFLOAT.

Figure 4-7 Edit the PASSFML Service

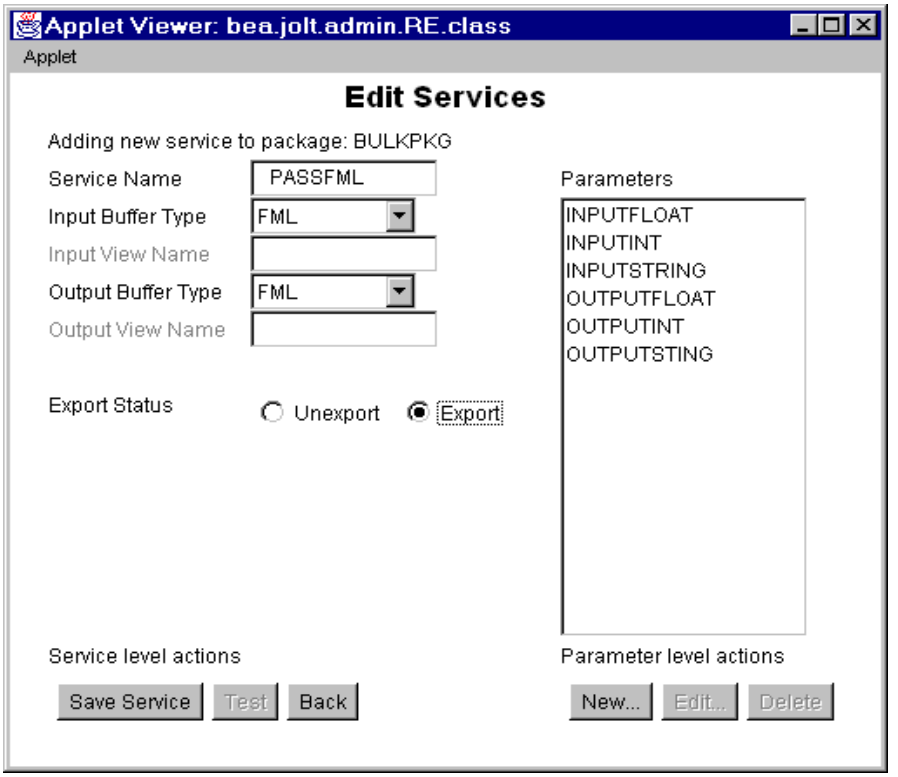
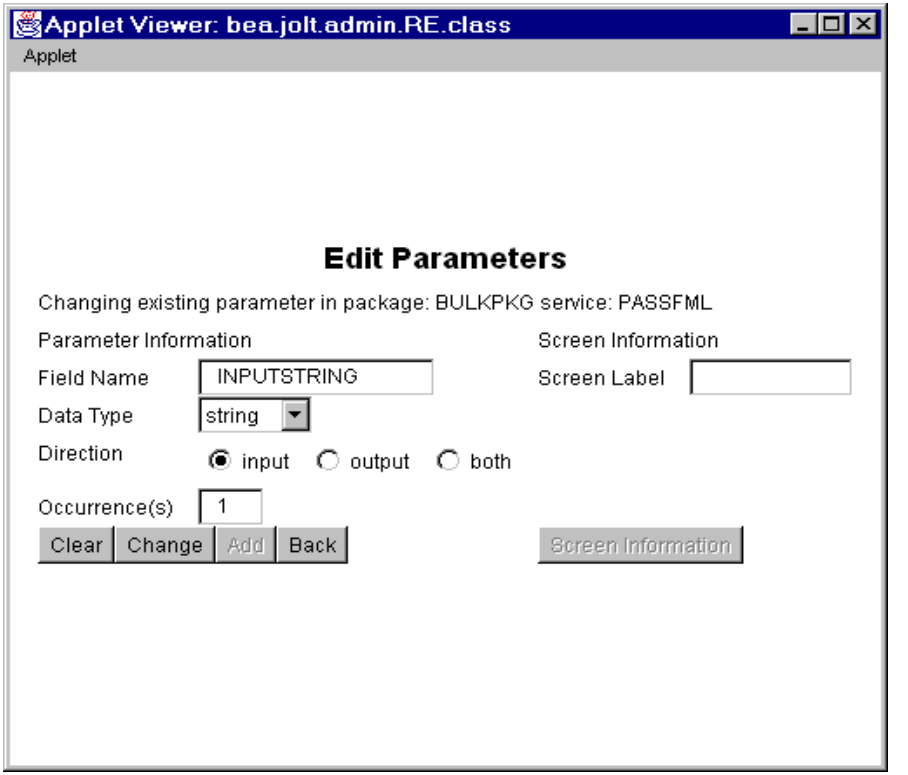


Figure 4-8 Edit the PASSFML Parameters



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 4-6 tryOnFml.c Code Example

```
/*
 * tryOnFml.c
 *
 * Copyright (c) 1997 BEA Systems, Inc.  All rights reserved
 *
```

```
 * Contains the PASSFML 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 )
{
FLDLENlen;
FBFR*svcinfo = (FBFR *) rqst->data;
charinputString[256];
longinputInt;
floatinputFloat;
FBFR*fml_ptr;
intrt;
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 Tuxedo typed buffer. The VIEW buffer provides a way to use C structures and COBOL records with the Tuxedo system. The VIEW typed buffer enables the 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. Since the 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 example consists of three parts:

- The `simpview.java` Jolt client that contains the code used to connect to Tuxedo and uses the VIEW buffer type (in the following listing)
- The `simpview.v16` file that contains the Tuxedo VIEW field definitions
- The `simpview.c` code sample containing 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.

simpview.java Client Code

The following listing 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. Since all Jolt exceptions are derived from `java.lang.RunTimeException`, the Java Virtual Machine (JVM) will catch these exceptions if the application does not. (A well-written application would 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` Tuxedo application. This service takes the data from the client VIEW buffer, creates a new buffer and passes it back to the client as a new VIEW buffer. The following example assumes that a session object has already been instantiated.

Listing 4-7 simpview.java Code Example

```
/* Copyright 1997 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 Tuxedo service "SIMPVIEW"
ViewSvc = new JoltRemoteService("SIMPVIEW",session);
// Set the input parametes 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 following entries show the Tuxedo VIEW field definitions for the `simpview.java` example.

Listing 4-8 simpview.v16 Field Definitions

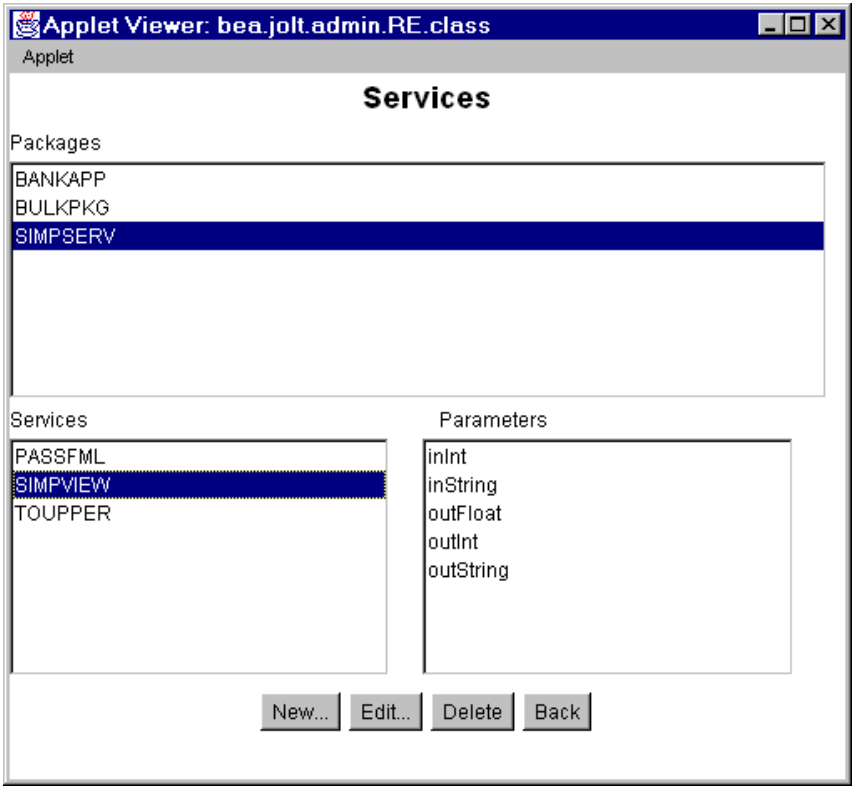
```
#
# VIEW for SIMPVIEW. This view is used for both input and output. The
# service could also have used separate input and output views.
# The first 3 params are input params, the second 3 are outputs.
#
VIEW SimpView
$
#type  cname         fbname count  flag   size   null
string inString      - 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

Define VIEW in the Repository Editor

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

Note: If you are not familiar with using the Jolt Repository Editor, refer to "Using the Jolt Repository Editor" for more information about defining a service.

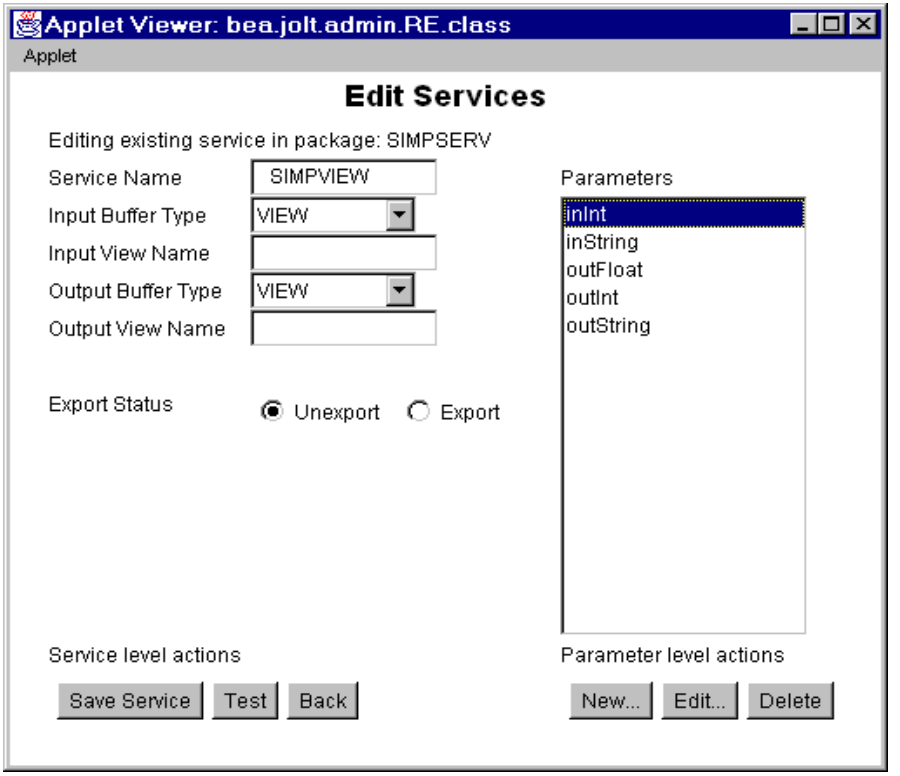
Figure 4-9 Add SIMPVIEW Service



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 4-10 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.c Server Code

In the following server code, the input and output buffers are VIEW. The code accepts the VIEW buffer data as input and outputs the same data as VIEW.

Listing 4-9 simpview.c Code Example

```
/*
 * SIMPVIEW.c
 *
 * Copyright (c) 1997 BEA Systems, Inc.  All rights reserved
 *
 * Contains the SIMPVIEW 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 {
 *
 *charinString[32];
 *longinInt;
 *floatinFloat;
 *charoutString[32];
 *longoutInt;
 *floatoutFloat;
 *};
 */
/*
 *  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);
}
```

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 Tuxedo

The following section 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 of the 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 will generally cause 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 version 1.0, 1.0.1, 1.0.2) machine running on a Sun platform. If you want your applet to work on JDK 1.0, 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 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 Tuxedo service function and then immediately start another thread that sends another request to a 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 program illustrates the use of two threads in a Jolt application.

Listing 4-10 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 Tuxedo Services or other Tuxedo clients. The Jolt Class Library contains classes that support the following types of Tuxedo notifications for handling event-based communication:

- **Unsolicited Event Notifications**. These are notifications that a Jolt client receives as a result of a 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 Tuxedo Event Broker. The notifications are only received when the Jolt client subscribes to an event *and* any Tuxedo client or server issues a system-posted event or `tppost()` call.

API for Event Subscription

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 “API Reference in Javadoc.”

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 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 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 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, since 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 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 Tuxedo data buffer type. Jolt clients (for example, the handler) will 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 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 in the `get*()` methods above.

Tuxedo Event Subscription

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

Jolt clients are only able to subscribe to events and notifications that are generated by other components in Tuxedo (such as a Tuxedo Service or Client). Jolt clients are not able to 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: This is *not* the recommended method for enabling unsolicited notification. If you want unsolicited notification, the application should explicitly do so (as described in the JoltUserEvent class). The reason for this is explained in the following section.

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 Tuxedo. In 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 only a single event, unsubscribing disables both the 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 remembers 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 Tuxedo Notifications

The example code provided in the following listing shows how to use the Jolt Class Library for receiving notifications and includes the use of the JoltSession, JoltReply, JoltMessage and JoltUserEvent classes.

Listing 4-11 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 includes a method (the `clear()` method) that allows you to remove existing attributes from an object and, in effect, provides for the reuse of the object. The `reuseSample.java` example illustrates how to use the `clear()` method for clearing parameter values.

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

Listing 4-12 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 `extendSample.java` example 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 which makes use of the Tuxedo bankapp funds TRANSFER service.

The example 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 only one of many different ways to extend from JoltRemoteService.

Listing 4-13 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 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);
            }

       }
}
```

Application Deployment and Localization

The Jolt Class Library allows you to build Java applications that execute from within a client Web browser. For these types of applications, you need to address the following application development tasks:

- Deploying your Jolt application in an HTML page
- Localizing your Jolt application for different languages and character sets

The following sections describe these application development considerations.

Deploying a Jolt Applet

When you deploy a Jolt applet, you need to consider the three components that operate together to make the applet function in a Web browser environment:

- Requirements for the 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 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 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.

5 Using JoltBeans

Formerly available as an add on, JoltBeans are included in BEA Jolt. Using JoltBeans, you can create Jolt client applications with the ease of using JavaBeans. JoltBeans are JavaBeans components that are used 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.

"Using Jolt Beans" covers the following topics:

- Overview of Jolt Beans
- JoltBeans Terms
- Adding JoltBeans to Your Java Development Environment
- JavaBeans Events and Tuxedo Events
- How JoltBeans Use JavaBeans Events
  - JoltSessionBean
  - JoltServiceBean
  - JoltUserEventBean
- Jolt Aware GUI Beans
- Using the Property List and the Property Editor to Modify the JoltBeans Properties
- JoltBeans Class Library Walkthrough
  - Building the Sample Form
  - Wiring the JoltBeans Together

- Using the Jolt Repository and Setting the Property Values

- JoltBeans Programming Tasks
  - Using Transactions with JoltBeans
  - Using Custom GUI Elements with the JoltService Bean
  - How JoltBeans Use JavaBeans Events

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're 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 has been certified by BEA for use with JoltBeans. However, JoltBeans are also compatible with other Java development environments such as Visual Age.

The first topics in this section provide a general, conceptual overview of how JoltBeans work, as well as a description of each Jolt bean and how it interacts with Tuxedo events. The JoltBeans walkthrough demonstrates the specific steps required to create a Jolt client that interacts with Tuxedo services.

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

Refer to the following terms as you work with JoltBeans:

JavaBeans
: 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 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 (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, you must set up your development environment to include JoltBeans. To set up your Java development environment, you must:

- Set the class path in your development environment to include all the 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're using.

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 have placed 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, as explained in the next section, "Using Development and Runtime JoltBeans."

Using Development and Runtime JoltBeans

The `.jar` files containing JoltBeans contain two versions of each JoltBean, a development version and a runtime version. The development version of each JoltBean name ends with the suffix `Dev`. The runtime version of each class name ends with the suffix `Rt`. For example, the development version of the class, JoltBean, is `JoltBeanDev`, while the runtime 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 development Beans have graphic properties ("bean information") that allow you to work with them as graphic icons in your development environment.

The runtime version of JoltBeans does not have these additional properties. You do not need the additional development properties of the beans at runtime. The runtime 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 runtime beans are used when compiling your application.

Basic Steps For Using JoltBeans

After you have added the development version of JoltBeans to the Component Library of your Java development environment, the basic steps in using JoltBeans during development are as follows:

1. Drag the beans from the JoltBeans component palette of your development environment to the Java form-designer for a Jolt client application or applet.
2. 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.
3. Finally, add the application logic to the event callbacks.

