Oracle® Tuxedo

Using Oracle Jolt with Oracle WebLogic Server

11*g* Release 1 (11.1.1.3.0)

December 2011

ORACLE

Oracle Tuxedo Using Oracle Jolt with Oracle WebLogic Server, 11*g* Release 1 (11.1.1.3.0)

Copyright © 1996, 2011, Oracle and/or its affiliates. All rights reserved.

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

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

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

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

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

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

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

Contents

1. Introduction to Oracle Jolt for Oracle WebLogic Server

Key Features 1-1
How Jolt for WebLogic Works 1-2
Relationship Between Jolt for WebLogic and Tuxedo. 1-2
Essential Components of the Jolt Architecture 1-3
WebLogic Server Startup 1-4
Connecting to a WebLogic Server from a Client Browser 1-5
How a Servlet Connects to Tuxedo 1-6
What Happens if the Request Fails 1-6
Responding to the Client Browser 1-6
Disconnecting from the Jolt Server 1-7
Using the Example Packages 1-7

2. Configuring Jolt for WebLogic Server

Configuring Jolt for Tuxedo 2-1
Configuring Jolt for WebLogic Server 2-1
Jolt Startup Class and Connection Pool 2-1
Jolt Shutdown Class 2-4
Displaying Jolt in the WebLogic Administration Console 2-5
Resetting the Jolt Connection Pool 2-5
Command-line Method 2-6
Administration Console Method 2-6

3. Implementing Jolt for WebLogic

Importing Packages 3-1
Configuring a Session Pool 3-2
Using a Servlet Session Pool 3-3
Calling a Tuxedo Service 3-4
Sending a ServletDataSet 3-4
Adding Parameters to the Dataset 3-4
Accessing a Tuxedo Service Through Jolt 3-5
Converting Java Data Types to Tuxedo Data Types 3-5
Receiving Results from a Service 3-6
Using the Result.getValue() Method 3-7
Using the ServletResult.getStringValue() Method 3-7
Using a Transaction 3-8
Handling Exceptions 3-8

A. Class Hierarchy

Oracle Jolt Class Hierarchy for the Oracle WebLogic Server API A-1

B. Simple Servlet Example

Example Components and Prerequisites B-1
Using the Example B-2
Step 1. Perform Preparatory Steps B-3
Step 2. Start the WebLogic Server B-4
Step 3. Configure the Servlet in WebLogic Server B-4
Step 4. Stop and Restart the WebLogic Server B-6
Step 5. Compile the Servlet B-6
Step 6. Display the simpapp.html Form B-6
Step 7. Post the FORM Data from the Browser B-7

Step 8. Process the Request B-8
Step 9. Return the Results to the Client B-9

C. Servlet with Enterprise JavaBean Example

About the Servlet with JavaBean Example C-2
Preparing to Use the Servlet with JavaBean Example C-3
Set Up Your Environment C-3
Build the Example C-4
Run the Servlet with JavaBean Example C-4

CHAPTER 1

Introduction to Oracle Jolt for Oracle WebLogic Server

With Oracle Jolt for Oracle WebLogic Server, you can enable Oracle Tuxedo services for the Web, using the Oracle WebLogic Server as the front-end HTTP and application server.

Oracle Jolt is a Java-based client API that manages requests to Oracle Tuxedo services via a Jolt Service Listener (JSL) running on the Tuxedo server. The Jolt API is embedded within the WebLogic API, and is accessible from a servlet or any other Oracle WebLogic application.

Because Oracle Jolt for Oracle WebLogic Server is an extension to the Jolt Java class library, the Jolt Java client class library can be used in HTTP servlets running in the WebLogic Server. Oracle Jolt for Oracle WebLogic Server also uses Java HTTP servlets to provide an interface between HTML browser clients and Oracle Tuxedo services.

Hereafter, Oracle Tuxedo is referred to as *Tuxedo*, Oracle Jolt is referred to as *Jolt*, and Oracle WebLogic is referred to as *WebLogic* for readability.

This topic includes the following sections:

- Key Features
- How Jolt for WebLogic Works
- Using the Example Packages

Key Features

Key features of the Oracle Jolt for Oracle WebLogic Server architecture include:

- Enabling the use of Java HTTP servlets to provide a dynamic HTML front-end for Tuxedo applications
- Providing session pooling to use Tuxedo resources efficiently
- Supporting transactions
- Integrating session pool management into the WebLogic Console

Note: Jolt for WebLogic does not provide access to asynchronous Tuxedo event notifications.

How Jolt for WebLogic Works

This section describes the major components used for communication in Jolt, and how Oracle Jolt for Oracle WebLogic Server works, including:

- How the connection is initialized when the server is started
- The flow of information through:
  - An end-user Web browser
  - The WebLogic Server
  - The Tuxedo transaction processing system

Relationship Between Jolt for WebLogic and Tuxedo

Using Oracle Jolt for Oracle WebLogic Server, you can access your underlying Tuxedo system from the Web. This access allows you to write Web-enabled applications that can interact with other systems and databases in your Tuxedo domain.

The system described here is accessed through a standard Web browser. This Web browser is served by the WebLogic Server, which uses a customized Java HTTP servlet to handle the interactive HTTP requests of the browser. (An HTTP servlet is a Java class that handles an HTTP request and delivers an HTTP response.) The custom HTTP servlet uses the Jolt for WebLogic API to talk to a Jolt Server that can be on a remote machine or behind a security firewall.

The Jolt Server lives within the Tuxedo domain and determines which Tuxedo services are accessible to each client. The Jolt Server invokes the requested Tuxedo service and sends any results back to the WebLogic Server. You can then compile the results into a servlet-generated Web page, and send them to the browser. In doing so, you create a highly accessible and user friendly interface to Tuxedo services from anywhere on the Internet or intranet.

Essential Components of the Jolt Architecture

The fundamental object types that maintain the communications connection from the WebLogic Java HTTP servlet to the Jolt Server and from the Jolt Server to Tuxedo, are as follows:

- *`Session`*

  A session object represents a physical connection with the Tuxedo system.

- *`SessionPool`*

  A session pool contains one or more sessions. The sessions in the session pool are reused for efficiency. Your WebLogic servlet uses sessions to invoke services in Tuxedo through the methods of a session pool. Session pools are initialized by the WebLogic server at startup and configured by attributes in the `config.xml` file.

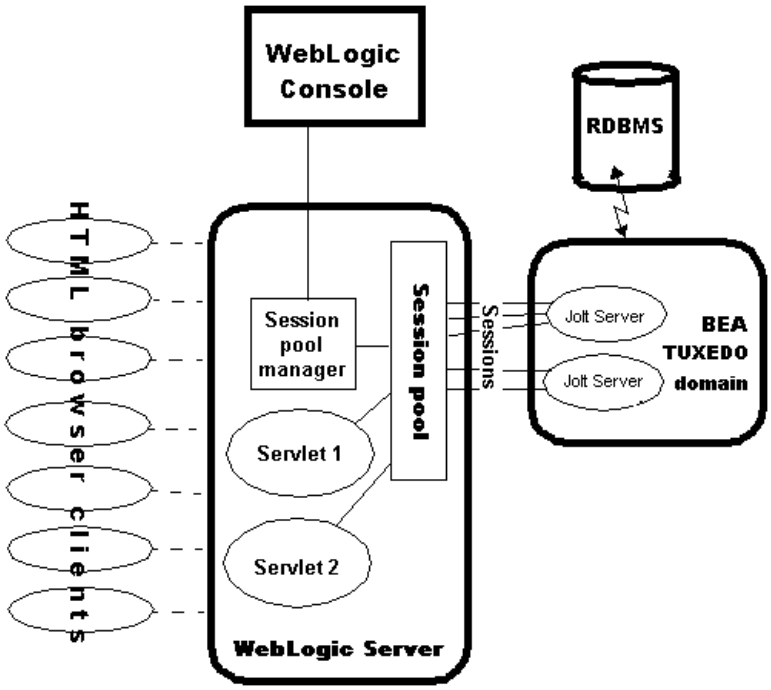
  Note: For Oracle WebLogic Server 6.0 or later, the xml-based `config.xml` configuration file has replaced the `weblogic.properties` file. For more information about the `config.xml` file, refer to the *Oracle WebLogic Server Administration Guide*.

