SUN SEEBEYOND

eWAY™ ADAPTER FOR WEBLOGIC USER'S GUIDE

Release 5.1.1

Version 20060615170420

Contents

Chapter 1

Introducing the WebLogic eWay

Welcome to the *Sun SeeBeyond eWay™ Adapter for WebLogic User's Guide.* This document describes the integration between BEA WebLogic™ application Server and the Sun Java Composite Application Platform Suite (Java CAPS) using the WebLogic eWay Adapter (referred to as the WebLogic eWay throughout this document).

What's in This Chapter

1.1 About WebLogic Application Server

WebLogic Server

BEA defines WebLogic Server as a fully featured, standards-based, application server providing the foundation on which an enterprise builds its applications. More specifically, WebLogic Application Server is used to build Web applications that share data and resources with other systems, and then generate dynamic information for Web pages and other user interfaces.

WebLogic Application Server streamlines the process of building distributed, scalable, highly available systems by offering services that users previously had to write themselves, including connectivity, business logic, re-usability, security, concurrency (access is serialized), and transactionally (using XA to assure a successful transfer/update or rollback).

Other features offered by WebLogic include:

- **Object Pooling** – conserves system resources by placing objects in a pool, so that the next request for the object does not require a re-allocation of memory.

- **Thread and Connection Pooling** – works much the same way as Object Pooling to save memory and connection resources.
- **Clustering** – allows easy movement or distribution of applications to other machines.

1.2 About the WebLogic eWay

The WebLogic eWay is an application specific eWay that facilitates integration between applications built on the WebLogic platform and eGate using the Enterprise Java Bean (EJB) component model (synchronous communication). The eWay also provides an example of how to exchange JMS messages between WebLogic server and eGate asynchronously.

1.3 What's New in This Release

The WebLogic eWay includes the following changes and new features:

- Version Control: An enhanced version control system allows you to effectively manage changes to the eWay components.
- Multiple Drag-and-Drop Component Mapping from the Deployment Editor: The Deployment Editor now allows you to select multiple components from the Editor's component pane, and drop them into your Environment component.
- Support for Runtime LDAP Configuration: eWay configuration properties now support LDAP key values.
- Connectivity Map Generator: Generates and links your Project's Connectivity Map components using a Collaboration or Business Process.
- Dependency on Client Libraries: Uses IIOP to reduce dependency on specific Application Server client libraries.
- WebLogic 9: Provides added support of WebLogic 9 for synchronous communication.

Many of these features are documented further in the *Sun SeeBeyond eGate™ Integrator User's Guide* or the *Sun SeeBeyond eGate™ Integrator System Administration Guide*.

1.4 About This Document

This document includes the following chapters:

- **Chapter 1 "Introducing the WebLogic eWay"**: Provides an overview description of the product as well as high-level information about this document.

- **Chapter 2 "Installing the WebLogic eWay"**: Describes the system requirements and provides instructions for installing the WebLogic eWay.
- **Chapter 3 "Setting Properties of the WebLogic eWay"**: Provides instructions for configuring the eWay to communicate with your legacy systems.
- **Chapter 4 "WebLogic Server Components"**: Provides an overview various Sun Microsystem Java 2 Enterprise Edition (J2EE) Applications and WebLogic Server technologies employed in the WebLogic Server.
- **Chapter 5 "WebLogic eWay Component Communication"**: Provides an overview of how components of the WebLogic eWay Adapter communicate with the WebLogic Application Server.
- **Chapter 6 "Configuring WebLogic Server for Asynchronous Communication"**: Provides directions for configuring WebLogic Server for asynchronous interaction with eGate.
- **Chapter 7 "Using the WebLogic OTD Wizard"**: Provides instructions for creating Object Type Definitions (OTDs) to be used with the WebLogic eWay.
- **Chapter 8 "Implementing the WebLogic eWay Sample Projects":** Provides instructions for installing and running the sample Projects.
- **Chapter 9 "Sun Microsystems Sample Message Driven Beans"**: Provides further information on the messaging objects designed to route messages from clients to other Enterprise Java Beans.

1.4.1 Scope

This user's guide provides a description of the WebLogic eWay Adapter. It includes directions for installing the eWay, configuring the eWay properties, and implementing the eWay's sample Projects. This document is also intended as a reference guide, listing available properties, functions, and considerations.

1.4.2 Intended Audience

This guide is intended for experienced computer users who have the responsibility of helping to set up and maintain a fully functioning Java Composite Application Platform Suite system. This person must also understand any operating systems on which the Java Composite Application Platform Suite will be installed (Windows and UNIX), and must be thoroughly familiar with Windows-style GUI operations.

1.4.3 Text Conventions

The following conventions are observed throughout this document.