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

JavaBeans Events and Tuxedo Events

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

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

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

The following example shows how the Tuxedo asynchronous notification mechanism is used in JoltBeans applications.

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

When the client is no longer interested in the event, it calls `unsubscribe()` on the JoltUserEventBean.

Note: If the client wants only to 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

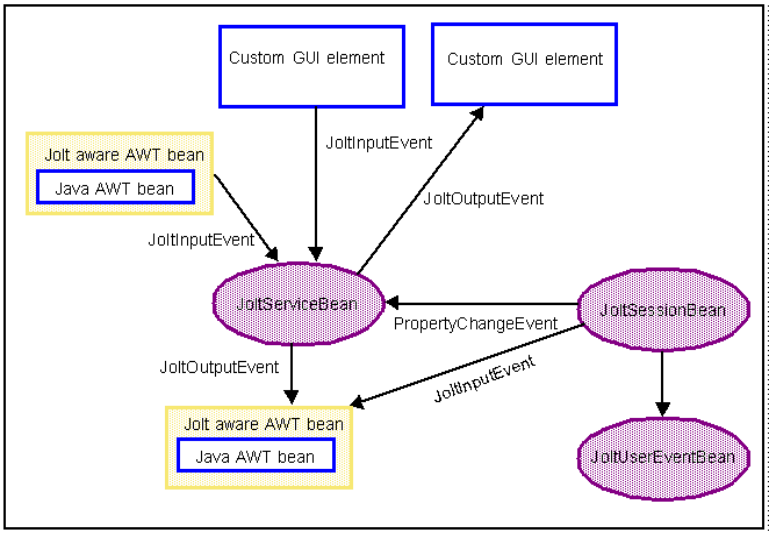
A Jolt client applet or application that has been built using JoltBeans typically consists of Jolt-aware GUI Beans, such as JoltTextField or JoltList, and JoltBeans, such as JoltServiceBean and JoltSessionBean. The main mode of communication between Beans is by JavaBeans events.

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 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 shows a graphical representation of the possible relationships among the JoltBeans.

Figure 5-1 Possible Interrelationships Among the 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 ease.

Refer to the online API Reference in Javadoc in this help system 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 Tuxedo session, encapsulates the functionality of the JoltSession, JoltSessionAttributes, and JoltTransaction classes. The JoltSessionBean offers properties to set session and security attributes, such as sending a timeout or a Tuxedo user name, as well as methods to open and close a Tuxedo session.

The JoltSessionBean sends a PropertyChange event when the 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 5-1 JoltSessionBean Properties and Descriptions

Property	Description
AppAddress	Set the IP address (host name) and port number of the JSL or the Jolt Relay. The format is `//host:port number` (e.g., `myhost:7000`).
AppPassword	Set the 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 Tuxedo session does or does not exist.
ReceiveTimeOut	Set the RECVTIMEOUT value.
SendTimeOut	Set the SENDTIMEOUT value.
SessionTimeOut	Set the SESSIONTIMEOUT value.

Table 5-1 JoltSessionBean Properties and Descriptions (Continued)

Property	Description
UserName	Indicate the Tuxedo user name, if required.
UserPassword	Indicate the Tuxedo user password, if required.
UserRole	Indicate the Tuxedo user role, if required.

JoltServiceBean

The JoltServiceBean represents a remote 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 5-2 JoltServiceBean Properties and Descriptions

Property	Description
ServiceName	The name of the Tuxedo service represented by this JoltServiceBean.
Session	The JoltSessionBean associated with the bean that allows access to the 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 Tuxedo events. The Tuxedo event to subscribe to or unsubscribe from is defined 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 5-3 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 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 may 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 this textfield is linked to 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 this checkbox is linked to 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 this choice is linked to 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 may 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 online Javadoc class reference library included with Jolt.

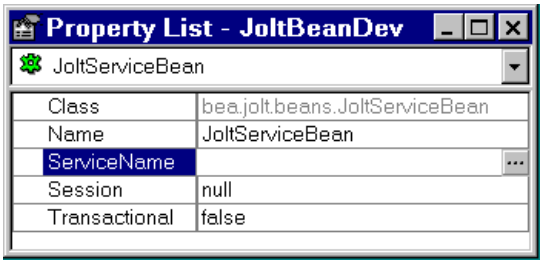
Using the Property List and the Property Editor to Modify the JoltBeans Properties

The values of most JoltBeans properties can be modified by simply editing the right column of the Property List, as shown in the following figure.

Custom Property Editors are provided, for some properties of JoltBeans.

The Custom Property Editors, accessed from the Property List, include dialog boxes that are used to modify the property values. You can invoke the Custom Property Editors from the Property List by selecting the button with the ellipsis ("...") that is next to the value of the corresponding property value.

Figure 5-2 Example of the Property List and Ellipsis Button



When the ellipsis button is selected, the Property Editor shown in the following figure displays.

Figure 5-3 Custom Property Editor Dialog Box

The Custom Property Editor of JoltBeans reads cached information. Initially, there is no cached information 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.

Details of the logon and an example of using the Property List and Property Editor are shown in the "Using the Jolt Repository and Setting the Property Values" section of the "JoltBeans Class Library Walkthrough."

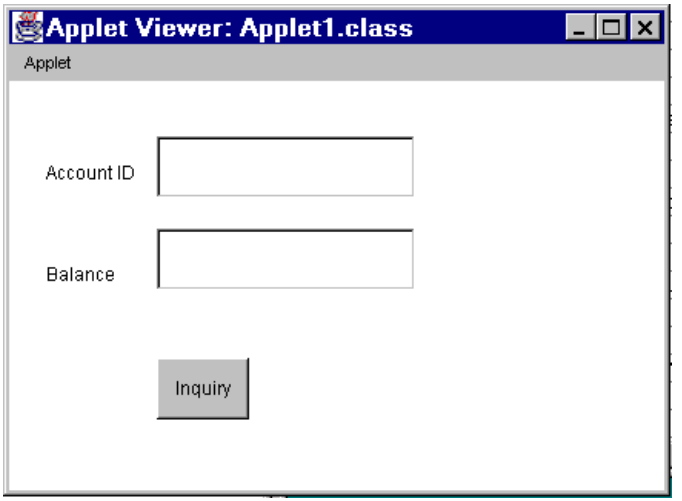
JoltBeans Class Library Walkthrough

This walkthrough describes how to build an applet that is used to:

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

This is 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 Tuxedo. To run this sample, the Tuxedo server must be running.

Figure 5-4 Sample Inquiry Applet



Refer to Figure 5-6 for an example of each item. To begin, select the following beans, shown in the following table.

Table 5-4 Required Form Components

Component	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 Tuxedo session.
JoltTextField	Gets input from the user (in this case, ACCOUNT_ID).
JoltTextField	Displays the result (in this case, SBALANCE).
JoltServiceBean	Accesses a Tuxedo service. (In this case, INQUIRY from BANKAPP).
Button	Initiates an action.
Label	Describes the field on the applet.

Building the Sample Form

The following example is created using the Visual Café 3.0 development environment. The example demonstrates how to build an applet that allows you to enter an account ID and use a Tuxedo service to get and show the account balance. The basic steps to create this example are as follows:

1. 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 will differ from the look of Visual Café 3.0. You can complete this sample applet in a previous version Visual Café, however, the steps executed in the Interaction Wizard differ slightly from this example.

Placing JoltBeans onto the Form Designer

1. First, 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 form designer.

   The following figure shows how a JoltBean is selected and dragged and dropped onto the palette of Form Designer.

   The next figure shows how JoltBeans appear when they are placed on the palette of the Form Designer.

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

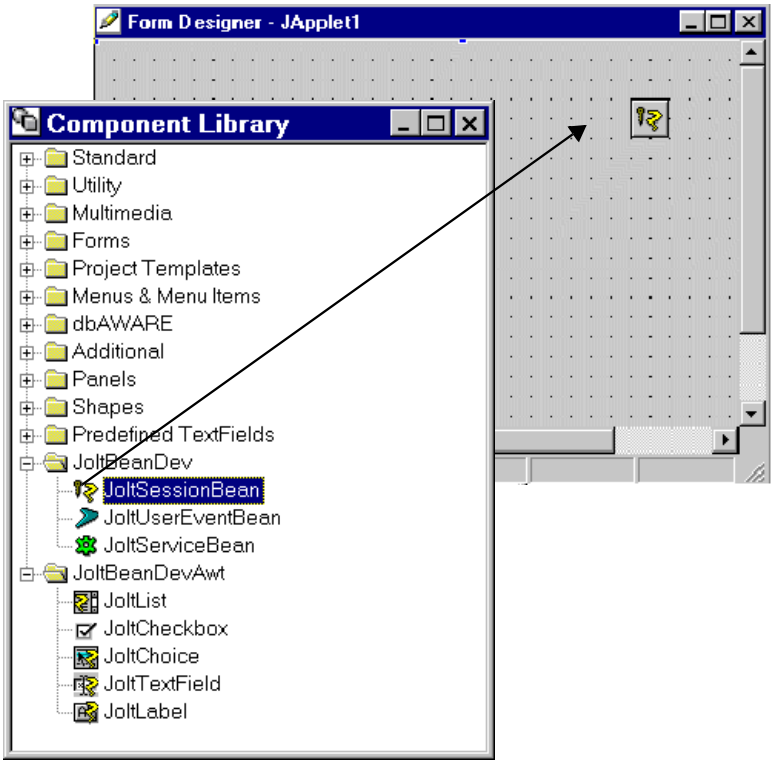
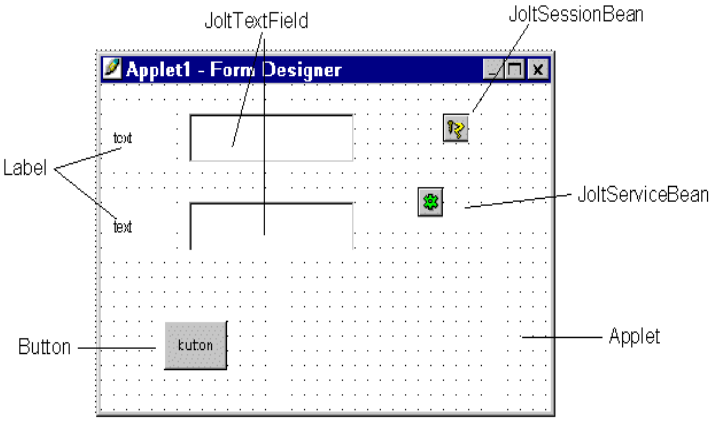


Figure 5-6 Visual Café 3.0 Form Designer



3. Next, 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 example in the next figure 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"section in this chapter.

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 5-5 Required and Recommended Property Values

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

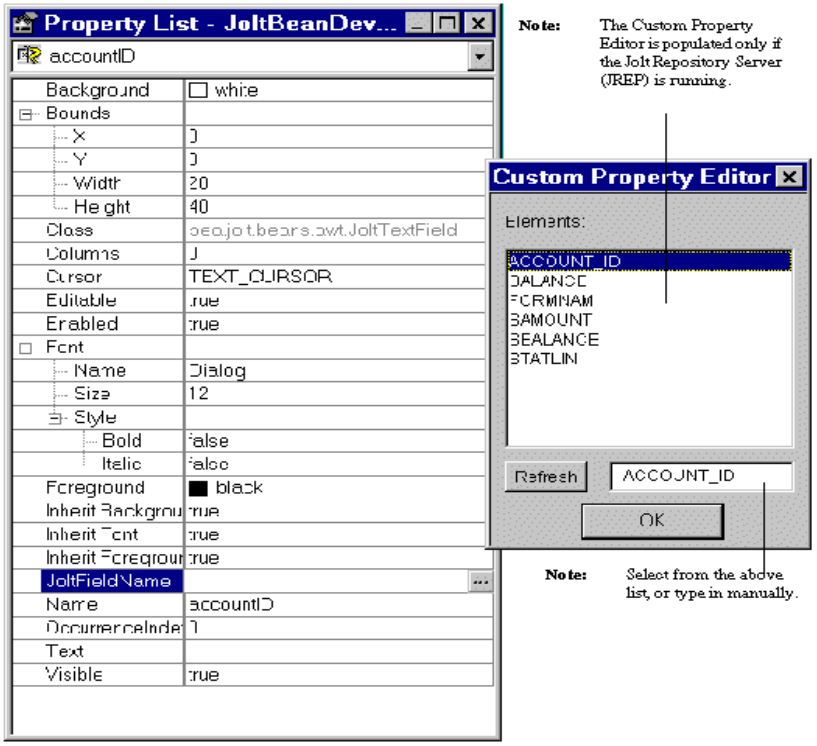
Table 5-5 Required and Recommended Property Values

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 next table and the section, “Using the Jolt Repository and Setting the Property Values” for general guidelines on JoltBean properties.

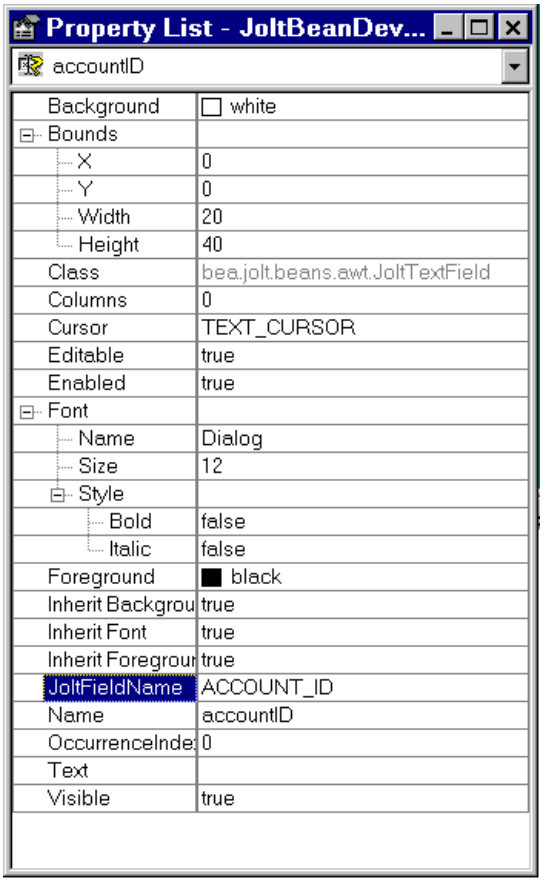
Figure 5-7 Example of JoltTextField Property List and Custom Property Editor



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 displays. Select or type the new field name (in this example, “ACCOUNT_ID”). Select **OK**.

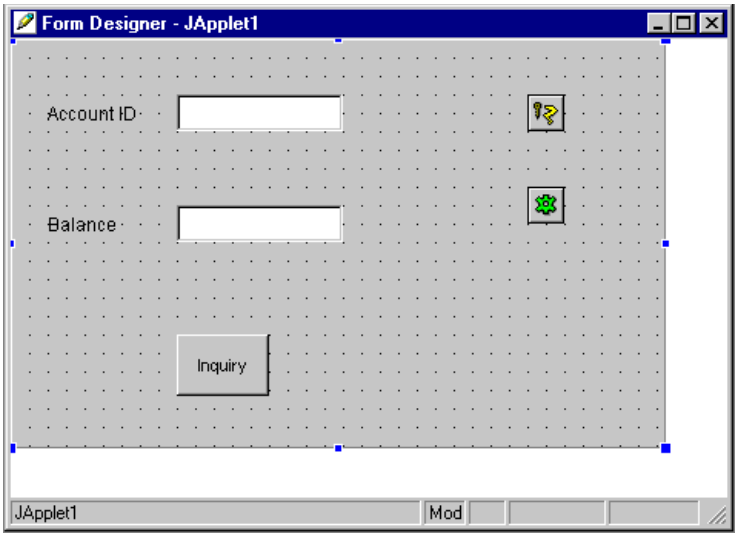
Note: The properties that are visible in the Custom Property Editor are cached locally, therefore, if the source database is modified you must use the Refresh button to see the current, available properties.

Figure 5-8 Revised JoltFieldName in the JoltTextField Property List



The change is reflected in the Property List shown in the previous figure and on the text field shown in the next figure.

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



The previous figure shows how the text on the button and the textfields changes after the text is added to the property list fields for these beans.

6. After you set the properties to the right values (refer to the "Required and Recommended Properties" table for additional information on the required and recommended property values), define how the beans will interact by wiring them together using the Visual Café Interaction Wizard.