- *`SessionPoolManager`*

  Use the session pool manager to get a reference to a session pool and to create, administer, and remove session pools. The session pool manager is created just before the WebLogic Server initializes the first session pool.

Figure 1-1 shows the architecture for Oracle Jolt for Oracle WebLogic Server.

Figure 1-1 Oracle Jolt for Oracle WebLogic Server Architecture



WebLogic Server Startup

The WebLogic standards-based, pure-Java application server assembles, deploys, and manages distributed Java applications. It supports distributed component services and enterprise database access, including Enterprise JavaBeans, Remote Method Invocation (RMI), distributed JavaBeans, and Java Database Connect (JDBC).

The WebLogic Server's Administration Server is populated with JavaBean-like objects Sun Microsystem's Java Management Extension (JMX) standard. These objects provide management access to domain resources.

The Administration Server contains both configuration MBeans and run-time MBeans. Configuration MBeans provide both SET (write) and GET (read) access to configuration attributes. Run-time MBeans provide a snapshot of information about domain resources, such as current HTTP sessions or the load on a JDBC session pool. When a particular resource in the

domain (such as a Jolt connection pool) is instantiated, an MBean is created to collect information about that resource.

Note: For more information about configuration and run-time MBeans, refer to the *Oracle WebLogic Server Administration Guide*.

The WebLogic Server is configured to initialize the session pools at startup through the `config.xml` file. A special startup class, `PoolManagerStartUp`, is invoked by the WebLogic Server with a number of parameters. This class functions as follows:

- Creates a session pool manager if one does not already exist
- Creates a session pool according to the given parameters
- Adds the new session pool to the pool manager

Note: Start the Jolt servers before attempting to create a session pool; otherwise the startup classes will fail, and they will not attempt to commit again.

The number of session pools created depends on the number of `JoltConnectionPools` that are configured in the `config.xml` file.

Connecting to a WebLogic Server from a Client Browser

In addition to its other Java services, the WebLogic Server is a fully functional HTTP server that supports Java HTTP servlets. In general, each servlet must be registered with a virtual name in the `config.xml` file.

A servlet may be invoked directly, that is, may actually present HTML to the browser, or may be invoked indirectly from an HTML form when the user submits the form. When the WebLogic Server receives a request containing the registered virtual name of a servlet, it invokes the appropriate servlet's `service()` method. For more information on HTTP servlets, refer to the *Programming WebLogic HTTP Servlets* guide.

The HTTP servlet's `service()` method (which invokes either the servlet's `doPost()` or `doGet()` method, depending on the context) is invoked and passes an `HttpServletRequest` object containing the HTTP data sent from the browser. In the example packages described in "Using the Example Packages" on page 1-7, the client's query data is used in a transaction call to Tuxedo, and the response is built into the new HTML page.

How a Servlet Connects to Tuxedo

A servlet obtains a reference to the session pool manager that was created and initialized by the WebLogic Server when it started. The pool manager is used to retrieve the session pool that was configured in the `config.xml` file. This session pool references the appropriate Jolt Server in a Tuxedo domain. A servlet uses the session pool to invoke a specific Tuxedo service.

Tuxedo services are described and exported (declared accessible) on the Jolt Server in the Jolt Repository. In the Jolt Repository, the service's expected input and output parameter types are declared. A servlet must supply the expected input parameters; Oracle Jolt for Oracle WebLogic Server uses specialized `ServletSessionPool` objects that can accept their input directly from an `HttpServletRequest` object. The output is returned in a `ServletResult` object.

What Happens if the Request Fails

The session pool distributes the requests equally among the sessions in the pool. It selects the least busy session to call the Tuxedo service. If the selected session is terminated before the Tuxedo service is called, the session pool redirects the service call to a different session, then establishes a new session to replace the disconnected one. The session pool uses a round-robin algorithm to select and establish a connection to a primary Jolt Server. If no primary Jolt Servers responds, the session pool connects to a failover server.

If no sessions are available from a session pool, or the session pool is suspended, then a `SessionPoolException` is thrown.

Multiple requests can be grouped into a single transaction. When a transaction fails, a `TransactionException` is thrown. This exception should be caught by the servlet and handled appropriately. (Usually, the servlet performs a rollback.)

Responding to the Client Browser

Provided the service call was successful, the following events occur:

- The desired results are extracted from the `ServletResult` object.
- The results are processed by the servlet and incorporated into an HTML page for presentation to the user's browser. The HTML page can be built in one of two ways:
  - With WebLogic's easy-to-use Java Server Pages (JSP) service that lets you embed Java in a standard HTML page.
  - Using a more sophisticated programmatic approach with WebLogic `htmlKona`.

- The WebLogic Server returns the HTML page to the client via the `HttpServletResponse` object.

Disconnecting from the Jolt Server

The WebLogic Server is also configured to shut down the existing session pool connections to Tuxedo through the `config.xml` file.

Register the class `PoolManagerShutDown` so that the Jolt session pool is cleaned up properly when the WebLogic Server shuts down. `PoolManagerShutDown` does not require an attribute in the `config.xml` file.

Using the Example Packages

Two example packages are included with Oracle Jolt for Oracle WebLogic Server. These packages are described in Appendix B, "Simple Servlet Example," and Appendix C, "Servlet with Enterprise JavaBean Example." They demonstrate how Jolt is used in WebLogic servlets to access Tuxedo services. You can build, run, and inspect these examples to help you decide how to use WebLogic to extend Tuxedo services to the Internet.

- Simple Servlet Example

  A FORM-based HTML front end that submits a string to an HTTP servlet. The servlet in turn sends this string to a Tuxedo service. The returned data is compiled into a dynamically-generated HTML file, and sent back to the client browser.

- Servlet with Enterprise JavaBean Example

  The Enterprise JavaBean (EJBean) example package contains the classes and other files necessary to set up and run an EJBean stateful session to a Tuxedo Server that is using Jolt.

CHAPTER 2

Configuring Jolt for WebLogic Server

Configuring a Jolt Session Pool connection between Tuxedo and WebLogic Server requires two procedures:

- Configuring Jolt for Tuxedo
- Configuring Jolt for WebLogic Server

Configuring Jolt for Tuxedo

Refer to the *Using Oracle Jolt* for instructions on setting up a Jolt Service Listener (JSL) within Tuxedo. In *Using Oracle Jolt*, it is assumed that JSL services have already been configured within the Tuxedo domain. The guide only describes how to establish a session pool connection to these services from WebLogic Server.

Configuring Jolt for WebLogic Server

This section describes how to set up an Oracle Jolt connection pool between the WebLogic Server and the JSL in the Tuxedo domain. Your WebLogic Server must have access to the host running the JSL.

Jolt Startup Class and Connection Pool

You must instruct WebLogic Server to invoke the `PoolManagerStartUp` class whenever the WebLogic Server is started or restarted. This invocation establishes the pool connection to Tuxedo from the `config.xml` file, as shown in the following example.

Note: For WebLogic Server 6.0 or later, Jolt startup classes and connection pool attributes are configured via the configuration MBeans in the Administration Console. For more information about configuration and run-time MBeans, refer to the *Oracle WebLogic Server Administration Guide*.

```
<StartupClass
  ClassName="bea.jolt.pool.servlet.weblogic.PoolManagerStartUp"
  FailureIsFatal="false"
  Name="MyStartup Class"
  Targets="myserver"
/>
<JoltConnectionPool
   ApplicationPassword="tuxedo"
   MaximumPoolSize="5"
   MinimumPoolSize="3"
   Name="MyJolt Connection Pool"
   PrimaryAddresses="//TUXSERVER:6309"
   RecvTimeout="300"
   SecurityContextEnabled="true"
   Targets="myserver"
   UserName="joltuser"
   UserPassword="jolttest"
   UserRole="clt"
/>
```

The startup class in the preceeding example instructs WebLogic Server to invoke the `PoolManagerStartUp` class when the WebLogic Server starts. The `JoltConnectionPool` specifies initialization arguments that are passed to the `PoolManagerStartUp` class. If you do not want the `SessionPool` to try to reestablish the connection in case any of the JSL is forced to shutdown, set the JVM property `jolt.sessionPoolKeepAlive=false` when starting up the Weblogic Server.

Jolt Connection Pool Attributes

The Jolt connection pool attributes are defined as follows:

`Application Password`	(Optional) Tuxedo application password. This is required only if the Tuxedo authentication level is `USER_AUTH` or `APP_PW`.
`MininumPoolSize`	(Required) Specifies the initial session pool size when the session pool is created.
`MaximumPoolSize`	(Required) Specifies the maximum session pool size. Each session within a pool can handle up to 50 outstanding requests at any one time.
`Name`	(Optional) Defines a name for this session pool that should be unique from the names of other session pools. This is an optional argument, but it is recommended that you use it to avoid ambiguity. The `SessionPoolManager` allows only one session pool to remain unnamed. You can access this unnamed session pool from your application by supplying `null` in place of the *`poolname`* string argument to the `getSessionPool()` method. **Note:** We *strongly* recommend that you name every session pool.
`PrimaryAdresses`	(Required) Defines a list of the addresses of the primary Jolt Server Listeners (JSLs) on the Tuxedo system. These are defined in the format: *`//hostname:port`* where *`hostname`* is the name of the server where the JSL is running, and *`port`* is the port on which the JSL is configured to listen for requests. You can specify multiple addresses in a semicolon-separated (;) list. **Note:** You must specify at least one primary JSL *`hostname:port`* address.
`Failover Addresses`	(Optional) You can specify a list of failover Jolt Server Listeners in the same format used for *`appaddrlist`* above. Jolt attempts to use these failover JSL(s) if the primary JSLs listed above fail. These JSLs need not reside on the same host as the primary JSLs.
`RecvTimeout`	(Required) Specifies the amount of time the client should wait to receive a response before timing out.

SecurityContext Enabled	(Optional) Enables or disables the security context for this connection pool. This option should be enabled if you want to implement authentication propagation between WebLogic Server and Jolt. If identity propagation is desired, then the Jolt Service Handler (JSH) must be started with the -a option. If this option is not set, but SecurityContext is enabled, the JSH will not accept this request. If the SecurityContext attribute is enabled, then the Jolt client will pass the username of the caller to the JSH. If the JSH gets a message with the caller's identity, it calls impersonate_user() to get the appkey for the user. JSH caches the appkey, so the next time the caller makes a request, the appkey is retrieved from the cache and the request is forwarded to the service. A cache is maintained by each JSH, which means that there will be a cache maintained for all the session pools connected to the same JSH.
Targets	(Required) Specifies the target servers for the connection pool.
UserName	(Optional) Tuxedo user name. This is required only if the Tuxedo authentication level is USER_AUTH.
UserPassword	(Optional) Tuxedo user password. This is required only if the Tuxedo authentication level is USER_AUTH.
UserRole	(Optional) Tuxedo user role. This is required only if the Tuxedo authentication level is USER_AUTH or APP_PW.

It is recommended that you configure one Jolt session pool for each application running on the WebLogic Server.

Jolt Shutdown Class

To configure WebLogic Server to disconnect the Jolt session pools from Tuxedo when it shuts down, add the following lines to the WebLogic Server config.xml file:

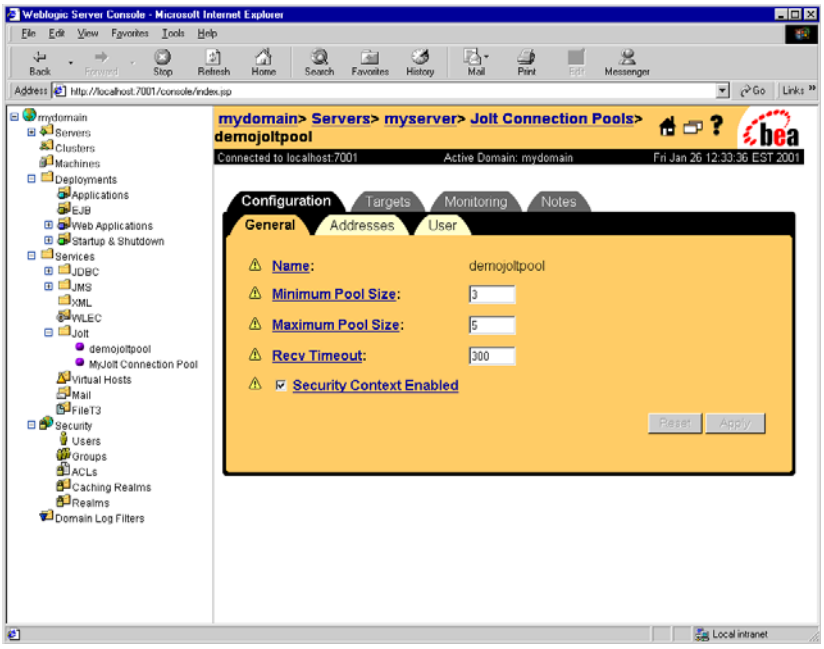
```
<ShutdownClass
  ClassName="bea.jolt.pool.servlet.weblogic.PoolManager ShutDown."
/>
```

The shutdown class instructs WebLogic Server to invoke the PoolManagerShutDown class when the WebLogic Server shuts down.

Displaying Jolt in the WebLogic Administration Console

If you are connecting to a WebLogic Server that has Jolt correctly installed and configured, when you start the Administration Console you will see a configuration MBean for the Jolt connection pool displayed in the Administration Console, as shown in Figure 2-1.

Figure 2-1 WebLogic Server Console with Jolt Connection Pool



For each Jolt connection pool there is an individual `MBean` that displays the pool name, maximum connections, pool state, and statistics about the connection status.

Note: For more information about MBeans, refer to the *Oracle WebLogic Server Administration Guide*.

Resetting the Jolt Connection Pool

You can reset the Jolt connection pool without having to restart WebLogic Server. The `resetConnectionPool()` method calls the `SessionPoolManager.stopSessionPool()`

method to shut down all the connections in the pool. It then calls the `SessionPoolManager.createSessionPool()` method to restart the connection pool.