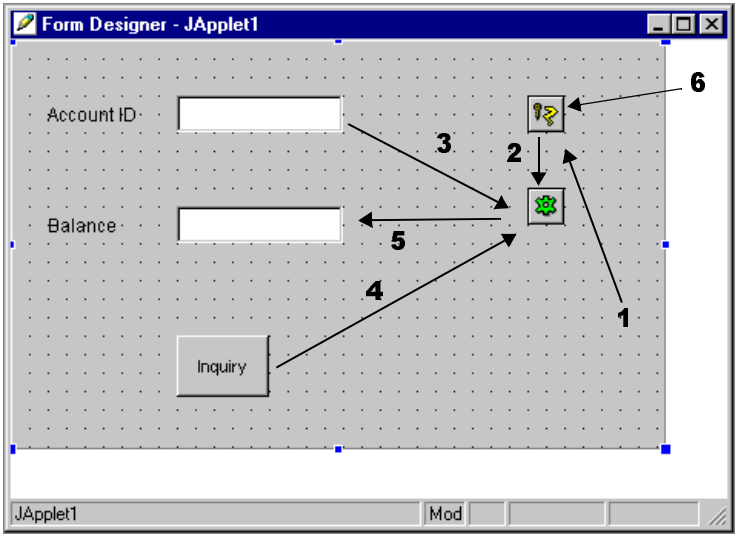
Wiring the JoltBeans Together

After all the beans are positioned on your form and the properties are set, wire the beans and their events together. Figure 5-10 gives an example of the flow to help you determine the order when you are ready 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 numbers of the steps that follow.

Figure 5-10 How JoltBeans are Connected on the Form

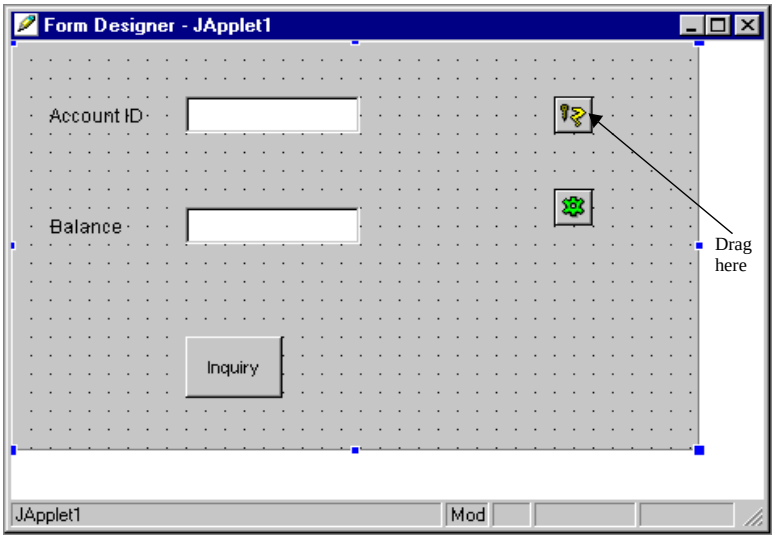


Step 1: Wire the JoltSessionBean logon

1. Click the Interaction Wizard button. Click in the applet window and drag a line to the JoltSessionBean.

   The Visual Cafe Interaction Wizard window displays.

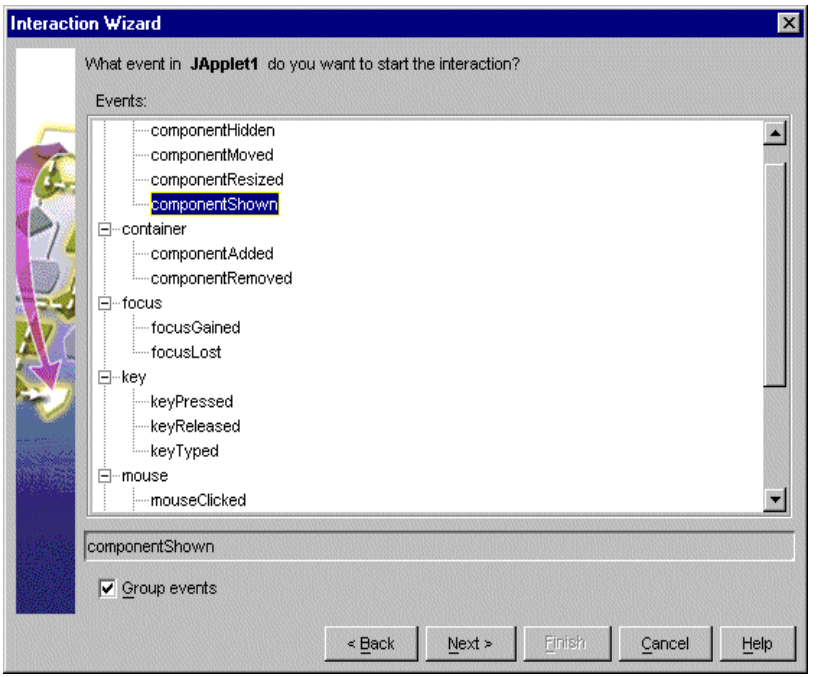
Figure 5-11 Wire the Applet to the Jolt Session Bean



2. Select ComponentShown (as shown below) as the event you want to start the interaction. Click **Next**.

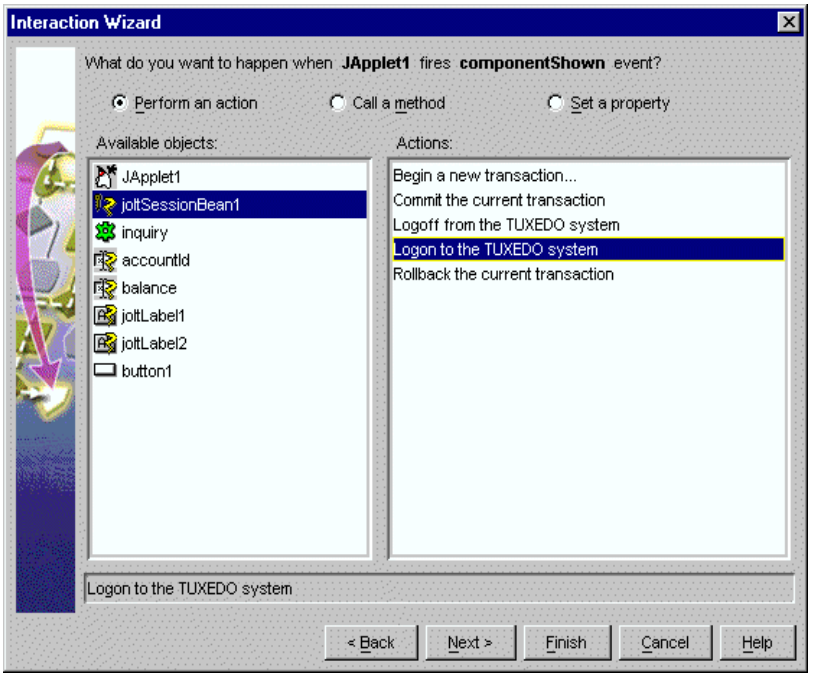
   The Visual Café Interaction Wizard displays.

Figure 5-12 Visual Cafe Interaction Wizard



3. With the “Perform an action” radio button enabled, select “Logon to the Tuxedo system.” Click **Finish**.

Figure 5-13 Select “Logon to the Tuxedo System

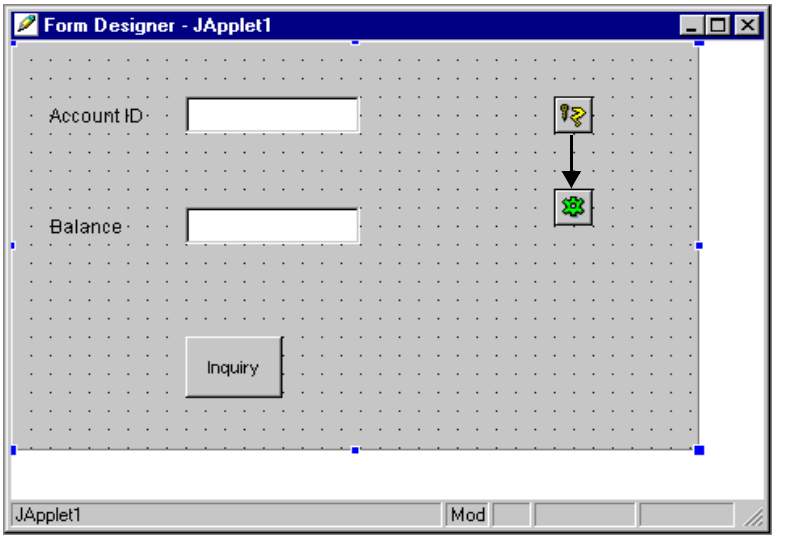


Completing these three steps enables the `logon()` method of the JoltSessionBean to be triggered by an applet (for example, ComponentShown) that is sent when the applet is opened for the first time.

Step 2: Wire JoltSessionBean to JoltServiceBean using propertyChange

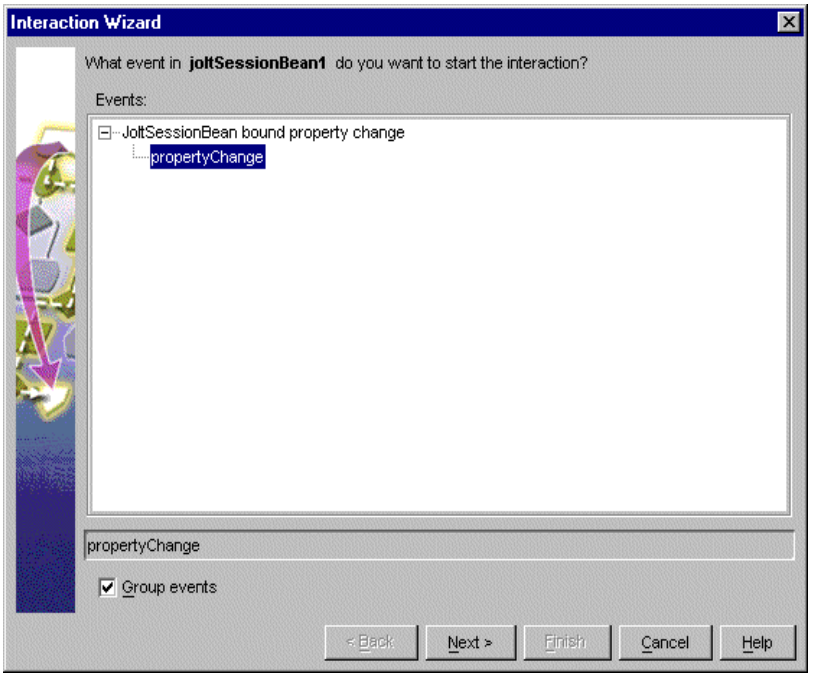
1. Click the Interaction Wizard button. Click on the JoltSessionBean and drag a line to the JoltServiceBean (as shown in the following figure).

Figure 5-14 Wire the JoltSessionBean to the JoltServiceBean



The Interaction Wizard displays and asks, “What event in joltSessionBean1 do you want to start the interaction?” (as shown in the following figure).

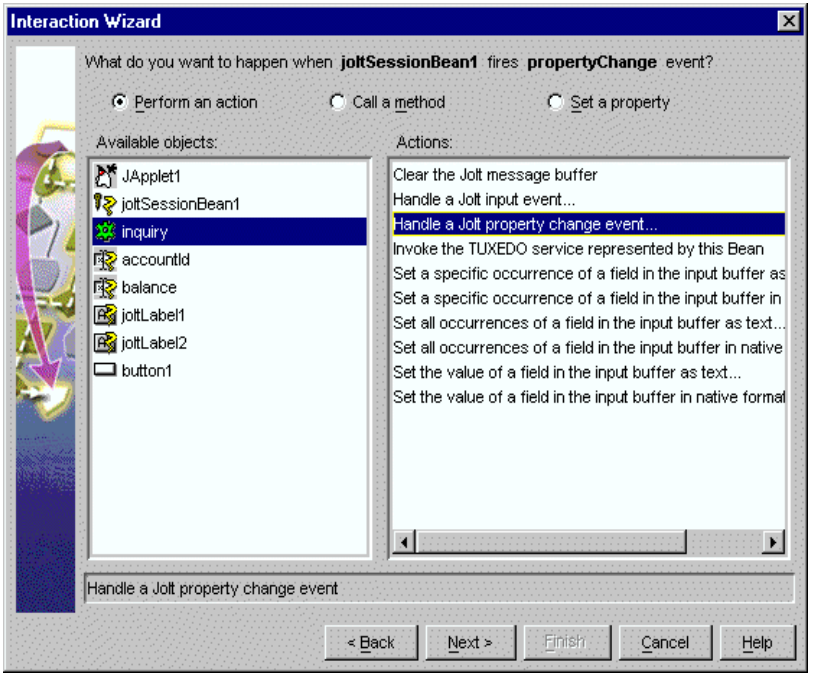
Figure 5-15 Select propertyChange as the Event



2. Select propertyChange as the event that starts the interaction. Click **Next.**

   The Interaction Wizard window displays and asks, “What do you want to happen when joltSessionBean1 fires propertyChange event?” and provides a list of object and actions (as shown in the following figure).

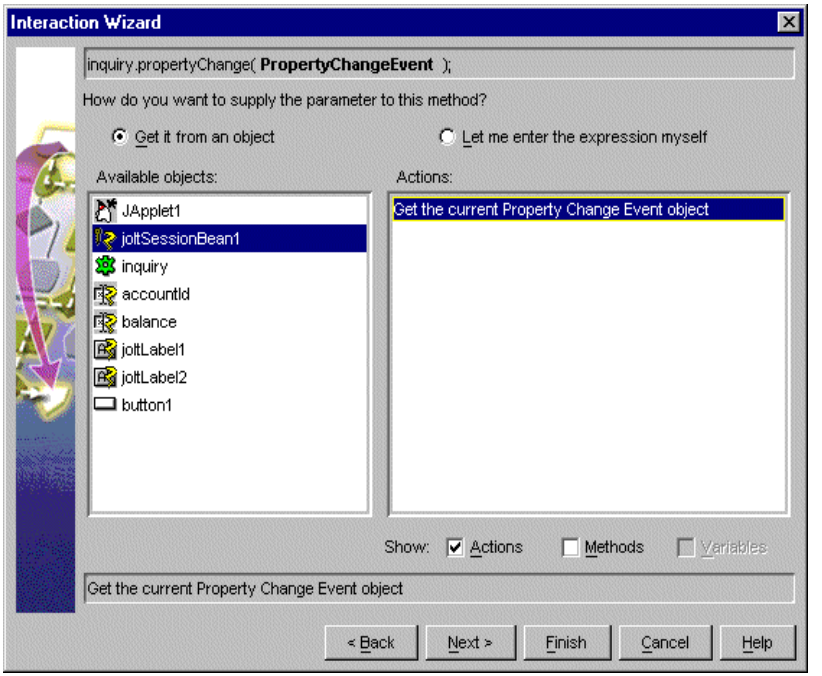
Figure 5-16 Select “Handle a Jolt property change event”



3. Select “Handle a Jolt property change event” as the method. Click **Next**.

   The Interaction Wizard window displays and asks, “How do you want to supply the parameter to this method?” and provides a list of available objects and actions to choose from (as shown in the following figure).

Figure 5-17 Select joltSesssionBean1



4. Select joltSessionBean1 as the object that supplies the action.

5. Select “Get the current Property Change Event object” as the action. Click **Finish**.

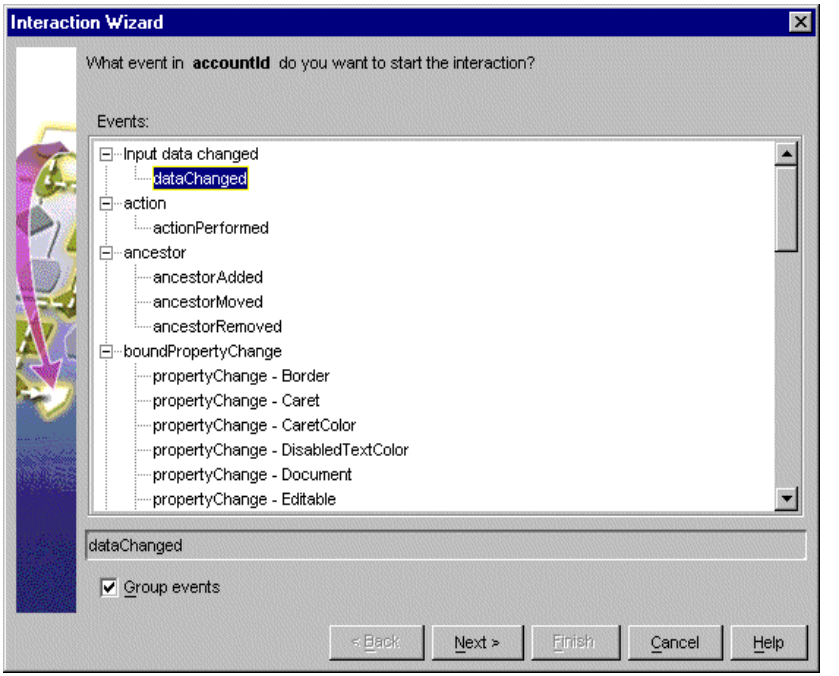
   Completing these three steps enables the JoltSessionBean to send a propertyChange event when `logon()` completes. The JoltServiceBean listens to this event and associates its service with this session.