Command-line Method

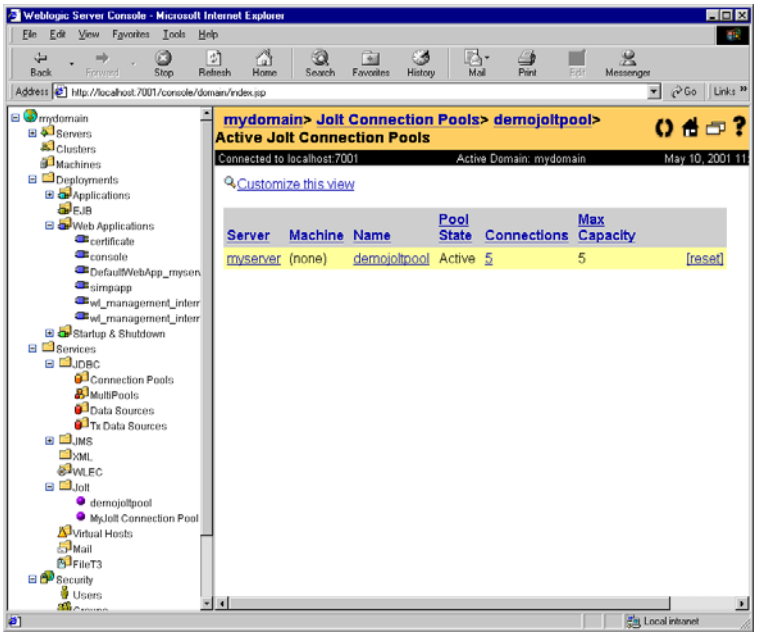
The `resetConnectionPool` method can be invoked from the Administration Console command-line interface by using the following command:

```
java weblogic.Admin -url t3://localhost:7001 -username system -password
gumby1234 -invoke -mbean
mydomain:Name=myserver.jolt.demojoltpool,Type=JoltConnectionPoolRuntime,Lo
cation=myserver -method resetConnectionPool
```

Administration Console Method

The Jolt connection pool can also be reset from the GUI console by using the following method:

1. Under Services in the left frame, click the Jolt service folder.
2. Click the configured Jolt Connection Pool that you would like to monitor.
3. In the right frame, click the Monitoring tab, and then click the Monitor all Active Pools link. The console lists all the connection pools that have been configured.
4. Click the Monitor all instances of... link next to the Jolt connection pool that you would like to monitor. The console displays the Active Jolt Connection Pool.



5. Click the Reset Connection Pool icon at the end of the row to reset the connection pool.

CHAPTER 3

Implementing Jolt for WebLogic

Setting up Jolt to connect to Tuxedo from your WebLogic application or servlet requires the following steps:

- Importing Packages
- Configuring a Session Pool
- Using a Servlet Session Pool
- Accessing a Tuxedo Service Through Jolt
- Converting Java Data Types to Tuxedo Data Types
- Receiving Results from a Service
- Using a Transaction

See page B-1 for a simple example that establishes a connection and accesses a Tuxedo service from an HTTP servlet.

Importing Packages

The Jolt Java class packages are automatically installed when you install Jolt for WebLogic Server. To use Oracle Jolt for Oracle WebLogic Server, import the following class packages that were installed with Jolt into your servlet:

```
bea.jolt.pool.*
```

```
bea.jolt.pool.servlet.*
```

There are other classes you must import into any servlet; for more information on writing Java servlets, read the *Programming WebLogic HTTP Servlets* guide.

Configuring a Session Pool

You can access session pools through the `SessionPoolManager` class. WebLogic Server uses a variation of the session pool called a servlet session pool. The servlet session pool provides extra functionality that is convenient for use inside an HTTP servlet.

When you configure a servlet session pool through the WebLogic Administration Console, the following information is added to the `config.xml` configuration file:

```
<StartupClass
  ClassName="bea.jolt.pool.servlet.weblogic.PoolManagerStartUp"
  FailureIsFatal="false"
  Name="MyStartup Class"
  Targets="myserver"
/>
<JoltConnectionPool
   ApplicationPassword="tuxedo"
   MaximumPoolSize="5"
   MinimumPoolSize="3"
   Name="MyJolt Connection Pool"
   PrimaryAddresses="//TUXSERVER:6309"
   RecvTimeout="300"
   SecurityContextEnabled="true"
   Targets="myserver"
   UserName="joltuser"
   UserPassword="jolttest"
   UserRole="clt"
/>
```

When WebLogic is started (or restarted), it invokes the `PoolManagerStartUp` class and its associated `startupArgs`. On the first invocation, the `PoolManagerStartUp` class creates a `ServletSessionPoolManager` object, which contains every `ServletSessionPool` configured in the `config.xml` configuration file.

Subsequent calls add another `ServletSessionPool` to the same `ServletSessionPoolManager`. You must add an entry for each session pool, using a unique virtual name binding for each, as shown in the preceding example. The WebLogic Server creates a new `ServletSessionPool` as defined in the `config.xml` file.

For additional information about property settings and a list of definitions, see "Jolt Startup Class and Connection Pool" on page 2-1.

Accessing a Servlet Session Pool

Once a WebLogic Server is configured to set up a Jolt session pool on startup, you can access and use the Jolt session pool from your Java application or servlet. As described earlier, in the WebLogic Server all `ServletSessionPool` objects are managed by the same `ServletSessionPoolManager`.

```
ServletSessionPoolManager poolMgr = (ServletSessionPoolManager)
  SessionPoolManager.poolmanager;
```

The WebLogic Server uses a `ServletSessionPoolManager` class that is derived from `SessionPoolManager`. The `ServletSessionPoolManager` manages `ServletSessionPool` objects, which offer additional HTTP servlet methods.

`SessionPoolManager` provides several methods for managing the administration of a session pool. In the following example, the `SessionPoolManager` is used to retrieve the `SessionPool` that has been named *`joltpoolname`*:

```
SessionPool sPool = poolMgr.getSessionPool("joltpoolname");
```

However, because the WebLogic Server uses the subclass `ServletSessionPoolManager`, the above example actually returns a `ServletSessionPool` object in the guise of a `SessionPool`.

You must cast the `SessionPool` to a `ServletSessionPool`, as in the following code example:

```
ServletSessionPool ssPool =
  (ServletSessionPool) poolMgr.getSessionPool("joltpoolname");
```

Because WebLogic Server creates and configures the `ServletSessionPoolManager`, it is likely that this is the only method you will use. Other `SessionPoolManager` methods allow you to create, suspend, stop, or shut down individual or all the session pools it manages. We recommend that you leave these administrative operations to the WebLogic Server by configuring and managing your session pools using the WebLogic `config.xml` configuration file.

Using a Servlet Session Pool

The reference to the named `ServletSessionPool` from the pool manager represents a pool of sessions (or connections) to the Jolt Server in Tuxedo. The size of this pool and the Tuxedo

system to which it connects are abstracted from the application code and are defined in the WebLogic `config.xml` configuration file. When you initiate a request, the `SessionPool` uses the least-busy connection available.

Calling a Tuxedo Service

A Jolt request usually consists of a single call to a Tuxedo service using the `call()` method of the `SessionPool`. You supply the name of the Tuxedo service and a set of parameters to the `call()` method, and it returns a set of results from the Tuxedo service. It is possible to make multiple calls within a single transaction, which allows a servlet to comply with transactional demands of a Tuxedo application or preserve integrity between databases. This transaction is described in more detail in “Using a Transaction” on page 3-9.

Sending a ServletDataSet

The `ServletSessionPool` provides overloaded `call()` methods for use inside an HTTP servlet. These methods accept their input parameters in terms of an `HttpServletRequest` object, and therefore can conveniently be passed the same `HttpServletRequest` object that was passed into your HTTP servlet's `doPost()` or `doGet()` methods. However, in this instance, you must ensure that the names of the HTTP posted *`name=value`* pairs correspond to those expected by the Tuxedo service. The ordering is not important, because the data is ultimately converted into a Java Hashtable. Other non-related data in the `HttpServletRequest` will not disrupt the Tuxedo service.

A Tuxedo service is invoked from within an HTTP servlet with the following method:

```
ssPool.call("serviceName", request);
```

where `ssPool` is a reference to a `ServletSessionPool`, *`"serviceName"`* is the name of the Tuxedo service you wish to call, and the *`request`* argument is the `HttpServletRequest` object associated with the servlet.

The `ServletSessionPool.call()` method internally converts the `HttpServletRequest` into a `ServletDataSet`, which can be submitted to a regular `SessionPool`.

Adding Parameters to the Dataset

You may wish to add extra data to the parameter set before calling the Tuxedo service. For example, you may need to add a parameter representing the date and time of the request. You would not expect to receive this parameter from the `FORM` data in the `HttpServletRequest`.

Instead, add it in the servlet, then submit the augmented data set to the Tuxedo service. The following example illustrates this procedure:

```
// Create a new dataset

ServletDataSet dataset = new ServletDataSet();

// Import the HttpServletRequest into the dataset.

dataset.importRequest(request);

// Insert an extra parameter into the dataset.

dataset.setValue("REQUEST_TIME", (new Date()).toString());

// Send the dataset to the named service.

ssPool.call("service_name", dataset, null);
```

This code example demonstrates the manual conversion of the `HttpServletRequest` object into a `ServletDataSet` object. In this new format you can add extra parameters using the `setValue()` method. The new value is associated with a key, represented by a string. Next, the `call()` method that is inherited from the `SessionPool` is invoked. This method accepts the `ServletDataSet` class, but requires an extra argument for use with transactions. Supply `null` for this last parameter, indicating that you are not using a transaction to group multiple session calls. See"Using a Transaction" on page 3-8 for more details.

Accessing a Tuxedo Service Through Jolt

To access an existing Tuxedo service through Jolt, you must define and export the service in the Jolt Repository. For details, refer to the *Using Oracle Jolt* sections "Using the Jolt Repository Editor" and "Bulk Loading Oracle Jolt Services." The Jolt service definition defines the parameters that are expected by the Tuxedo application service.

Converting Java Data Types to Tuxedo Data Types

The following table is a mapping between Java types and Tuxedo parameter types required by a Tuxedo service. Use the appropriate Java types for the value of the `DataSet` or `ServletDataSet`. If you specify any parameter as a Java `String`, it is translated automatically to the appropriate type according to the service definition in the Jolt Repository.

This feature is also used to convert all data inside an `HttpServletRequest` object, because all parameters associated with the request are represented in string format. Otherwise, use the type

specified in the table below. Providing the correct data type may improve efficiency because no lookup is required to convert from a string.

Oracle Tuxedo Type	Java Type
char	Byte
short	Short
long	Integer
float	Float
double	Double
char*	String
CARRAY	byte[ ]
XML	byte[ ]

A Tuxedo `CARRAY` is specified in a Java string by describing each byte value as a two-digit hexadecimal number. You specify multiple bytes by concatenating these hexadecimal digit-pairs together. For example, the string `"FF0A20"` would represent the Tuxedo type `CARRAY { 255, 10, 32 }`.

Receiving Results from a Service

The `ServletSessionPool.call()` method returns a `ServletResult` object that contains the results from the Tuxedo service. If the service call fails, an exception is thrown. You should always attempt to catch exceptions and handle them appropriately. Refer to "Appendix A, Oracle Jolt Exceptions" in *Using Oracle Jolt* for details about the possible exceptions that can occur.

The following example retrieves a `ServletResult` object using the `ServletSessionPool.call()` method in an HTTP servlet:

```
ServletResult sResult = ssPool.call("service_name", request);
```

where `ssPool` is a `ServletSessionPool`, and *`request`* is an `HttpServletRequest`.

The `ServletSessionPool.call()` method returns a `Result` object that you must cast as a `ServletResult` object. The `ServletResult` object provides extra methods for retrieving data as Java Strings.

Provided the call was successful, the individual parameters can be retrieved from the `Result` or `ServletResult` object using various forms of the `getValue()` method.

Using the Result.getValue() Method

The data is retrieved from a `ServletResult` by providing a key that corresponds to the parameter names of the Tuxedo service, as defined in the Jolt Repository. You supply the key to the appropriate `getValue()` method, which returns the corresponding value object.

The `Result.getValue()` method also expects a default value object; this is returned if the key lookup fails. It is your responsibility to cast the returned object to the appropriate type, as defined by the Tuxedo service. For example, this line of code:

```
Integer answer = (Integer) resultSet.getValue("Age", null);
```

sets the integer `answer` to the returned value in the `ServletResult` identified by the key `"Age"`, or returns `null` if this key does not exist in the `ServletResult`. Refer to the table in "Converting Java Data Types to Tuxedo Data Types" on page 3-5 for the Java equivalents of the Tuxedo types.

It is possible to have an array of values associated with a key. In this case, the simple `getValue()` method returns the first element of an array in this instance. Use this method signature in that case:

```
public Object getValue(String name, int index, Object defVal)
```

to reference a particular indexed element in an array value.

Using the ServletResult.getStringValue() Method

`ServletResult` extends `Result`, and provides the additional methods:

```
public String getStringValue(String name,
                             int index,
                             String defVal)


public String getStringValue(String name,
                             String defVal)
```

These methods behave like the `getValue()` methods of the `Result` class, except that they always return a Java string equivalent of the value object expected. The `CARRAY` is converted into

a string of two digit hexadecimal byte values as described in “Converting Java Data Types to Tuxedo Data Types” on page 3-5.

Using a Transaction

You can use a transaction object to group multiple service calls into an atomic action, maintaining data integrity within your application logic. You obtain a transaction from a session pool with the method:

```
Transaction trans = ssPool.startTransaction(timeout);
```

where the transaction object *trans* holds the reference to the transaction, *ssPool* is the SessionPool or ServletSessionPool object, and the `timeout` argument for the transaction is specified in seconds.

Once a transaction obtains a session, that session cannot be used by other transactions until the transaction is committed, aborted, or times out. The session may, however, still be used by single requests that are not part of a transaction. If a transaction fails to obtain a session from the pool, this method throws a `bea.jolt.pool.TransactionException`. If the session pool is suspended, the method throws a `bea.jolt.pool.SessionPoolException`.

Each time your application uses the `call()` method, you should supply the transaction object as the last parameter. For example:

```
ssPool.call("svcName", request, trans);
```

You can make multiple calls in the same transaction. The calls will not complete until you either commit or roll back the transaction using the methods of the transaction object. The `trans.commit()` method completes the transaction. This method returns 0 if the commit was successful, or throws a `TransactionException` if the transaction failed to commit.

If you need to abort the transaction, use the `Transaction.rollback()` method. This method attempts to abort the transaction. It returns 0 if successful; otherwise it throws a `TransactionException`.

Handling Exceptions

Errors or failures that may occur when Jolt initiates a Tuxedo service call are reported to your application through Java exceptions. Always enclose the `call()` method within a `try / catch` block and attempt to deal with any exceptions appropriately. The `call()` method can throw any of the following exceptions for the following reasons:

- `bea.jolt.pool.ApplicationException`

Thrown when an error occurs in the logic of the Tuxedo service. For example, a client illegally attempts to use a withdrawal service to withdraw more money from an account than the current balance. The `ApplicationException` is thrown when the Tuxedo service returns a `TPESVCFAIL`. Application-specific information about the error can be included in the `Result` object that was returned from the service invocation. You can access the `Result` object through the `ApplicationException.getResult()` method.