Step 3: Wire the accountID JoltTextField as input to the JoltServiceBean using JoltInputEvent

1. Click the Interaction Wizard button. Select the accountID JoltTextField bean and drag a line to the JoltServiceBean.

   The Interaction Wizard displays.

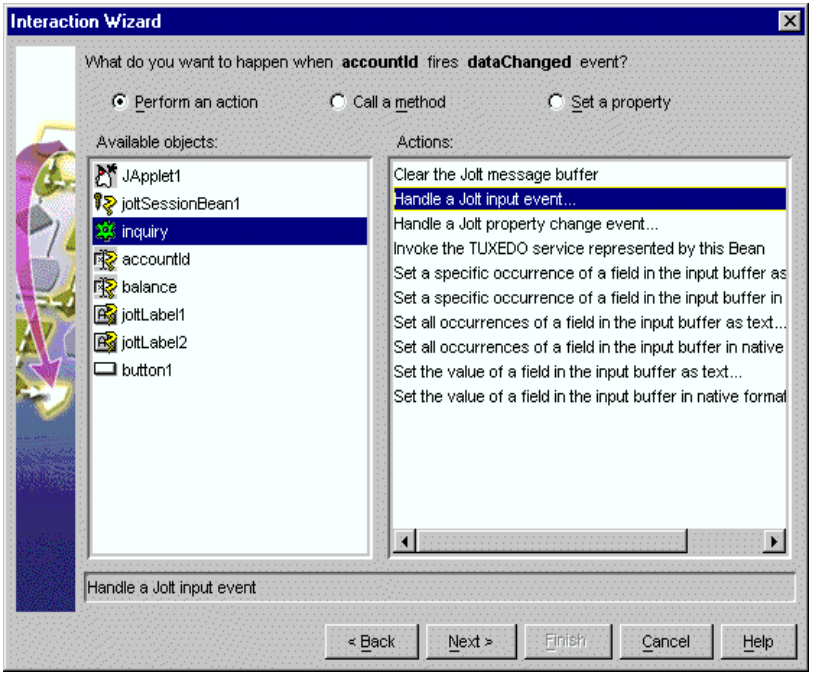
Figure 5-18 Select dataChanged as the Event to Start the Interaction



2. Select dataChanged as the event. Click **Next**.

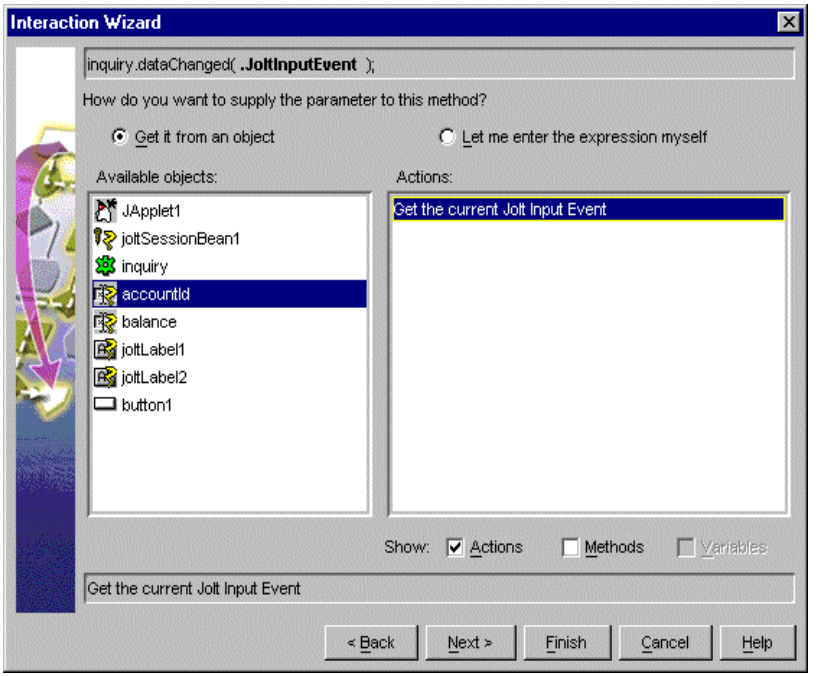
3. With the joltServicebean “inquiry” selected as the object supplying the parameter, select “Handle a jolt input event” as the action. Click **Next**.

Figure 5-19 Choose “inquiry” and “Handle a Jolt Input Event”



The Interaction Wizard window displays, asking “How do you want to supply the parameter to this method?” and providing a list of available objects and actions to choose from (as shown in the following figure).

Figure 5-20 Select "accountId" and Get the current Jolt Input Event



4. With "accountId" selected as the object, select "get the current Jolt Input Event" as the action. Click **Finish**.

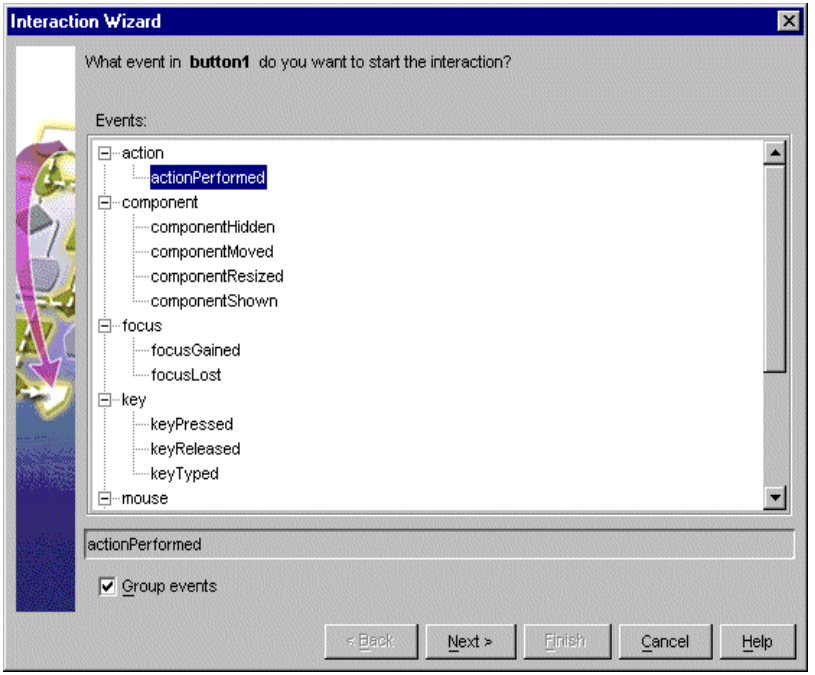
   Completing these four steps enables you to type the account number in the first text field. The JoltFieldName property of this JoltTextField is set to "ACCOUNT_ID". Whenever the text inside this text box changes, it sends a JoltInputEvent to the JoltServiceBean. (The JoltServiceBean listens to JoltInputEvents from this textbox.) 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 button. Select the Inquiry Button and drag a line to the JoltServiceBean.

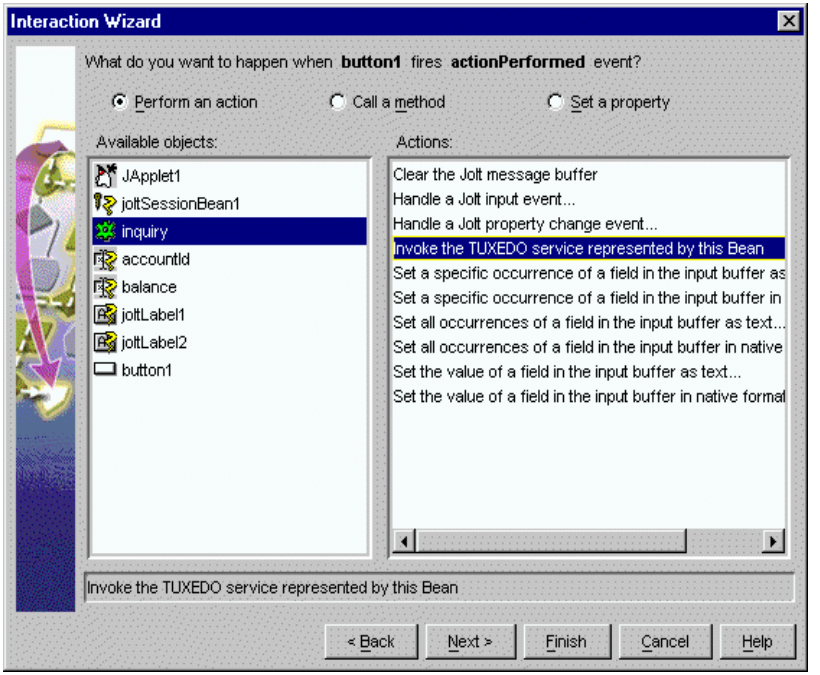
   The Interaction Wizard window displays.

Figure 5-21 Select "action Performed" as the event



2. Select "action Performed" as the event. Click **Next**.

Figure 5-22 Select “inquiry” and “Invoke Tuxedo Service...”



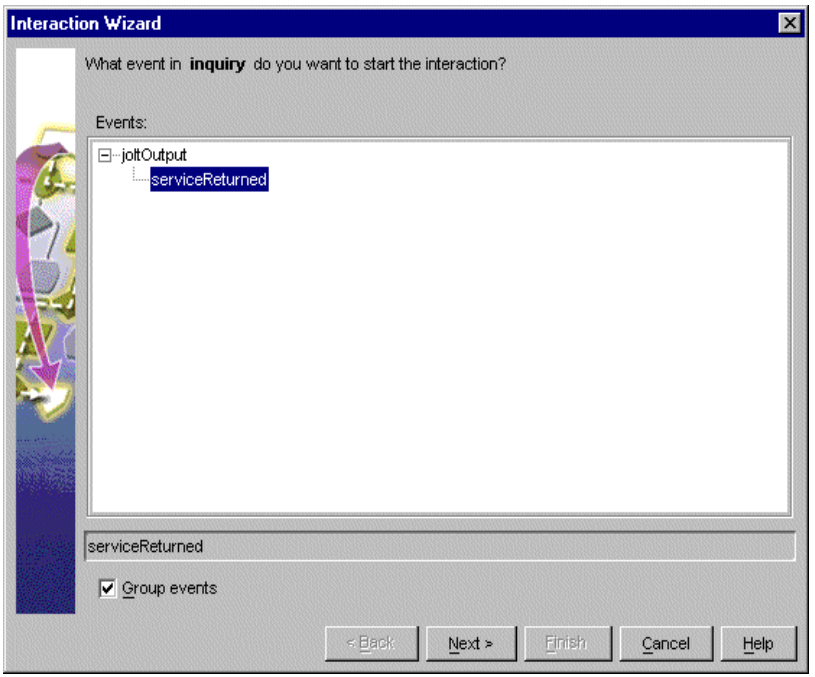
3. Select “Invoke Tuxedo Service represented by this bean” as the action. Click **Finish**.

   Completing these two steps 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

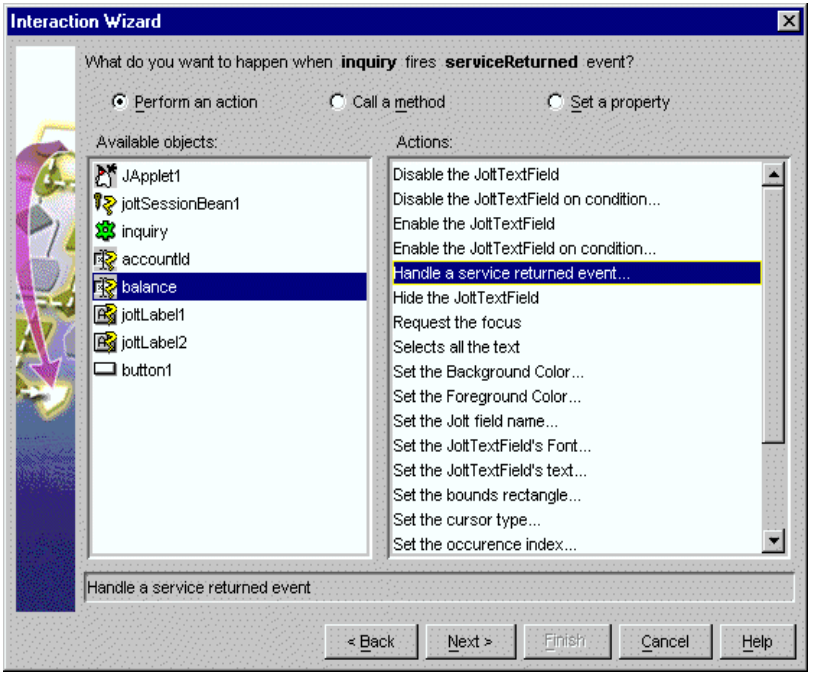
1. Click the Interaction Wizard button. Select the JoltServiceBean and drag a line to the AmountJoltTextField bean. The Interaction Wizard displays (as shown in the following figure).

Figure 5-23 Select "serviceReturned" as the event



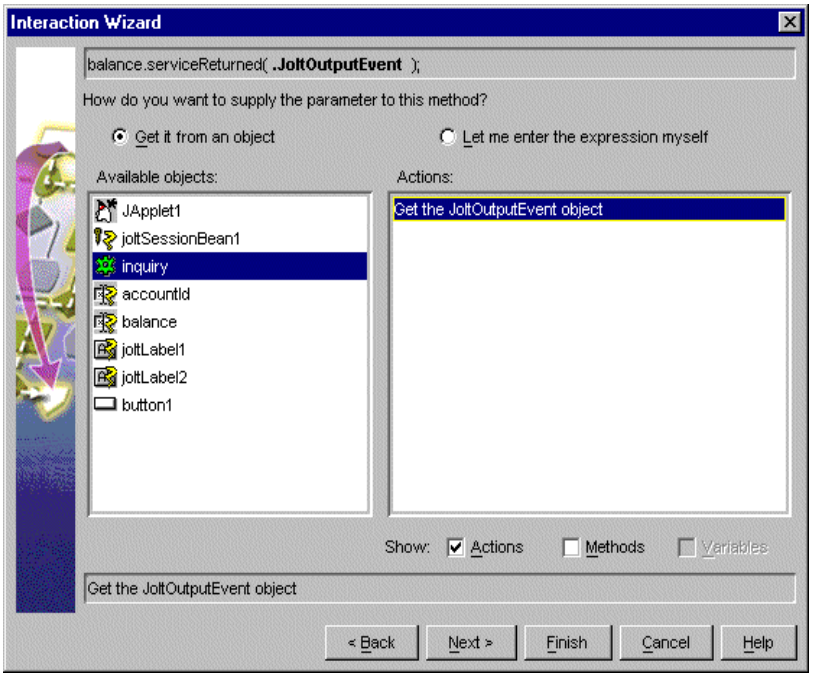
2. Select serviceReturned as the event. Click **Next**.

Figure 5-24 Select “balance” and “Set the JoltTextField’s text”



3. Select “balance” as the available object, and “Handle a service returned event...” as the action. Click **Next**.

Figure 5-25 Select “inquiry” and “Get the value of field...”



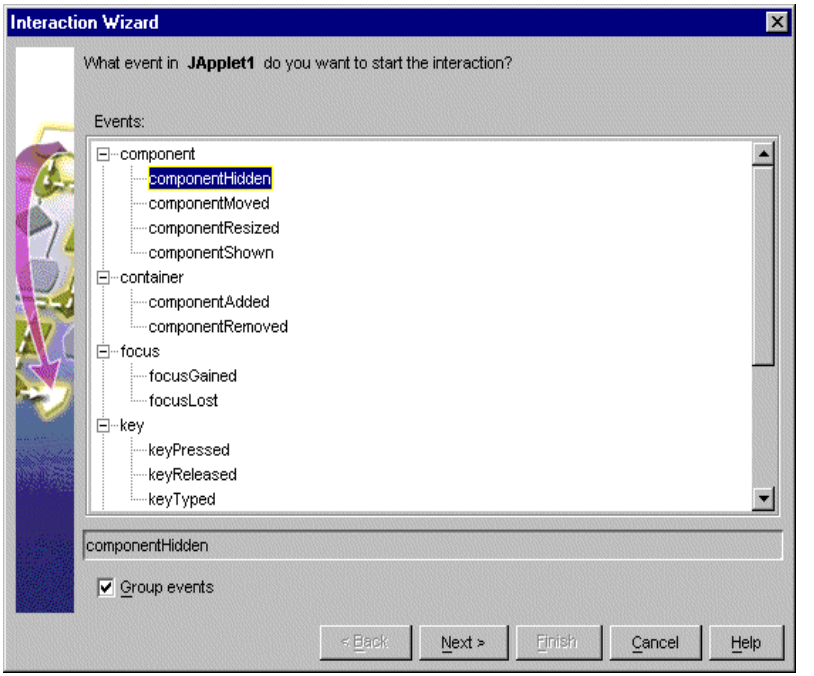
4. Select “inquiry” as the object, and “Get the JoltOutputEvent object” as the action. Click **Finish**.

   Completing these three steps 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

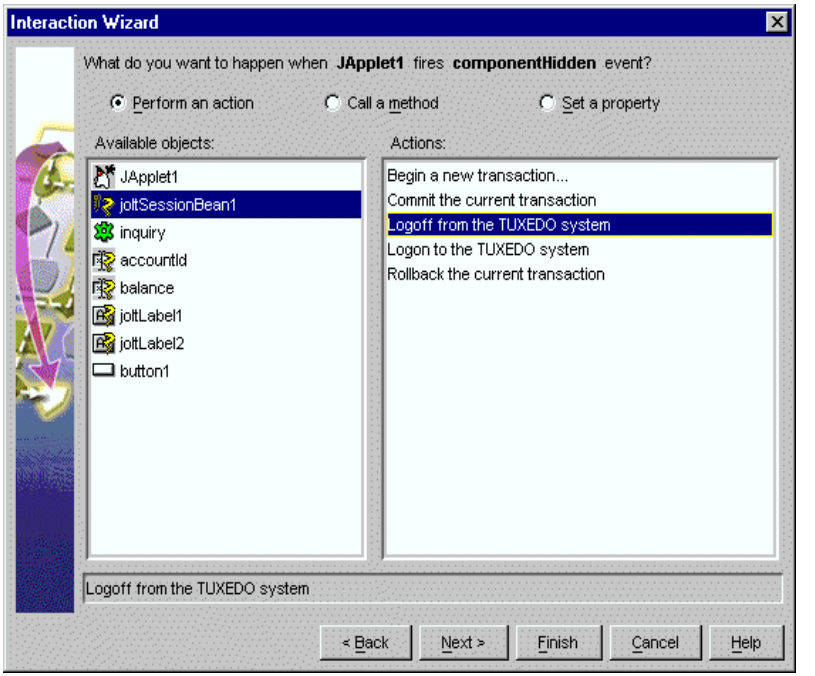
1. Click the Interaction Wizard button. Click in the applet window (not on another bean) and drag a line to the JoltSessionBean. The Interaction Wizard displays (as shown in the following figure).

Figure 5-26 Select "Interaction Wizard"



2. Select componentHidden as the event. Click **Next**.

Figure 5-27 Select "joltSessionBean1" and "Logoff from the Tuxedo system"



3. With "joltSessionBean1" selected as the object, select "Logoff from the Tuxedo system" as the action (as shown in the previous figure). Click **Finish**.

   Completing these two steps 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 recommeded that you fill in the empty catch blocks for exceptions. Check the message window for any compilation errors and exceptions.

Refer to the following table and the following figure for additional information.

Running the Sample Application

To run the sample application, you must have the Tuxedo server running. Then enter an account number in the Account ID textfield. You may 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 5-6 JoltBean Specific Properties

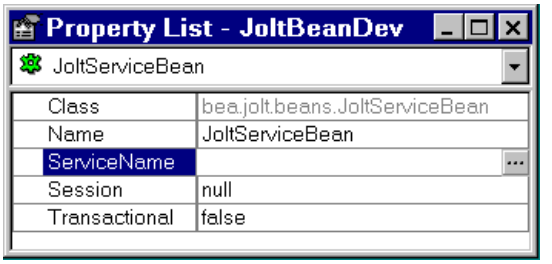
JoltBean	Property	Input Description
JoltSessionBean	appAddress	e.g., `//host:port`
	userName, Password or AppPassword	Type your Tuxedo user name and passwords.

Table 5-6 JoltBean Specific Properties

JoltBean	Property	Input Description
JoltServiceBean	serviceName isTransactional	INQUIRY, for example. 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 filter	Refer to the Tuxedo `tpsubscribe` calls.
All Jolt-aware GUI beans	joltFieldName occurrenceIndex	ACCOUNT_ID, for example Multiple fields of the same name. Index starts at 0.
JoltCheckbox	TrueValue and FalseValue	The field value corresponding to the state of the checkbox.

The property editor reads cached information from the repository and returns names of the available services and data elements in a list box. An example of the ServiceName property editor is shown in the following figure:

Figure 5-28 JoltServiceBean Property Editor



1. Select the service name by clicking on the ellipsis in the ServiceName field shown in the previous figure.

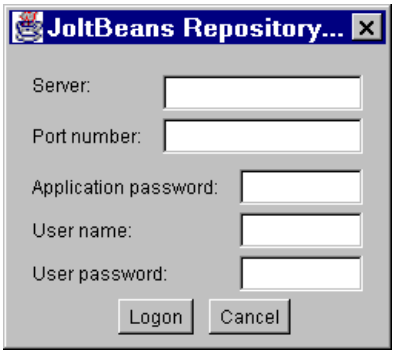
   The Custom Property Editor for ServiceName shown in the following figure displays.

Figure 5-29 Custom Property Editor for serviceName

Note: If you cannot or do not want to connect to the Repository database, simply type the service name in the text box and proceed 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 displays.

Figure 5-30 JoltBeans Repository Log On



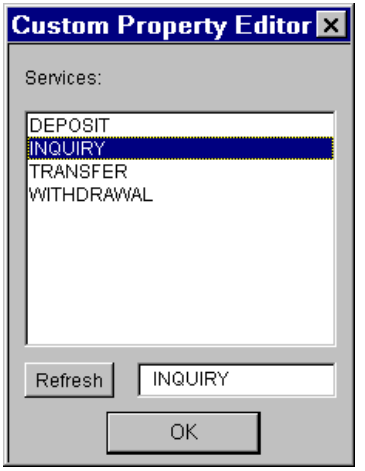
3. Type the Tuxedo or Jolt Relay Machine name for Server and the JSL or Jolt Relay Port Number. Type password and user name information (if required) and click **Logon**.

4. The Custom Property Editor loads its cache from the repository.

5. Select the appropriate service name in the list box shown in Figure 5-31. Enter the property value (service or field name) directly. A text box is provided. Click **OK** on the property editor dialog (shown in the following figure). The bean property is set with the contents of the textbox.

6. Click **OK** on the Custom Property Editor dialog (shown in the following figure).

Figure 5-31 Property Editor with Selected Service



JoltBeans Programming Tasks

Additional programming tasks include:

- Using Transactions with JoltBeans
- Using Custom GUI Elements with the JoltService Bean

Using Transactions with JoltBeans

Your 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 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.

6 Using Servlet Connectivity for 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 means that HTML clients can invoke Tuxedo services without directly connecting to Tuxedo. HTML clients can instead connect to a Web server, through HTTP, where the Tuxedo service request is executed by a generic servlet. Using a Jolt session, the servlet on the web server administers the Tuxedo service request by connecting to the Tuxedo Server through the Jolt Server Handler (JSH) or the Jolt Server Listener (JSL), which then makes the Tuxedo service request. This capability allows many types of HTML clients to make remote 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 section covers the following topics:

- What is a Servlet?
- How Servlets Work With Jolt
- Writing and Registering HTTP Servlets

- Jolt Servlet Connectivity Sample
- Additional Information on Servlets

What is a Servlet?

A servlet is any Java class that can be invoked and executed on a server, usually on behalf of a client. A servlet works on the server, while an applet works on the client. An HTTP servlet is a Java class that handles an HTTP request and delivers an HTTP response. HTTP servlets reside on an HTTP server and must extend the JavaSoft javax.servlet.http.Http Servlet Class so that they may 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 that 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

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 in 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 Tuxedo. The following servlet tasks are illustrated by the Simpapp sample:

- How to use a property file to create a session pool.
- How to get the session pool manager.
- How to retrieve the session pool by name.

- How to invoke a Tuxedo service.
- How to process the result set.

This example demonstrates how a servlet may connect to 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 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.
- Tuxedo 7.1 or above with SimpApp sample running.
- Jolt

Installing the SimpApp Sample

1. Install the Jolt class library (`jolt.jar`) and Servlet Connectivity for Tuxedo class library (`joltjse.jar`) to 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 classes library and Servlet Connectivity for 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 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 WebLogic, add the following line to the `weblogic.properties` file:

```
weblogic.httpd.register.simpapp=examples.jolt.servlet.SimpAppSe
rvlet
```

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 Tuxedo. Bankapp illustrates the following:

- How to use a property file to create a session pool.
- How to get the session pool manager.
- How to retrieve a session pool by name.
- How to invoke a Tuxedo service.
- How to 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.
- Tuxedo 7.1 with BankApp sample running.
- Jolt

Installation Instructions

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

- How to use the administrative API to control the session pools.
- How to retrieve the statistics through PageCompiledServlet in Servlet Connectivity for 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
- Jolt

Installation Instructions

1. Install the Jolt class library and Servlet Connectivity for Tuxedo class library to 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, see the following sites:

BEA WebLogic Servlet Documentation

`http://www.weblogic.com/docs/classdocs/API_servlet.html`

Java Servlets

`http://jserv.java.sun.com/products/java-server/documentation/webserver1.1/index_developer.html`

Servlet Interest Group

`servlet-interest@java.sun.com`

7 Using Jolt ASP Connectivity for Tuxedo

The Jolt ASP Connectivity for Tuxedo provides an easy-to-use interface for processing and generating dynamic HTML pages. You do not need to learn how to write Common Gateway Interface (CGI) transactional programs to access Tuxedo services.

The following topics are discussed in this section:

- Key Features
- ASP Connectivity Enhancements for Jolt
- How the Jolt ASP Connectivity for Tuxedo Works
- The ASP Connectivity for Tuxedo Toolkit
- Jolt ASP Connectivity for Tuxedo Walkthrough
- Overview of the ASP for Tuxedo Walkthrough
- Getting Started Checklist
- Overview of the TRANSFER Service
- TRANSFER Request Walkthrough
- Initializing the Jolt Session Pool Manager
- Submitting a TRANSFER Request from the Client

- Processing the Request
- Returning the Results to the Client

Key Features

The Jolt ASP Connectivity for Tuxedo, an extension to the Jolt class library, enables Tuxedo services and transactions to be invoked from a Web server using a scripting language.

Some of the benefits of this architecture include:

- The HTML interface is preserved.
- The need to download Java class files is eliminated along with the delays associated with the download.
- Session Pooling efficiently utilizes the Tuxedo resources.
- Leverages industry standard HTTP protocol with encryption, and firewall configuration for the Web server.

Note: Asynchronous notification is not available in the ASP Connectivity for Tuxedo. It is recommended that Jolt enabled Java clients (applets) be written using a retained connection to support asynchronous notification.

ASP Connectivity Enhancements for Jolt

Jolt includes the following enhancements to ASP Connectivity for Tuxedo:

- The package name for JoltWAS has been changed from `bea.web` to `bea.jolt.pool`.
- The package name for Tuxedo-ASP Connectivity has been changed from **JoltWAS for IIS** to `bea.jolt.pool.asp`.

- All Java class names for Tuxedo-ASP Connectivity have been renamed with the prefix of **Asp** and have new ActiveX component names (for example, `BEAJOLTPOOL.AspSessionPoolManager`). It is recommended that existing JoltWAS for IS customers use the new ActiveX component names.
- A new `AspSessionPool.callEx()` method is added. It allows users to call a service with a container class `AspDataSet` object for arbitrary data types instead of the string array in the `AspSessionPool.call()` method.
- New `AspPoolManagerConfig` and `ServletPoolManagerConfig` classes are added to simplify the creation of the session pool manager and the session pools. The session pool uses the `java.util.Properties` class to pass in the following session pool properties:
  - poolname
  - appaddrlist
  - failoverlist
  - minipoolsize
  - maxpoolsize
  - username
  - userpassword
  - userrole
  - apppassword

How the Jolt ASP Connectivity for Tuxedo Works

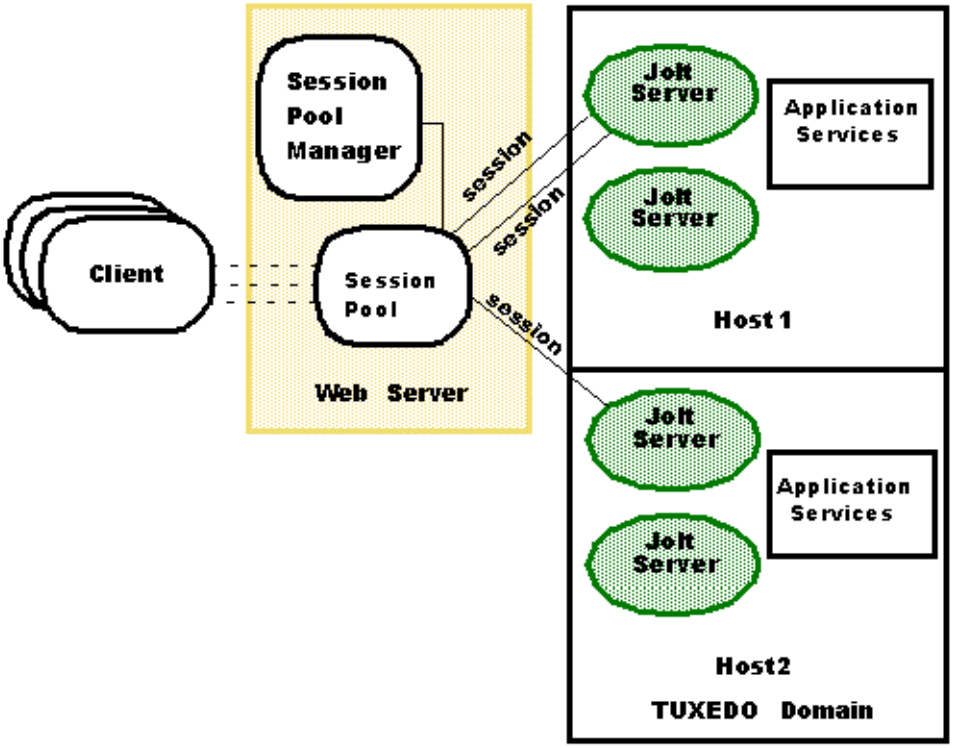
The Jolt ASP Connectivity for Tuxedo architecture includes three main components: a session, a session pool, and a session pool manager. A *session* object represents a connection with the Tuxedo system. A *session pool* represents many physical connections between the Web server and the Tuxedo system. It also associates a session with an HTTP request.

The *session pool manager* is responsible for maintaining a set of session objects, each having a unique session identifier.

1. If the Web application has not been initialized, the Web Application initializes the session pool manager, creates a session pool, and establishes sessions (also known as connections) with the Jolt Server.
2. When a service request arrives, the Web application gets a session pool object from the session pool manager. The session pool invokes the service call using the session that is the "least busy," based on the number of outstanding call requests on a given session.
3. If the selected session is terminated by the Jolt server, the session pool object restarts a new session or reroutes the request to another session. If the session pool manager is unable to get any session, a null session object is returned.

A graphical representation of the ASP Connectivity for Tuxedo architecture is shown in the following figure.

Figure 7-1 Jolt ASP Connectivity for Tuxedo Architecture



Refer to the online "API Reference in Javadoc" for additional information about the `SessionPool` class and `SessionPoolManager` class.

The ASP Connectivity for Tuxedo Toolkit

The ASP Connectivity for Tuxedo Toolkit is an extension to the Jolt Class Library. The Toolkit allows the Jolt Client Class Library to be used in a Web Server (such as Microsoft Active Server) to provide an interface between HTML clients or browsers, and a Tuxedo application.

Samples delivered with the software support four services: INQUIRY, WITHDRAWAL, DEPOSIT, and TRANSFER. This section explains the steps you follow to use an HTML client interface with the TRANSFER service of the Tuxedo bankapp application. The TRANSFER service illustrates the use of parameters with multiple occurrences. This walkthrough explains the use of the TRANSFER service only.

Jolt ASP Connectivity for Tuxedo Walkthrough

A complete listing of all the examples used in this chapter are distributed with the Jolt software. In this section, segments of code from these samples are used to illustrate the use of the Toolkit. The samples delivered with the software support four services: INQUIRY, WITHDRAWAL, DEPOSIT, and TRANSFER. This chapter explains the steps you can follow to use an HTML client interface to the TRANSFER service of the Tuxedo bankapp application. The TRANSFER service illustrates the use of parameters with multiple occurrences. This walkthrough explains the use of the TRANSFER service only.

Note: The walkthrough illustrates the use of the ASP Connectivity For Tuxedo with Microsoft IIS and VBScript.