Be sure to use the full package name of the `bea.jolt.pool.ApplicationException`, because Jolt defines another exception whose full package name is `bea.jolt.ApplicationException`.

- `bea.jolt.JoltException`

  A `JoltException` is the super class of all the following exceptions. These exceptions all signify that a system error has occurred that is not part of the application logic. `JoltException` is documented in Appendix A, "Oracle Jolt Exceptions," in the *Using Oracle Jolt*.

- `bea.jolt.pool.SessionPoolException`

  Thrown when an error occurs in the Jolt SessionPool. For example, this may occur if all sessions are busy, or if the session pool is suspended.

- `bea.jolt.ServiceException`

  Thrown when an error occurs related to invoking the Tuxedo service that contains the application. For example, a service timeout, or a non-existent service is called.

- `bea.jolt.TransactionException`

  Thrown when a transaction cannot be either started, committed, or aborted.

APPENDIX A

Class Hierarchy

Oracle Jolt Class Hierarchy for the Oracle WebLogic Server API

The following listing shows the class hierarchy for the Oracle Jolt for Oracle WebLogic Server API. Refer to the appropriate Java documentation for details about each class and method.

```
Package-bea.jolt.pool
Package-bea.jolt.pool.servlet
Package-bea.jolt.pool.servlet.weblogic

Class java.lang.Object
    Class bea.jolt.pool.Connection
    Class java.util.Dictionary
        Class java.util.Hashtable
        (implements java.lang.Cloneable, java.io.Serializable)
          Class bea.jolt.pool.DataSet
            Class bea.jolt.pool.Result
              Class bea.jolt.pool.servlet.ServletResult
             Class bea.jolt.pool.servlet.ServletDataSet
           Class bea.jolt.pool.SessionPoolManager
             Class bea.jolt.pool.servlet.ServletSessionPoolManager
    Class bea.jolt.pool.Factory
        Class bea.jolt.pool.SessionPool
          Class bea.jolt.pool.servlet.ServletSessionPool
    Class java.lang.Throwable
```

```
 (implements java.io.Serializable)
    Class java.lang.Exception
      Class java.lang.RuntimeException
        Class bea.jolt.pool.ApplicationException
Class bea.jolt.pool.Transaction
Class bea.jolt.pool.UserInfo
```

APPENDIX B

Simple Servlet Example

This example demonstrates how to use Oracle Jolt to connect to Oracle Tuxedo from a WebLogic servlet. It uses the WebLogic Server to deliver an HTML FORM front end in a standard Web browser.

Text entered by a user into the FORM is sent back to the WebLogic Server via the HTTP POST method that is serviced by a registered WebLogic HTTP Servlet, which calls a Tuxedo service using Oracle Jolt. The text received by the servlet is sent to a Tuxedo service, where it is transposed to uppercase before being returned to the servlet. The form is compiled into a dynamically-generated HTML page by the servlet, then sent back to the Web browser, where the uppercase version of the original text is displayed.

This topic includes the following sections:

- Example Components and Prerequisites
- Using the Example

Example Components and Prerequisites

There are two parts to the `simpapp` example for Jolt for WebLogic Server:

- The HTTP servlet that is shipped with the examples that are installed in the `samples` directory where Oracle Tuxedo is installed.
- The Tuxedo service application that is shipped with the Tuxedo examples that are installed with Oracle Tuxedo. The Tuxedo `simpapp` server contains the ***`TOUPPER`*** service, which converts a given string to uppercase.

The source code for the Jolt servlet `simpapp` example is located in the `/samples/jolt/wls/servlet/` directory in the Tuxedo distribution.

The `simpapp` sample directory contains the following files:

File Name	Description
`SimpAppServlet.java`	Sample source code that issues a call to Tuxedo and returns an HTML page with the results
`simpapp.html`	HTML form for user input
`simpapp.rep`	REP file for repository bulk loading
`web.xml`	Configuration XML file for Web applications

A complete listing of the Tuxedo server-side source code of the `simpapp` application service is located in `$TUXDIR/samples/atmi/simpapp` on UNIX systems and in `%TUXDIR%\samples\atmi\simpapp` on Windows 2003 systems (where *`TUXDIR`* is the Tuxedo home directory).

To run this example, you should be familiar with:

- The Oracle Tuxedo architecture and the `simpapp` application
- Oracle Jolt
- HTML
- Java language and servlet API
- WebLogic Server HTTP servlets

Using the Example

The `simpapp` example is easy to follow. Just launch the `simpapp.html` page from the WebLogic Server. The `simpapp.html` page loads an HTML form which contains a text field for entering the string. Type in a string and click the Post button to submit the string as a post request. The `SimpAppServlet` formats the string you typed for use with the Jolt for WebLogic class libraries, and then dispatches the request to the Tuxedo `TOUPPER` service, which transposes the string to uppercase and returns it for display in the browser.

Configuring the `simpapp` servlet example requires the following steps:

- Step 1. Perform Preparatory Steps
- Step 2. Start the WebLogic Server
- Step 3. Configure the Servlet in WebLogic Server
- Step 4. Stop and Restart the WebLogic Server
- Step 5. Compile the Servlet
- Step 6. Display the simpapp.html Form
- Step 7. Post the FORM Data from the Browser
- Step 8. Process the Request
- Step 9. Return the Results to the Client

Step 1. Perform Preparatory Steps

1. Check that you have a supported browser installed on your client machine:
   - Netscape Communicator 4.7 or later
   - Internet Explorer 5.0 or later
2. The client machine must have a network connection to the WebLogic Server that is used to connect to the Tuxedo environment.
3. Configure and boot Tuxedo and the `simpapp` example.
4. Follow the directions in the Tuxedo user documentation to bring up the server-side simpapp application. Make sure the `TOUPPER` service is available.
5. Set up the Jolt Server. Refer to the *Using Oracle Jolt* for information about how to configure a Jolt Server.
   - Note the hostname and port number associated with your Jolt Server Listener (JSL).
   - Use the Jolt Repository BulkLoader file to ensure that the `TOUPPER` service is defined in the Jolt Repository.

     The `simpapp` example directory has a `simpapp.rep` file that contains the `TOUPPER` service definition. Your system administrator should use the Jolt Repository BulkLoader to add this service definition to the existing Jolt Repository on the Tuxedo server. The Jolt Repository BulkLoader package is supplied with the Jolt distribution for Tuxedo. Refer to *Using Oracle Jolt* for details on how to install this.

On the Tuxedo server, the following code example uses the Jolt BulkLoader to add the `TOUPPER` service definition:

```
$ java bea.jolt.admin.jbld //host:port simpapp.rep
```

where *`host`* and *`port`* are the hostname and port number of your Jolt Server Listener (JSL), and the `simpapp.rep` is the BulkLoader file provided by Oracle Jolt, located in one of the following locations:

`$TUXDIR/samples/jolt/wls/servlet/` on UNIX

`%TUXDIR%\samples\jolt\wls\servlet\` on Windows 2003

6. Confirm that you have properly set up your `CLASSPATH` during installation. The WebLogic Server classes library contains the three `.jar` files that you will need to run this example:
   - `jolt.jar`
   - `joltjse.jar`
   - `joltwls.jar`.

Step 2. Start the WebLogic Server

If you are using a Windows 2003 system, you can start the WebLogic Server from the Start menu. Otherwise, use the `startWebLogic` script on the command line, in the root directory of the WebLogic Server distribution.

For more information on starting the WebLogic Server, see "Starting and Stopping the WebLogic Server" in the *Oracle WebLogic Server Administration Guide*.