To use the information in the following sections, you should be familiar with:

- BEA Tuxedo and the sample Tuxedo application, bankapp
- BEA Jolt

- HTML (Hypertext Markup Language)
- VB Script
- Object-oriented programming concepts

Overview of the ASP for Tuxedo Walkthrough

Follow these steps to complete the ASP Connectivity for Tuxedo walkthrough:

- Review the Getting Started Checklist
- Review the Overview of the TRANSFER Service
- Complete the Steps in the TRANSFER Request walkthrough
  - Initializing the Jolt Session Pool Manager
  - Submitting a TRANSFER Request from the Client
  - Processing the Request
  - Returning the Results to the Client

Getting Started Checklist

Review this checklist before starting the TRANSFER Request Walkthrough.

Note: This checklist applies to Microsoft Active Server Pages only.

1. Ensure that you have a supported browser installed on your client machine. The client machine must have a network connection to the Web server that is used to connect to the Tuxedo environment.
2. Configure and boot Tuxedo and the Tuxedo bankapp example.

a. Make sure the TRANSFER service is available.

b. Refer to the BEA Tuxedo user documentation for information about completing this task.

3. Refer to *Installing the BEA Tuxedo System* for information about how to configure a Jolt Server.

   a. Note the *hostname* and *port number* associated with your Jolt Server Listener (JSL).

   b. Ensure that the TRANSFER service is defined in the Jolt Repository.

   c. Test the TRANSFER service using the Jolt Repository Editor to make sure it is accessible to Jolt clients.

4. Make sure you have Microsoft IIS 4.0 up and running.

   a. Check that script execution permission is enabled in the Web server application properties.

   b. Refer to the user documentation that accompanies the Microsoft IIS server for instructions.

5. Install the Jolt Asp Connectivity For Tuxedo classes. These classes are contained in the `joltasp.jar` file. Be sure these classes are in your class path and available to your Web Server.

6. Install the teller sample application.

7. The code samples shown in "TRANSFER Request Walkthrough" are available from a sample application delivered with the Jolt Asp Connectivity For Tuxedo software. The following table lists the files in the sample application. These files are a valuable reference for the walkthrough and are located in `<extract_directory>/teller`.

Table 7-1 Bankapp Sample Source Files

File Name	Description
`tellerForm.asp`	Initializes the Jolt Session Pool Manager and displays available bankapp services.
`transferForm.htm`	Presents an HTML form for user input.

Table 7-1 Bankapp Sample Source Files

File Name	Description
`tlr.asp`	Processes the HTML form and returns results as an HTML page.
`web_admin.inc`	VBScript functions for initializing the Jolt Session Pool Manager.
`web_start.inc`	VBScript functions for initializing the Jolt Session Pool Manager.
`web_templates.inc`	VBScript functions for caching HTML templates.
`templates/transfer.temp`	HTML templates used for returning results.

Overview of the TRANSFER Service

The TRANSFER Service in bankapp moves funds between two accounts. The service takes two account numbers, an input amount, and returns two balances—one for each account. In addition, the service returns an error message if there is an application or system error.

A TRANSFER is a WITHDRAWAL and a DEPOSIT executed as a single transaction. The transaction is created on the server, so the client does not need to create a transaction.

The client interface consists of an HTML page with a form used to enter the required data — account numbers and a dollar amount. This data is sent to the Web server as a `"POST"` request.

In the Web server, this request is processed using a VBScript Active Server Page. This program extracts the input data fields from the request, formats them for use with the Jolt ASP Connectivity For Tuxedo class library, and dispatches the request to the TRANSFER service in the bankapp application. The TRANSFER service returns the results of the transaction. These results are returned to the VBScript program that merges them into a dynamically created HTML page. This page is returned to the client via the Web server infrastructure.

In the final part of this walkthrough, run the necessary HTML pages and server-side VBScript logic to execute a TRANSFER.

TRANSFER Request Walkthrough

This section explains what happens when you execute a TRANSFER request. Every step is not illustrated here, only those steps that are necessary.

Included are:

- Initializing the Jolt Session Pool Manager
- Submitting a TRANSFER Request from the Client
- Processing the Request
- Returning the Results to the Client

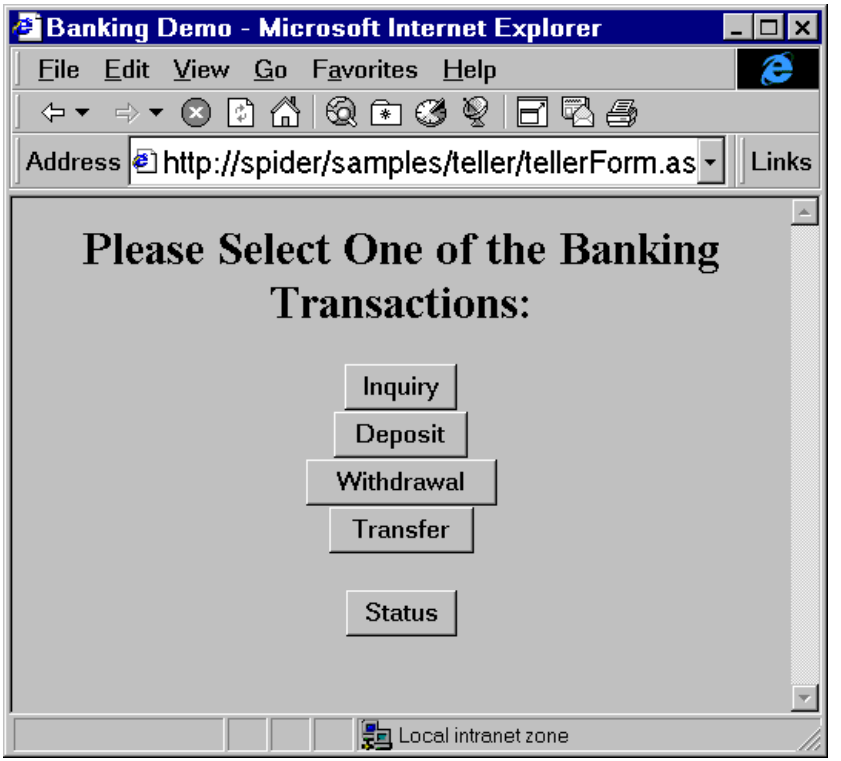
Initializing the Jolt Session Pool Manager

To start the walkthrough, use the browser on your client to connect to the Web server where the Jolt Asp Connectivity For Tuxedo classes are installed. The first page to download is `tellerForm.asp` (see the following figure for an example of a `tellerForm.asp` page). If the teller sample has been installed as described in step 6 of the "Getting Started Checklist," the URL for this page will be:

`http://<web-server:port>/teller/tellerForm.asp`

Note: The use of the port number is optional, depending on how your Web server is configured. In most cases, you are not required to add the ":port" in the URL.

Figure 7-2 tellerForm.asp Example



The page, `tellerForm.asp` contains VBScript procedures required to initialize the Jolt Session Pool Manager. The initialization code is contained in an ASP Script block. This code tells the Web server to execute this block of code on the server, instead of sending it to the client.

Listing 7-1 tellerForm.asp: Initialize the Jolt Session Pool Manager

```
<%
'// Initialize the session manager and cache templates
Call web_initSessionMgr(Null)
Call web_cacheTemplates()
%>
```

The VBScript procedure `web_initSessionMgr()` calls other VBScript procedures to establish a pool of Jolt Sessions. A Jolt Session is established between the Jolt ASP Connectivity For Tuxedo in the Web Server and the Jolt Servers that reside in your Tuxedo application. One of the procedures called is `web_start()`. This procedure (in the file `web_start.inc`) should have been edited as part of the teller application installation process in step 6 of the "Getting Started Checklist".

The procedure `web_cacheTemplates()` reads various HTML template files into a memory cache. This step is not required, but it improves performance.

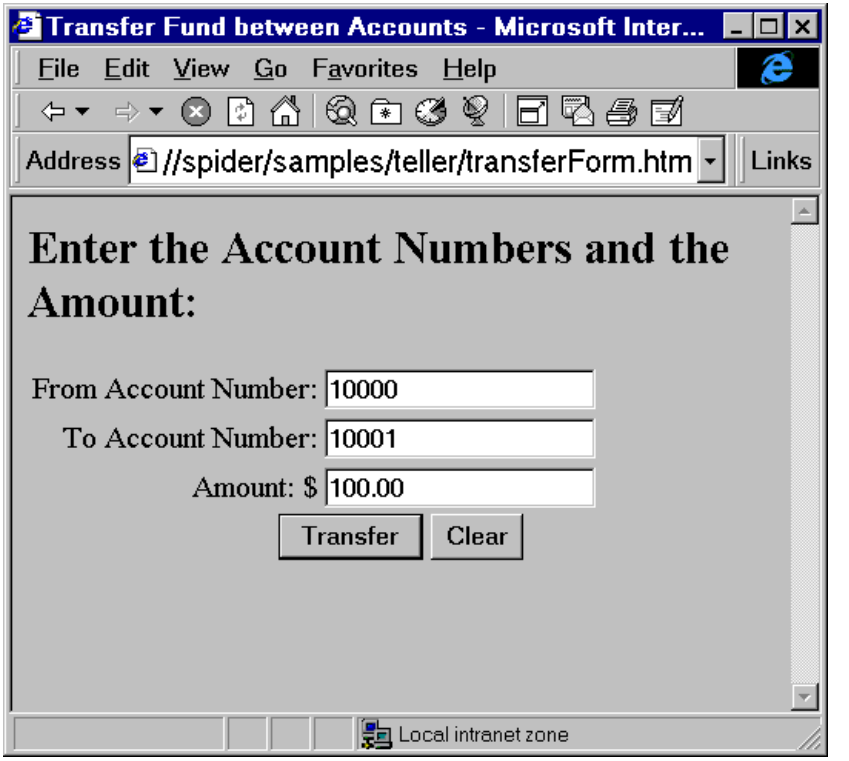
Listing 7-2 tellerForm.asp: Allow the user to choose TRANSFER service

```
<INPUT TYPE="button" VALUE="Transfer"
    onClick="window.location='transferForm.htm'">
```

The HTML segment above displays a button labeled "Transfer." When this button is selected, the browser loads the page `transferForm.htm`. This page presents a form used to enter the data required by the TRANSFER service.

Submitting a TRANSFER Request from the Client

Figure 7-3 transferForm.htm Example



The form in the previous figure is generated by the page `transferForm.htm`. This page presents you with a form for input. The page consists of three text fields (two account numbers and a dollar amount), and a button that, when pressed, causes the TRANSFER service to be invoked.

The code segment in the following figurethe following listing shows the key HTML elements for this page. The **highlighted** elements in the following listing correspond to the elements in the following table.

Listing 7-3 transferForm.htm: TRANSFER Form

```
<FORM NAME="teller" ACTION="tlr.asp" METHOD="POST">
<TABLE>
<TR><TD ALIGN=RIGHT>From Account Number: </TD>
    <TD><INPUT TYPE="text" NAME="ACCOUNT_ID_0"></TD></TR>
<TR><TD ALIGN=RIGHT>To Account Number: </TD>
    <TD><INPUT TYPE="text" NAME="ACCOUNT_ID_1"></TD></TR>
<TR><TD ALIGN=RIGHT>Amount: $</TD>
    <TD><INPUT TYPE="text" NAME="SAMOUNT"></TD></TR>
</TABLE>
<CENTER>
<INPUT TYPE="hidden" NAME="SVCNAME" VALUE="TRANSFER">
<INPUT TYPE="submit" VALUE="Transfer">
<INPUT TYPE="reset" VALUE="Clear">
</CENTER>
</FORM>
```

Table 7-2 Key HTML Elements and Descriptions

Element	Description
`ACTION="tlr.asp"`	When the "submit" button is pressed, the contents of this form are delivered to a page called `tlr.asp` on the Web server for processing.
`NAME="ACCOUNT_ID_0"`	Shows the use of a field with multiple occurrences. The TRANSFER service expects two input account numbers, both called "ACCOUNT_ID". By using a convention of appending an *underscore* and *occurrence_number* (e.g., _0, _1) to the field name, both the name of a field and its occurrence can be passed to the program on the Web Server.
`NAME="SAMOUNT"`	Shows the use of an input field that has a single occurrence. In this example, there is nothing appended to the name of the field.

The HTML form field names used in this example exactly match the Tuxedo field names expected by the TRANSFER service. This is not required, but doing so facilitates processing on the server because you do not have to map these inputs to Tuxedo field names. This is done by the Jolt ASP Connectivity For Tuxedo classes.

The hidden field SVCNAME is assigned a value of TRANSFER. This field does not appear on the client form, but it is sent to the Web server as part of the request. The VBScript program retrieves the value of this field in order to determine which Tuxedo service is to be called (in this example, the service is TRANSFER).

Complete the fields `From Account Number`, `To Account Number`, and `Amount`. (10000 and 10001 are valid bankapp account numbers). Press the "Transfer" button. The data entered on the form is sent to the Web server for processing by the program `tlr.asp` as specified in the ACTION field of the form.

Processing the Request

When the Web server receives the TRANSFER request, it runs the program `tlr.asp`. Client requests are turned into a Request object in the Web server. This Request object has members containing all the data that was input to the form along with other form data, such as hidden fields. The Web server makes the Request object available to the program being invoked.

The program `tlr.asp` contains only VBScript. The first action performed by this program verifies that the Jolt Session Pool Manager is initialized. The code example shown in the following listing performs the initialization check and returns an HTML error page if the pool is not initialized.

Listing 7-4 tlr.asp: Verify the Jolt Session Pool Manager is Initialized

```
<%
If Not IsObject(Application("mgr")) Then
%>
    <HTML>
    <HEAD><TITLE>Error</TITLE></HEAD>
    <BODY><CENTER>
    <H2>Session Manager is not initialized</H2>
    <P>Make sure that you access the correct HTML
    </CENTER></BODY>
    </HTML>
<%
End If
%>
```

If the session pool is initialized, the program continues to process the request. The program locates a Session from the Session Pool Manager shown in the following listing.

Listing 7-5 tlr.asp: Locate a Session

```
Set pool = Application("mgr").getSessionPool(Null)
```

Once a valid session is located, the program retrieves an HTML template that is used to return the results to the client. In this example, these templates were cached in the initialization section. The template retrieved is identified by the name of the service being invoked, `Request("SVCNAME")` shown in the following listing.

Listing 7-6 tlr.asp: Retrieve a Cached HTML Template

```
'// Choose the response template
If IsEmpty(Application("templates")) Then
   Set template = Server.CreateObject("BEAWEB.Template")
Else
   Select Case Request("SVCNAME")
      Case "INQUIRY"
         Set template = Application("templates")(INQUIRY)
```

```
      Case "DEPOSIT"
         Set template = Application("templates")(DEPOSIT)
      Case "WITHDRAWAL"
         Set template = Application("templates")(WITHDRAWAL)
      Case "TRANSFER"
         Set template = Application("templates")(TRANSFER)
   End Select
End If
```

Next, call the Tuxedo service. In this example, the input data from the Request object is passed to the `call()` method of the session. The `call()` method uses the built-in ASP Request object as input. The results of the `call()` are stored in the `output` object and an array, `iodata`.

Listing 7-7 tlr.asp: Invoke the Tuxedo Service

```
Set output = pool.call(Request("SVCNAME"), Null, Nothing)
Set iodata(1) = output
```

After you invoke the Tuxedo service, the `output` object and the second element of the array `iodata` contain the results of the service call.

Note: In this example, because the initial form specified field names match the Tuxedo service parameter names, the Request object can be used in the `call()` method. If these names do not match, create an input array with "name=value" elements for each service parameter before invoking the `call()` method.

Returning the Results to the Client

At this stage, no results have been returned to the client. The final step sends an HTML page containing the results of the service call back to the client. The HTML page consists of the template merged with the data returned by the service call shown in the previous listing.

The template file contains placeholders for variable (call-specific) data. These placeholders are identified by the special tag <%=NAME%>. In the code example shown in the following listing, an index is used to indicate which occurrence of a parameter name is used. For example, ACCOUNT_ID[0] specifies the first occurrence of the field ACCOUNT_ID.

Listing 7-8 transfer.temp: Placeholders for TRANSFER Results

```
<TABLE BORDER=1>
<TR><TD></TD><TD ALIGN=CENTER><B>Account #</B></TD>
   <TD ALIGN=CENTER><B>Balance</B></TR>
<TR><TD ALIGN=RIGHT><B>From:</B></TD><TD><%=ACCOUNT_ID[0]%></TD>
   <TD><%=SBALANCE[0]%></TR>
<TR><TD ALIGN=RIGHT><B>To:</B></TD><TD><%=ACCOUNT_ID[1]%></TD>
   <TD><%=SBALANCE[1]%></TR>
</TABLE>
```

To substitute the placeholders in the template with the actual values of the data returned from the service call, use the eval() method of the Template object shown in the following listing. This method matches placeholders in the template file with fields of the same name in the results data and replaces them accordingly. A check for valid results (output object) is done as shown in the following listing. If there is no output object, an error template page is returned.

Listing 7-9 tlr.asp: Template Processing

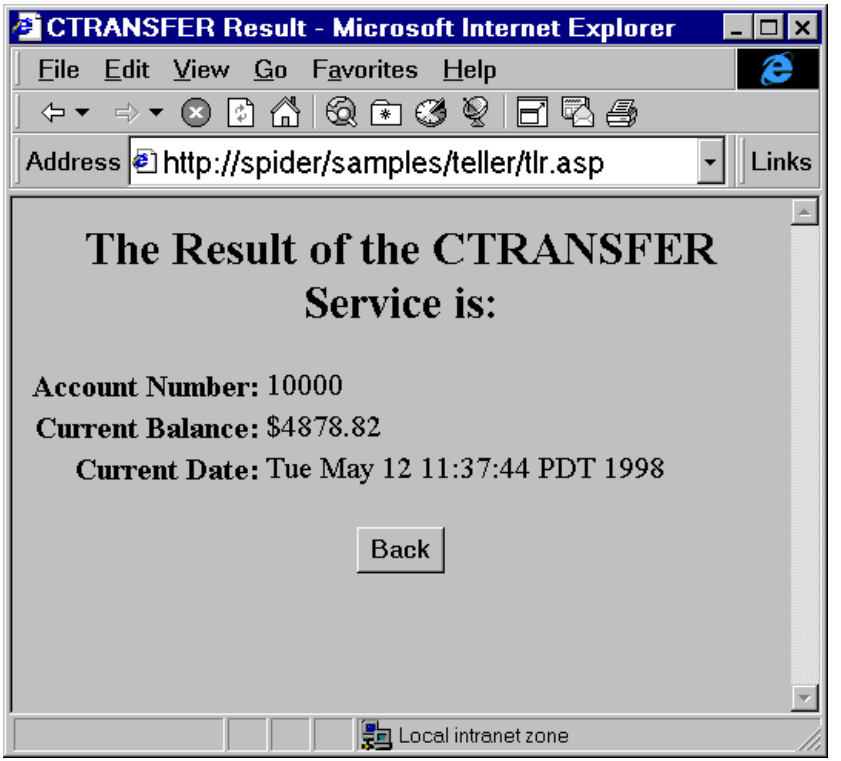
```
path = Application("templatedir")
If (Not IsObject(output)) Or (output is Nothing) Then
    Call template.evalFile(path & "\nosession.temp", Null)
Elseif output.noError() Then
    Call template.eval(iodata)
Elseif output.applicationError() Then
    Call template.evalFile(path & "\error.temp", iodata)
Else
    '// System error
    Dim errdata(0)
    Set errdata(0) = Server.CreateObject("BEAWEB.TemplateData")
    Call errdata(0).setValue("ERRNO", output.getError())
    Call errdata(0).setValue("ERRMSG", output.getStringError())
```

```
    Call template.evalFile(path & "\syserror.temp", errdata)
End If
```

Note: The array `iodata` contains both the input request and the results from the service call. This is useful if you want the results page to contain data that is part of the input.

When the template is processed, the resulting HTML is returned to the client as shown in the following figure.

Figure 7-4 tlr.asp Results Page



A Tuxedo Errors

This appendix describes the Jolt Class Library errors and exceptions. The Jolt Class Library returns both Jolt and Tuxedo errors and exceptions. The Jolt Class Library errors and exceptions are also listed for each class, constructor, and method listed in the API Reference in Javadoc. Tuxedo Errors are described briefly in this appendix. For a complete explanation of Tuxedo errors, refer to *BEA Tuxedo System Messages.*

Tuxedo Errors

Expanded references to Tuxedo will be available in a future release of the Jolt product documentation. If you require an immediate, expanded reference for Tuxedo related errors, see the *Tuxedo System Reference Manual*.

Table A-1 Tuxedo Errors

Error	Description
TPEABORT	A transaction could not commit because the work performed by the initiator, or by one or more of its participants, could not commit.
TPEBADDESC	A call descriptor is invalid or is not the descriptor with which a conversational service was invoked.
TPEBLOCK	A blocking condition exists and TPNOBLOCK was specified.
TPEDIAGNOSTIC	Dequeuing a message from the specified queue failed. The reason for failure can be determined by the diagnostic value returned through *ctl* structure.
TPEEVENT	An event occurred; the event type is returned in revent.
TPEHAZARD	Due to a failure, the work done on behalf of the transaction can have been heuristically completed.
TPEHEURISTIC	Due to a heuristic decision, the work done on behalf of the transaction was partially committed and partially aborted.
TPEINVAL	An invalid argument was detected.
TPEITYPE	The type and subtype of the input buffer is not one of the types and subtypes that the service accepts.
TPELIMIT	The caller's request was not sent because the maximum number of outstanding requests or connections has been reached.
TPEMATCH	svcname is already advertised for the server but with a function other then func.
TPEMIB	The administrative request failed. *outbuf* is updated and returned to the caller with FML32 fields indicating the cause of the error as is discussed in **MIB(5)** and **TM_MIB(5)**.

Table A-1 Tuxedo Errors

Error	Description
TPENOENT	Cannot send to svc because it does not exist or is not the correct type of service.
TPEOS	An operating system error has occurred.
TPEOTYPE	The type and subtype of the reply are not known to the caller.
TPEPERM	A client cannot join an application because it does not have permission to do so or because it has not supplied the correct application password.
TPEPROTO	A library routine was called in an improper context.
TPERELEASE	`tpadmcall()` was called with the TUXCONFIG environment variable pointing to a different release version configuration file.
TPERMERR	A resource manager failed to open or close correctly.
TPESVCERR	A service routine encountered an error either in `tpreturn`(3) or `tpforward`(3). For example, bad arguments were passed.
TPESVCFAIL	The service routine sending the caller's reply called.
TPESYSTEM	A System/T error occurred.
TPETIME	A time-out occurred.
TPETRAN	The caller cannot be placed in transaction mode.
TPGOTSIG	A signal was received and TPSIGRSTRT was not specified.

B System Messages

Jolt system messages and code references will be available in a future release of the Jolt product documentation. If you require an immediate, expanded reference, refer to *BEA Tuxedo System Messages*.

This appendix includes:

- Jolt System Messages
- Repository Messages
- FML Error Messages
- Information Messages
- Jolt Relay Adapter (JRAD) Messages
- Jolt Relay (JRLY) Messages
- Bulk Loader Utility Messages

Jolt System Messages

Note: You can find error messages numbered 1000 to 1299 in the *BEA Tuxedo System Message Manual, Volume 2*, under "WSNATIVE MESSAGES (WSNAT_CAT).

1503 ERROR	**Could not initialize Jolt administration services.**	
	Description	Jolt administration services cannot be started.
	Action	Check the userlog for other messages to determine the proper course of action.
	See Also	*Tuxedo Administration Guide*
1504 ERROR	**Failed to advertise local Jolt administration service <service name>.**	
	Description	Jolt administration services cannot be started.
	Action	Check the userlog for other messages to determine the proper course of action.
	See Also	*Tuxedo Administration Guide*
1505 ERROR	**Failed to advertise global Jolt administration service <service name>.**	
	Description	Jolt administration services cannot be started.
	Action	Check the userlog for other messages to determine the proper course of action.
	See Also	*Tuxedo Administration Guide*

1506 ERROR	**Terminating Jolt administration services in preparation for shutdown.**	
	Description	The JSL has completed its shutdown and is exiting the system.
	Action	Informational message, no action required.
	See Also	*Tuxedo Administration Guide*
1510 ERROR	**Received network message with unknown context.**	
	Description	BEA Jolt protocol failure. Received a corrupted or an improper message.
	Action	Restart Jolt client.
1511 ERROR	**_tprandkey() failed tperrno = %d, could not generate random encryption key.**	
	Description	Tuxedo internal failure.
	Action	Restart Jolt servers.
1512 ERROR	**Sending of reply to challenge call to client failed.**	
	Description	JSH was unable to reply to Jolt client due to network error.
	Action	Restart client.
1513 ERROR	**Failed to encrypt ticket information.**	
	Description	BEA Tuxedo internal failure.
	Action	Retry the option. If the problem persists, contact BEA Technical Support.
1514 ERROR	**Incorrect ticket value sent by workstation client.**	
	Description	BEA Jolt protocol failure.
	Action	Retry the option. If the problem persists, contact BEA Technical Support.

1515 ERROR	**Tried to process unexpected message opcode 0x%1x.**	
	Description	BEA Jolt protocol failure. Client is sending Jolt messages with unknown opcodes.
	Action	Retry the option. If the problem persists, contact BEA Technical Support.
1516 ERROR	**Unrecognized message format, release %1d.**	
	Description	BEA Jolt protocol failure.
	Action	Make sure the client classes are at the appropriate version level.
1517 ERROR	**Commit handle and clientid have no matching requests.**	
	Description	Received a copy from Tuxedo that has no corresponding client.
	Action	No action required.
1518 ERROR	**Call handle and clientid have no matching requests.**	
	Description	Received a reply from Tuxedo that has no corresponding client.
	Action	No action required.
1519 ERROR	**Application password does not match.**	
	Description	Authentication error.
	Action	Check the application password.
1520 ERROR	**Init handle and clientid have no matching requests**	
	Description	A reply could not be sent to client. (May be due to client disconnect.)
	Action	No action required.
1521 ERROR	**Unrecognized message magic %ld.**	
	Description	Inappropriate message is sent to JSH/JSL.
	Action	Check the client sending erroneous messages.

1522 ERROR	**Memory allocation failure.**	
	Description	Machine does not have enough memory.
	Action	Check the machine resources.
1523 ERROR	**Memory allocation failure.**	
	Description	Machine does not have enough memory.
	Action	Check the machine resources.
1524 ERROR	**Failed to create encryption/decryption schedule.**	
	Description	BEA Tuxedo internal error.
	Action	Retry the option. If the problem persists, contact BEA Technical Support.
1525 ERROR	**Tried to process unexpected message opcode 0x%1x.**	
	Description	Received a message with invalid opcode.
	Action	Check the client.
1526 ERROR	**Jolt license has expired.**	
	Description	License for Jolt use has expired.
	Action	Contact BEA Technical Support.
1527 ERROR	**Expected argument to -c option.**	
	Description	Option -c needs an argument.
	Action	Provide a valid argument.
1528 ERROR	**Request for inappropriate session type.**	
	Description	Received a message without valid session information.
	Action	Restart the client.

1529 ERROR	**Session type must be RETAINED or TRANSIENT.**	
	Description	Server configuration does not match client request.
	Action	Check the -c argument of the JSL.
1530 ERROR	**Received RECONNECT message with invalid context.**	
	Description	Client context is cleaned. A -T option is specified to the JSL.
	Action	Check the -T option. Check the network errors also.
1531 ERROR	**Received invalid RECONNECT request**	
	Description	Received a RECONNECT request.
	Action	Restart client.
1532 ERROR	**Received J_CLOSE message with invalid context.**	
	Description	Timeout in connection.
	Action	If a request is sent after a timeout that is longer than the session timeout of the JSL, the JSH cannot validate the session ID.
1533 ERROR	**Sending of reply of close protocol failed.**	
	Description	BEA Jolt protocol failure.
	Action	Check the client.
1534 ERROR	**Sending of reply of reconnect protocol failed.**	
	Description	BEA Jolt protocol failed.
	Action	Check the client.
1535 ERROR	**Timestamp mismatch in close protocol.**	
	Description	BEA Jolt protocol failed.
	Action	Restart the client.

1536 ERROR	**Received J_RECONNECT message with invalid context.**	
	Description	BEA Jolt protocol failed. Session timed out before RECONNECT request arrived.
	Action	Restart the client.
1537 ERROR	**Timestamp mismatch in reconnect protocol.**	
	Description	BEA Jolt protocol failure.
	Action	Restart the client.
1538 ERROR	**Client address mismatch in reconnect protocol.**	
	Description	BEA Jolt protocol failure.
	Action	Restart the client.
1539 ERROR	**Failed to decrypt reconnect information.**	
	Description	BEA Jolt protocol failure.
	Action	Restart the client.
1540 ERROR	**Failed to encrypt reconnect information.**	
	Description	BEA Jolt protocol failure.
	Action	Restart the client.
1541 ERROR	**Received RECONNECT request for nonTRANSIENT client.**	
	Description	Improper request from client.
	Action	Restart the client.
1542 ERROR	**Unlicensed Jolt server.**	
	Description	The JSL is not licensed. The installation is incomplete, or it failed to burn the license into the JSL.
	Action	Reinstall Jolt with a valid Jolt license.

1543 ERROR	**Invalid Jolt license.**	
	Description	The license used for the Jolt installation is not for the Jolt product. The Tuxedo license may have been used during installation instead of the Jolt license.
	Action	Reinstall Jolt with a valid Jolt license.
1544 ERROR	**This Tuxedo is not Release <Tuxedo release number>.**	
	Description	Jolt is compatible with Tuxedo Release 6.1 or 6.2. The JSL has determined that the Tuxedo release is not compatible.
	Action	Install Tuxedo 6.1 or Tuxedo 6.2.
1545 ERROR	**Cannot determine if this Tuxedo is <Tuxedo release number>: service.TMIB failed.**	
	Description	This version of Tuxedo does not support the MIB. The Tuxedo release may be Tuxedo 6.0 or earlier.
	Action	Install Tuxedo 6.1 or 6.2 or check to ensure that your Tuxedo release is 6.1 or 6.2.
1546 WARN	**The version of this Tuxedo is not available; <Tuxedo release number> is assumed.**	
	Description	The MIB is supported with this version of Tuxedo, but the release number is unavailable. The Tuxedo version might not be a master binary. It might also be an internal version of Tuxedo.
	Action	No action is required.
1547 ERROR	**Memory allocation failure in JOLT_SUBSCRIBE.**	
	Description	Check resources of the machine.
	Action	Restart Tuxedo after increasing system resources.

1548 ERROR	**jolt_tpset_enq failed.**	
	Description	Internal system failure.
	Action	Restart the client. If problem persists, check field table files and directories and then restart the servers.
1549 ERROR	**[JOLT_EVENTS failed to set %s field. Ferror32=%d].**	
	Description	Unable to get the field definition for Tuxedo internal fields.
	Action	Check Tuxedo installation and restart the servers.
1550 ERROR	**JOLT_UNSUBSCRIBE - Invalid Subscription ID.**	
	Description	Application error.
	Action	Check the client and restart the client.
1551 ERROR	**Memory allocation failure in JOLT_UNSUBSCRIBE.**	
	Description	Resources are not enough.
	Action	Increase resources and restart Tuxedo.
1552 WARN	**Dropping notification message for Transient client %d.**	
	Description	Notification arrived when a transient client is not connected.
	Action	Information message only; no action required.
1553 WARN	**Dropping broadcast message for Transient client %d.**	
	Description	Notification arrived when a transient client is not connected.
	Action	Information message only; no action required.

1554 ERROR	**Expected numeric argument for -Z option.**	
	Description	-Z option expects 0, 40, or 128 as the argument.
	Action	Check the configuration file and specify a valid numeric argument for JSL.
1555 ERROR	**%d - Illegal argument for -Z option.**	
	Description	Incorrect argument value is specified.
	Action	Check the argument for -Z option and correct it.
1556 ERROR	**%d - Illegal argument for -Z option due to international license.**	
	Description	For international release only 0 or 40 are allowed.
	Action	Specify correct argument.
1557 ERROR	**Incorrect number of encrypted bit values from workstation client.**	
	Description	BEA Jolt protocol failure.
	Action	Call BEA Technical Support.
1558 ERROR	**Expected argument to -E option.**	
	Description	An argument is expected for `-E` option.
	Action	Specify correct option and restart Tuxedo.
1559 ERROR	**%s - Illegal argument to -E option.**	
	Description	Incorrect value is specified as argument to `-E` option.
	Action	Specify the correct option.
1560 ERROR	**Cannot initialize the code conversion for local %s.**	
	Description	Cannot find function to do the code conversion for internationalization.
	Action	Check the shared library.

1561 ERROR	**TUXDIR is not set.**	
	Description	TUXDIR environment variable is not set.
	Action	Set the variable to Tuxedo directory and restart Tuxedo.
1562 ERROR	**Error reading license file.**	
	Description	Jolt is not able to open Tuxedo license file in $TUXDIR/udataobj/lic.txt.
	Action	Copy the correct license file to $TUXDIR/udataobj/lic.txt.
1563 INFO	**Serial Number: <%s>, Expiration Date: <%s>.**	
	Description	Serial number and expiration date displays.
	Action	No action required.
1564 INFO	**Licensee: <%s>.**	
	Description	Licensee information displays.
	Action	No action required.
1565 ERROR	**Call handle and clientid have no matching requests.**	
	Description	Received a reply from Tuxedo that has no corresponding client.
	Action	No action required.
1566 INFO	**Message received without handle, ignored.**	
	Description	A Tuxedo message arrived without an identifying handle.
	Action	No action required.