Step 3. Configure the Servlet in WebLogic Server

Configuration of the Jolt connection pool and startup class MBeans for WebLogic Server 6.0 or later is done through Administration Console.

1. Copy the `simpapp.html` page into your WebLogic document root directory.

   By default, this is the `\config\mydomain\applications\simpapp` directory in your WebLogic Server distribution. The HTTP server built into WebLogic looks in this directory for HTML pages and other MIME types.

2. Start the WebLogic Server Administration Console by typing the following address in your browser:

   ```
   http://hostname:listenport#/console
   ```

3. Open the Services folder in the left frame of the console, and then click the Jolt folder. The Jolt Connection Pools table displays in the right frame showing all the Jolt connection pools defined in the domain.
4. Click the Create a New Jolt Connection Pool link. A tabbed dialog box displays in the right frame for configuring a new connection pool.
5. On the General tab, complete the following information:
   a. Enter values in the Name, Minimum Pool Size, Maximum Pool Size, and the Recv Timeout attribute fields.
   b. Select the Security Context Enabled check box to enable security context (to propagate the security information from the WebLogic Server environment to the Tuxedo environment).
   c. Click Create to create a connection pool instance with the name that you specified in the Name field. The new instance is added under the Jolt node in the left frame.
6. Click the Config-Addresses and the Config-User tabs individually to change the attribute fields or accept the default values as assigned, and then click Apply to save your changes.
7. Click the Targets tab and select an available server where you want the Jolt connection pool started.
8. Under the Deployments folder in the left frame, click the Startup & Shutdown folder. The Startup and Shutdown table displays in the right frame showing all the startup classes defined for your domain.
9. Click the Create a New Startup Class link. In the tabbed dialog box that displays in the right frame, configure a new startup class, as follows.
   a. Enter values in the Name, Class Name, and Arguments attribute fields.
   b. Select the Abort Startup on Failure check box to prevent starting the WebLogic Server whenever a failure occurs.
   c. For the Class Name, enter the following name:
      `bea.jolt.pool.servlet.weblogic.PoolManagerStartUp`
      There are no arguments for this startup class.
   d. Click Create to create a startup-class instance with the name that you specified in the Name field. The new instance is added under the Startup & Shutdown folder in the left frame.

10. Register the simpapp servlet as a Web application, as follows:

    a. Open the Deployments folder in the left frame of the console, and then click the Web Applications icon.

    b. On the Install or Update an Application dialog box, click the Install a New Web Application link.

    c. For Step 1, either accept the default a destination directory for the simpapp servlet or select a different one.

    d. For Step 2, enter the path to the simpapp servlet (or use the Browse feature), and then click the Upload button.

       The simpapp servlet is registered as a Web application in WebLogic and appears as an icon under the Deployments\Web Applications folder.

Step 4. Stop and Restart the WebLogic Server

In order to start the Jolt session pool, you must shut down the WebLogic Server, and then restart it. For more information on restarting the WebLogic Server, see "Starting and Stopping the WebLogic Server" in the *Oracle WebLogic Server Administration Guide*.

Step 5. Compile the Servlet

After restarting the WebLogic Server, compile the SimpAppServlet file, as follows:

1. Under your WebLogic \config\mydomain\applications\simpapp document root directory, create a new WEB-INF directory.

2. Copy the web.xml file from the Tuxedo installation directory \samples\jolt\wls\servlet\ into the new WEB-INF directory.

3. Compile the SimpAppServlet.java file, as follows:

```
javac -d %WL.HOME%\config\mydomain\applications\simpapp\WEB-INF\classes
SimpAppServlet.java
```

   This step also copies the necessary java classes into a WEB-INF\classes directory.

Step 6. Display the simpapp.html Form

1. Open your browser.

2. Enter the URL for the simpapp.html file. For example, the default URL is:

`http://localhost:port/simpapp/simpapp.html`

where *`localhost`* is the host name of the WebLogic Server, and *`port`* is the port at which the WebLogic Server is listening for login requests.

A page similar to the one shown in Figure B-1 is displayed:

Figure B-1 simpapp.html Example

If you have problems displaying the form, be sure that the `simpapp.html` file is in the WebLogic document root.

Step 7. Post the FORM Data from the Browser

Enter some text into the text field on the HTML page and submit it by clicking the POST button. Along with the text you entered, other parameters are submitted to the `simpapp` servlet class running in WebLogic Server.

The following is the relevant section from the `simpapp.html` file that describes the HTML form:

```
<form name="simpapp" action="simpapp" method="post">
<input type="hidden" name="SVCNAME" value="TOUPPER">

<table bgcolor=#dddddd border=1>
<tr>
<td>Type some text here and click the Post button:
<input type="text" name="string">
</td></tr>

<tr>
<td align=center><input type="submit" value="Post!">
</td></tr>
```

```
</table>
</form>
```

This HTML form specifies two input fields: the text you enter and a hidden field. In this example, the value of the hidden field actually specifies the name of the Tuxedo service to be invoked. Although putting the name of the Tuxedo service within the HTML page is flexible and efficient, it is not recommended for production use for security reasons. In this HTML page, you can submit an HTTP request specifying a different service name as the hidden field.

Note: Tuxedo service names are case-sensitive.

When the WebLogic Server receives the HTTP form request, it invokes the `doPost()` method of the `simpapp` servlet and passes the form data into an `HttpServletRequest`.

Step 8. Process the Request

Before the first request to the `simpapp` servlet, WebLogic initializes the servlet by calling its `init()` method. The Jolt session pool is established in the following manner:

```
ServletSessionPoolManager b_mgr =
 (ServletSessionPoolManager).SessionPoolManager.poolmanager;
```

Next, the servlet's `doPost()` method is executed. This method contains the code to get a connection from the `simpapp` session pool that was created during the startup of the WebLogic Server. The following code snippet shows the code that is used to retrieve the `simpapp` session pool.

```
// Get the "simpapp" session pool
ServletSessionPool session =
  (ServletSessionPool) b_mgr.getSessionPool("simpapp");
```

The Tuxedo service that will be called is identified in a hidden field, which is retrievable from the request object. Retrieve the service name parameter as follows:

```
String svcnm[] = req.getParameterValues("SVCNAME");
```

You retrieve the value of the `SVCNAME` field in a string array that contains a single value; use only the first element of the array. The value set for the `SVCNAME` hidden field in the form is `TOUPPER`. This is the name of the Tuxedo service that the servlet invokes, which is passed to the `call()` method as follows:

```
// Invoke a service and get the result.
result = session.call(svcnm[0], req);
```

The `session` object in this example is a `ServletSessionPool` that can accept the `HttpServletRequest` object directly. Internally, it converts the data into a Jolt DataSet object, which contains the parameters for the `TOUPPER` service.

Note: The `TOUPPER` service expects a case-sensitive parameter called `"STRING"`, so it is essential for the text field within the HTML form to be labeled exactly the same, that is, `"STRING"`. Note also that the other data fields, such as the *`SVCNAME,`* are not relevant as parameters but don't disrupt the Tuxedo service.

The form parameter is used to actually name the service, which you don't have to pass as a service parameter. It is passed automatically because it is already contained in the `HttpServletRequest` object.

The `TOUPPER` service converts the text in the `"STRING"` parameter to uppercase text and passes it back to the servlet in a `ServletResult` object that contains the results of an executed call, as well as details about exceptions if any are thrown during the service call.

Step 9. Return the Results to the Client

The final step constructs and sends an HTML page, which contains the results of the service call, back to the client through the HttpResponse output stream. The uppercase result is retrieved from the `ServletResult` object using the `result.getValue()` method.