1567 ERROR	**Expected argument to -j option.**	
	Description	`-j` requires an argument.
	Action	Specify `-j` argument (ANY/RETAINED/RECONNECT) in UBB and reboot Tuxedo system.
1568 INFO	**Compression threshold is set to %d.**	
	Description	Informative message.
	Action	No action required.
1569 ERROR	**No Tuxedo Encryption installed. Cannot use Diffie-Hellman.**	
	Description	Cannot find encryption libraries.
	Action	Contact Tuxedo support.
1570 WARN	**Jolt Client Connection Request timed out.**	
	Description	Jolt client sent connect request for JSH too late.
	Action	If problem persists, increase the value of `-I` option in JSL.
1571 WARN	**A Jolt Client has incorrect APPADDR.**	
	Description	A Jolt client has specified JSH address instead of JSL.
	Action	Change the client and specify correct address.
1572 WARN	**A Non Jolt Opcode is sent to JSH.**	
	Description	A request received by JSh has non Jolt opcode.
	Action	Check client's APPADDR.

Repository Messages

ERROR	**Usage: JREPSVR [-W] -P path -W writable repository.**	
	Description	An invalid option is specified or `-P` is not specified properly.
	Action	Review the Jolt documentation and ensure that the options are specified correctly.
ERROR	**Not enough memory**	
	Description	Not enough memory; please add more swap space.
	Action	Configure additional memory. Make sure the operating system parameters are set correctly for the amount of memory on the machine and the amount of memory that can be used by a process. Reduce the memory usage on the machine or increase the amount of physical memory on the machine.
ERROR	**Not enough disk space for "<repository-file-path>"**	
	Description	Ran out of disk space while adding or deleting Repository entries, or during garbage collection.
	Action	Configure additional disk space.
ERROR	**Cannot modify read-only repository "<repository-file-path>"**	
	Description	Denies attempt to add or delete an entry from a read-only repository.
	Action	Check the file permission and ensure that the file is writable.
ERROR	**"<repository-file-path>" is not a valid repository file.**	
	Description	The specified file is not valid; a valid repository file must have the string, "#!JOLT1.0" in the first line.
	Action	Extract the file from the Jolt distribution CD-ROM.

ERROR	**Can't open <repository-file-path>.**	
	Description	Unable to open the repository file.
	Action	Check to ensure that the file path is valid or its permission is correct.
ERROR	**Can't create <repository-file-path>: check permission or path.**	
	Description	Unable to create the repository file during garbage collection.
	Action	Check the file or directory permission.
ERROR	**Syntax error: <service definition>.**	
	Description	An invalid entry was detected when an attempt was made to add an entry to the repository. The entry must have ':' as a field separator.
	Action	Contact BEA Technical Support.
ERROR	**Garbage collection failed: <key> not found.**	
	Description	When the writable repository is shutdown, it performs garbage collection to collapse the repository file. If it detects an inconsistency, the garbage collection fails.
	Action	Contact BEA Technical Support.

FML Error Messages

ERROR	**Fielded buffer not aligned.**	
	Description	An FML function was called with a fielded buffer that is not properly aligned. Most machines require half-word alignment.
	Action	Use `Falloc` to retrieve an allocated, properly aligned buffer.
	See Also	*Tuxedo Reference Manual*
ERROR	**Buffer not fielded.**	
	Description	A buffer was passed to an FML function that has not been initialized.
	Action	Use `Finit` to initialize a buffer allocated directly by the application, or use `Falloc` to allocate and initialize a fielded buffer.
	See Also	*Tuxedo Reference Manual*
ERROR	**Invalid argument to function.**	
	Description	An invalid argument (other than an invalid field buffer, field identifier, or field type) was passed to an FML function. This can be a parameter where a non-NULL parameter was expected (for example, it can be an invalid buffer size, etc.).
	Action	See the manual page associated with the error for the correct parameter values.
	See Also	*Tuxedo Reference Manual*

ERROR	Unknown field number or type.	
	Description	An invalid field number was specified for an FML function, an invalid field number (0 or greater than 8192) was specified, or `Fname` could not find the associated field identifier for the specified name.
	Action	Most of the FML functions return this error; see the manual page associated with the function that returned this error. Check your code to make sure the field specified is valid.
	See Also	*Tuxedo Reference Manual*

Information Messages

INFO	**Repository “<repository-file-path>” (### records) is writable.**	
	Description	When a writable Repository server is brought up, it reports the number of records it found.
	Action	No action required.
INFO	**Repository “<repository-file-path>” (### records) is read-only.**	
	Description	When a read-only Repository server is brought up, it reports the number of records it found.
	Action	No action required.

Jolt Relay Adapter (JRAD) Messages

Note: You can find error messages numbered 1000 to 1299 in the *BEA Tuxedo System Message Manual, Volume 2*, under "WSNATIVE MESSAGES (WSNAT_CAT).

1500 ERROR	**Needs both -l -c options with arguments.**	
	Description	Needed options are without arguments.
	Action	Check and correct configuration file for JRAD entry.
1501 ERROR	**Malloc failed.**	
	Description	JRAD is not able to allocate dynamic memory.
	Action	Increase the system resources and restart the JRAD.
1502 ERROR	**Memory allocation failed.**	
	Description	JRAD is not able to allocate dynamic memory.
	Action	Increase the system resources and restart the JRAD.
1503 ERROR	**Memory allocation failed. Cannot send ESTCON.**	
	Description	JRAD is not able to allocate dynamic memory.
	Action	Increase the system resources and restart the JRAD.

1504 INFO	**Memory allocation failed. Cannot send ESTCON.**	
	Description	JRAD is not able to allocate dynamic memory.
	Action	Increase the system resources and restart the JRAD.
1505 ERROR	**Memory allocation failed. Cannot send ESTCON.**	
	Description	JRAD is not able to allocate dynamic memory.
	Action	Increase the system resources and restart the JRAD.
1506 ERROR	**Connection to JSL failed.**	
	Description	JSL is not running.
	Action	Check the address given with option `-c`.
1507 ERROR	**Sending message to JSL failed.**	
	Description	JSL is not running or network connection is down.
	Action	Restart the JRAD/JSL.
1508 INFO	**Sending message to JSH failed.**	
	Description	Network is down. Connection to the JSH failed.
	Action	Check the network and restart the JSL.
1509 ERROR	**Sending CONNECT reply to JRLY.**	
	Description	Unable to reach JRLY. Probably problem in the network.
	Action	Restart the JRLY and JRAD after check the network addresses.

1510 ERROR	**Sending SHUTDOWN reply to JRLY.**	
	Description	Unable to reach JRLY. Probably problem in the network.
	Action	Restart the JRLY and JRAD after check the network addresses.
1511 ERROR	**Incorrect Jolt message received from JRLY.**	
	Description	A non Jolt message is sent by JRLY.
	Action	No action required. JRLY process filters non Jolt messages already.
1512 ERROR	**Sending SHUTDOWN to JRLY failed.**	
	Description	Unable to send shutdown message to JRLY.
	Action	No action required.
1513 ERROR	**Sending CLOSE to JRLY failed for ID <%d>.**	
	Description	Unable to send CLOSE message for Relay ID to JRLY.
	Action	No action required.
1514 ERROR	**Sending CLOSE to JRLY failed.**	
	Description	Unable to send CLOSE message for Relay ID to JRLY.
	Action	No action required.
1515 ERROR	**Sending CLOSE to JRLY failed for ID <%d>.**	
	Description	Unable to send CLOSE message for Relay ID to JRLY.
	Action	No action required.
1516 ERROR	**Sending ESTCON to JRLY failed for ID <%d>.**	
	Description	Sending ESTCON message failed.
	Action	No action required.

1517 ERROR	**Invalid Handler Id. No corresponding address.**	
	Description	JRAD received a message without JSH identification.
	Action	No action required.
1518 ERROR	**Cannot connect to JSH with id <%d>.**	
	Description	JRAD received a message without JSH identification.
	Action	No action required.
1519 ERROR	**Invalid request from JRLY.**	
	Description	JRAD received a message without JSH identification.
	Action	No action required.
1521 ERROR	**JRLY connection is down.**	
	Description	JRLY connection is down.
	Action	No action required.
1522 ERROR	**JRLY connection is down.**	
	Description	JRLY connection is down.
	Action	No action required.
1523 ERROR	**JRLY connection is down.**	
	Description	JRLY connection is down.
	Action	No action required.
1525 ERROR	**JRLY connection is down.**	
	Description	JRLY connection is down.
	Action	No action required.

1526 INFO	**JRLY connection is UP.**	
	Description	A JRLY-JRAD connection is established.
	Action	No action required.
1531 ERROR	**Sending R_CLOSE | R_ACK failed.**	
	Description	Failed to send Relay protocol ack.
	Action	No action required.
1532 INFO	**JRLY connection is closed.**	
	Description	JRLY connection is down.
	Action	No action required.
1533 ERROR	**Bad hex number provided for external jrly address: %s.**	
	Description	Invalid `-H` option value.
	Action	Check `-H` option and provide correct value.
1534 ERROR	**Convert external jrly address to hex format failed: %s.**	
	Description	Invalid `-H` option value.
	Action	Check `-H` option and provide correct value.
1535 ERROR	**Bad hex number provided for connecting address: %s.**	
	Description	Invalid `-c` option value.
	Action	Check `-c` option and provide correct value.
1536 ERROR	**address conversion failed.**	
	Description	Invalid `-c` option value.
	Action	Check `-c` option and provide correct value.
1537 WARN	**Convert listening address to hex format failed: %s.**	
	Description	Invalid `-l` option value
	Action	Check `-l` option and provide correct value.

1538 WARN	**Convert connecting address to hex format failed: %s.**	
	Description	Invalid `-c` option value.
	Action	Check `-c` option and provide correct value.
1539 WARN	**Refusing connection to JRAD. JRLY connection exists.**	
	Description	A second JRLY is trying to connect to JRAD. Connection is refused by JRAD.
	Action	Provide correct CONNECT address for JRLY.
1540 WARN	**No JRLY process connected.**	
	Description	A dubious message arrived for JSL/JSH with no relay connected.
	Action	Check the network address in configuration.

Jolt Relay (JRLY) Messages

The following table lists the Jolt Relay messages.

ERROR	**Ignoring syntax error in configuration file line %d**	
	Description	The line in question doesn't contain an equal sign or (in case of the LISTEN and CONNECT tag) is missing the colon.
	Action	Verify the syntax of the configuration file at the specified line.
ERROR	**Ignoring unknown tag '%s' in configuration file line %d.**	
	Description	The line in question is does not contain one of the valid tags: LOGDIR, ACCESS_LOG, ERROR_LOG, LISTEN, CONNECT.
	Action	Verify the syntax of the configuration file at the specified line.
ERROR	**MSG_MALLOC: perror().**	
	Description	Memory allocation failed. The relay will exit.
	Action	Make more memory available on the machine on which the relay is running. Remove other unnecessary processes which may be running on the same host as the relay. Restart the relay.
ERROR	**Client structure != NULL for file descriptor %ld**	
	Description	An internal error occurred. The relay will continue to run, but a client process may have been disconnected.
	Action	None. If this message appears repeatedly and can be reproduced consistently notify BEA Technical Support.

ERROR	**Invalid file descriptor %ld**	
	Description	An internal error occurred. The relay will continue to run, but a client process may have been disconnected.
	Action	None. If this message appears repeatedly and can be reproduced consistently notify BEA Technical Support.
ERROR	**Could not open configuration file %s**	
	Description	The specified configuration file does not exist or is not readable. The relay will exit.
	Action	Check the file name and the permissions on the file and the directory.
ERROR	**No log directory specified.**	
	Description	LOGDIR was not specified in the configuration file or no value for it was given.
	Action	Verify the entry for the tag LOGDIR in the configuration file. Check that the correct configuration file is being used (`-f` parameter).
ERROR	**No access log file specified.**	
	Description	ACCESS_LOG was not specified in the configuration file or no value for it was given.
	Action	Verify the entry for the tag ACCESS_LOG in the configuration file.Check that the correct configuration file is being used (`-f` parameter).
ERROR	**No error log file specified.**	
	Description	ERROR_LOG was not specified in the configuration file or no value for it was given.
	Action	Verify the entry for the tag ERROR_LOG in the configuration file. Check that the correct configuration file is being used (`-f` parameter).

ERROR	**No JRLY host specified**	
	Description	The value for the LISTEN tag does not contain the host name or IP address or the relay host, e.g., LISTEN=host:port.
	Action	Verify the entry for the tag LISTEN in the configuration file. Check that the correct configuration file is being used (`-f` parameter).
ERROR	**No JRAD host specified.**	
	Description	The value for the CONNECT tag does not contain the host name or IP address or the JRAD host, e.g., CONNECT=host:port.
	Action	Verify the entry for the tag CONNECT in the configuration file. Check that the correct configuration file is being used (`-f` parameter).
ERROR	**No listener port specified or listener port <= 0.**	
	Description	The value for the LISTEN tag does not contain a valid port number on the relay host.
	Action	Verify the entry for the tag LISTEN in the configuration file. Check that the correct configuration file is being used (`-f` parameter).
ERROR	**No JRAD port specified or JRAD port <= 0.**	
	Description	The value for the CONNECT tag does not contain a valid port number on the relay host.
	Action	Verify the entry for the tag CONNECT in the configuration file.Check that the correct configuration file is being used (`-f` parameter).

ERROR	**Could not determine IP address of listener host**
Description	The relay could not look up the IP address of the host machine.
Action	If the host was specified as a host name replace it with the IP address and restart the relay. If it already was given as IP address make sure that the IP address is correct and that you're trying to start the relay on this host. Note that the address specified must be the address of the host on which the relay is running.

ERROR	**Cannot bind socket**
Description	The listener port specified in the configuration file is already being used by another application or still in a final wait state from a previous run of jrly.
Action	Either specify a different port number in the configuration file (and all HTML files containing the IP address and port number of the relay) or wait a few minutes. The command "netstat -a" displays existing connections.

ERROR	**Can’t open log file %s**
Description	Either the error log file or access log file (or both) could not be opened for writing.
Action	Check the configuration file for correct spelling of the LOGDIR. Make sure you have write permissions on this directory and the files specified. On Windows NT, the directory separators must be back slashes, not forward slashes.

ERROR	**WSAStartup failed (NT only)**	
	Description	The Winsock driver could not initialize. Possible causes: ♦ The underlying network subsystem is not ready for network communication Version 2.0 of Windows Sockets support is not provided by this particular Windows Sockets implementation. ♦ Limit on the number of tasks supported by the Windows Sockets implementation has been reached.
	Action	Check the networking software configuration on your system.
ERROR	**Couldn't load Winsock Driver version 2.X. (NT only)**	
	Description	The relay requires Winsock version 2 or higher, but could not load it.
	Action	Check the networking software configuration on your system. An older version of Windows Sockets support was detected.
ERROR	**FATAL ERROR: unknown message code %ld.**	
	Description	Internal error. The relay will exit
	Action	Restart the relay. If this message appears repeatedly and can be reproduced consistently notify BEA Technical Support.
ERROR	**connect: Connection refused**	
	Description	The relay could not connect to JRAD.
	Action	Make sure the relay adapter (JRAD) is running. Check that the CONNECT tag in the relay configuration file identifies the correct host and port on which the JRAD is running.

ERROR	**accept(): accept failed, errno: 24, strerror: Too many open files**	
	Description	The relay tried to open more files/sockets than the system limit.
	Action	The default maximum number of open file descriptors for a process is 64 on most UNIX systems. Set this number to at least 1024 (with the limit or ulimit commands).

Bulk Loader Utility Messages

ERROR	**File not found: %s**	
	Description	The specified file is not found.
	Action	Check the path again.
ERROR	**Error on line %d: %s value is null**	
	Description	A value is expected for this keyword.
	Action	Input the value.
ERROR	**Error on line %d: Invalid keyword: %s=%s**	
	Description	Keyword is not recognized.
	Action	Input the correct keyword value.
ERROR	**Error on line %d: Invalid number: %s**	
	Description	The numeric number is malformed.
	Action	Input the correct value.
ERROR	**Error on line %d: Invalid value: %s**	
	Description	The value of the parameter is out of range.
	Action	Input the correct value.
ERROR	**Error on line %d: Invalid value: %s**	
	Description	The data type of the parameter is invalid.
	Action	Input the correct value.

Index

L

M

N

O

P

R

S

T

U

V

W