The following is a simple example of passing this data back as HTML that the browser can display:

```
out.println("<p><center><font size=+1><b>"+
            result.getValue("STRING", "")+
            "</b></font></center><p><hr
            width=80%>");
```

The output stream produces a page similar to the one shown in Figure B-2:

Figure B-2 Output Stream Results Example

APPENDIX C

Servlet with Enterprise JavaBean Example

To use the Servlet with Enterprise JavaBean example, see the following sections:

- About the Servlet with JavaBean Example
- Preparing to Use the Servlet with JavaBean Example
- Run the Servlet with JavaBean Example

This Enterprise JavaBean (EJBean) example package contains the classes and other files necessary to set up and run an EJBean stateful session to a Tuxedo Server using Jolt. The package contents are as follows:

- Client application (client application documentation and source)
- Deployment
  - `DeploymentDescriptor.txt`
  - `manifest`
- Interfaces
  - `Teller` (remote interface documentation and source)
  - `TellerHome` (home interface documentation and source)
  - `TellerResult` (application-specific utility documentation and source)
  - `ProcessingErrorException` (application-specific exception documentation and source)

  - `TransactionErrorException` (application-specific exception documentation and source)

- Server (EJBean)
  - `TellerBean` (EJBean documentation and source)

About the Servlet with JavaBean Example

This example demonstrates an Enterprise JavaBean (EJBean), and provides an example of a simple interface for accessing the Tuxedo Server. You can find the source code for this example in the `/samples/jolt/wls/ejb/bankapp` directory included in the Oracle Tuxedo distribution.

Running this example before attempting to create your own EJBeans will show you the different steps involved. The example is a stateful session EJBean called `TellerBean` that contacts a Tuxedo Server using Jolt for WebLogic, and conducts transactions as follows:

- Contacts and calls a Tuxedo Server, and retrieves the returned results
- Uses a session EJBean
- Uses stateful persistence
- Uses application-defined exceptions and utilities
- Uses a client browser application

The client browser application performs these steps:

1. Contacts the teller home ("TellerHome") through JNDI to find the EJBean.
2. Creates a teller ("Terry").
3. The application then performs a series of transactions for the Teller that has just been created:
   - Gets the current balance for account 10000.
   - Performs Transaction 1: Deposits $100 into the account, and displays the balance.
   - Performs Transaction 2: Deposits $200 (more than the transaction limit of $300).

   Note: In Transaction 1, a single call is made, and is automatically committed. In Transaction 2, a `begin()` and `commit()` bracket two separate requests (a deposit and a withdrawal).

   - Attempts to withdraw $100 more than the balance of the account.

- Catches an ApplicationException, retrieves the status messages embedded in the exception, and rolls back Transaction 2.
- Gets the final balance for the account.
- Removes the teller.

You can see in Transaction 2 how the balance is successfully rolled back to what it was at the end of Transaction 1.

Preparing to Use the Servlet with JavaBean Example

To get the most out of this example, first read through the source code files to see what is happening. Start with `DeploymentDescriptor.txt` to find the general structure of the EJBean and which classes are used for the different objects and interfaces, and then look at `Client.java` to see how the application works.

The following sections provide details for using this example:

- Set Up Your Environment
- Build the Example
- Run the Servlet with JavaBean Example

Set Up Your Environment

You need to add a Jolt connection pool that connects to the public Tuxedo Server at Oracle, as described in "Step 3. Configure the Servlet in WebLogic Server," in *Appendix B, Simple Servlet Example*. When you're finished, the `config.xml` configuration file will contain the following sections:

```
<StartupClass
  ClassName="bea.jolt.pool.servlet.weblogic.PoolManagerStartUp"
  FailureIsFatal="false"
  Name="MyStartup Class"
  Targets="myserver"
/>
<JoltConnectionPool
   ApplicationPassword="tuxedo"
   MaximumPoolSize="5"
   MinimumPoolSize="3"
   Name="MyJolt Connection Pool"
```

```
   PrimaryAddresses="//TUXSERVER:6309"
   RecvTimeout="300"
   SecurityContextEnabled="true"
   Targets="myserver"
   UserName="joltuser"
   UserPassword="jolttest"
   UserRole="clt"
/>|
<ShutdownClass
  ClassName=”bea.jolt.pool.servlet.weblogic.PoolManager
  ShutDown.”
/>
```

Build the Example

After configuring your WebLogic Server development environment, you need to build the example. Oracle Jolt provides separate build scripts for Windows 2003 and UNIX, as follows:

- Windows 2003: `%TUXDIR%\samples\jolt\wls\ejb\bankapp\build.cmd`
- UNIX: `$TUXDIR/samples/jolt/wls/ejb/bankapp/build.sh`

The scripts build individual examples, such as this entry for Windows 2003:

```
$ build
```

To build under Microsoft’s JDK for Java, use:

```
$ build -ms
```

The scripts will build the example and place the files in the following default WebLogic Server directories on a Windows 2003 system:

- Client files in: `d:\bea\wlserver6.1\config\examples`
- EJBean in: `d:\bea\wlserver6.1\config\mydomain\applications`

Run the Servlet with JavaBean Example

When WebLogic Server is started in the default `\config\mydomain` directory, the EJBean example is automatically deployed in the `\applications` directory.

1. Start the WebLogic Server in the `\config\mydomain` directory. You can check that the EJBean has been deployed correctly either by checking the server command-line window, or by opening the Console and examining EJB under Deployments. You should see `ejb.jolt.bankapp` deployed and should be able to monitor its activity.

2. Open a separate command-line window, and then run the client by entering the following command:

   ```
   $ java examples.jolt.ejb.bankapp.Client
   ```

   If you are not running the WebLogic Server with its default settings, you will have to use the following command line:

   ```
   $ java examples.jolt.ejb.bankapp.Client "t3://WebLogicURL:Port"
   ```

   where the following parameters are defined as follows:

   – WebLogicURL—the domain address of the WebLogic Server

   – Port—the port listening for connections (`weblogic.system.ListenPort`)

   The following optional parameters are interpreted by the client in the order in which they are listed:

   – url—unique resource location of Server, such as `t3://localhost:7001`

   – user—username, default null

   – password—user password, default null

3. If you are running the `Client` example, you should get output that is similar to the following from the client application:

   ```
   4.Beginning jolt.bankapp.Client...
   5.
   6.Created teller Terry
   7.
   8.Getting current balance of Account 10000 for Erin
   9.Balance: 27924.02
   10.
   11.Start Transaction 1 for Erin
   12.
   13.  Depositing 100.0 for Erin
   14.  Balance: 28024.02
   15.
   16.End Transaction 1 for Erin
   17.
   18.Start Transaction 2 for Erin
   19.
   20.  Depositing 200.0 for Erin
   ```

```
21.  Balance: 28224.02
22.
23.  Withdrawing 28324.02 for Erin
24.  Transaction error:
25.  examples.jolt.ejb.bankapp.TransactionErrorException: Teller error:
application
26. exception:
27.Account Overdraft
28.
29.  Rolling back transaction for Erin
30.
31.End Transaction 2 for Erin
32.
33.Getting final balance of Account 10000 for Erin
34.Balance: 28024.02
35.
36.Removing teller Terry
37.
End jolt.bankapp.Client...
```

Note: Note how the final balance shows that *Transaction 2* was rolled back to the balance at the end of *Transaction 1*.

You can read more about EJBs in the *Programming WebLogic Enterprise JavaBeans* guide. To learn more about using Oracle Jolt, refer to the *Using Oracle Jolt* guide.