Table 1 Text Conventions

Text Convention	Used For	Examples
Bold	Names of buttons, files, icons, parameters, variables, methods, menus, and objects	▪ Click **OK**. ▪ On the **File** menu, click **Exit**. ▪ Select the **eGate.sar** file.
`Monospaced`	Command line arguments, code samples; variables are shown in ***`bold italic`***	`java -jar ***filename***.jar`
Blue bold	Hypertext links within document	See **Text Conventions** on page 9
Blue underlined	Hypertext links for Web addresses (URLs) or email addresses	http://www.sun.com

1.5 Related Documents

The following Sun documents provide additional information about the Sun Java Composite Application Platform Suite product:

- *Sun SeeBeyond eGate™ Integrator User's Guide*
- *Sun Java Composite Application Platform Suite Installation Guide*

1.6 Sun Microsystems, Inc. Web Site

The Sun Microsystems web site is your best source for up-to-the-minute product news and technical support information. The site's URL is:

http://www.sun.com

1.7 Documentation Feedback

We appreciate your feedback. Please send any comments or suggestions regarding this document to:

CAPS_docsfeedback@sun.com

Chapter 2

Installing the WebLogic eWay

This chapter explains how to install the WebLogic eWay.

What's in This Chapter

2.1 WebLogic eWay System Requirements

The WebLogic eWay Readme contains the latest information on:

- Supported Operating Systems
- System Requirements
- External System Requirements

The WebLogic eWay Readme is uploaded with the eWay's documentation file (**WebLogiceWayDocs.sar**) and can be accessed from the **Documentation** tab of the Sun Java Composite Application Platform Suite Installer. Refer to the WebLogic eWay Readme for the latest requirements before installing the WebLogic eWay.

2.2 Installing the WebLogic eWay

The Sun Java Composite Application Platform Suite Installer, a web-based application, is used to select and upload eWays and add-on files during the installation process. The following section describes how to install the components required for this eWay.

Note: *When the Repository is running on a UNIX operating system, the eWays are loaded from the Sun Java Composite Application Platform Suite Installer running on a Windows platform connected to the Repository server using Internet Explorer.*

2.2.1 Installing the WebLogic eWay on an eGate supported system

Follow the directions for installing the Sun Java Composite Application Platform Suite in the *Sun Java Composite Application Platform Suite Installation Guide*. After you have installed Core Products, do the following:

1. From the Sun Java Composite Application Platform Suite Installer's **Select Sun Java Composite Application Platform Suite Products Installed** table (Administration tab), click the **Click to install additional products** link.
2. Expand the **eWay** option.
3. Select the products for your Sun Java Composite Application Platform Suite and include the following:
   - **File eWay** (the File eWay is used by most sample Projects)
   - **WebLogicWay**

   To upload the WebLogic eWay User's Guide, Help file, Readme, and sample Projects, expand the **Documentation** option and select **WebLogicWayDocs**.
4. Once you have selected all of your products, click **Next** in the top-right or bottom-right corner of the **Select Sun Java Composite Application Platform Suite Products to Install** box.
5. From the **Selecting Files to Install** box, locate and select your first product's SAR file. Once you have selected the SAR file, click **Next**. Your next selected product appears. Follow this procedure for each of your selected products. The **Installation Status** window appears and installation begins after the last SAR file has been selected.
6. Once your product's installation is finished, continue installing the Sun Java Composite Application Platform Suite as instructed in the *Sun Java Composite Application Platform Suite Installation Guide*.

Adding the eWay to an Existing Java CAPS Installation

If you are adding the eWay to an existing Sun Java Composite Application Platform Suite installation, do the following:

1. Complete steps 1 through 4 above.
2. Once your product's installation is complete, open the Enterprise Designer and select **Update Center** from the Tools menu. The **Update Center Wizard** appears.
3. For Step 1 of the wizard, simply click **Next**.
4. For Step 2 of the wizard, click the **Add All** button to move all installable files to the **Include in Install** field, then click **Next**.
5. For Step 3 of the wizard, wait for the modules to download, then click **Next**.
6. The wizard's Step 4 window displays the installed modules. Review the installed modules and click **Finish.**
7. When prompted, restart the IDE (Integrated Development Environment) to complete the installation.

After Installation

Once you install the eWay, it must then be incorporated into a Project before it can perform its intended functions. See the *eGate Integrator User's Guide* for more information on incorporating the eWay into an eGate Project.

2.2.2 Extracting the Sample Projects

The WebLogic eWay includes sample Projects. The sample Projects are designed to provide you with a basic understanding of how certain database operations are performed using the eWay.

Steps to extract the Sample Projects include:

1. Click the **Documentation** tab of the Sun Java Composite Application Platform Suite Installer, then click the **Add-ons** tab.
2. Click the **WebLogic eWay Adapter** link. Documentation for the WebLogic eWay appears in the right pane.
3. Click the icon next to **Sample Projects** and extract the ZIP file. Note that the **WebLogic_eWay_Sample.zip** file contains an additional ZIP file for each sample Project.

   Refer to **"Importing a Sample Project" on page 69** for instructions on importing the sample Project into your repository via the Enterprise Designer.

2.3 ICAN 5.0 Project Migration Procedures

This section describes how to transfer your current ICAN 5.0.x Projects to the Sun Java Composite Application Platform Suite 5.1.1. To migrate your ICAN 5.0.x Projects to the Sun Java Composite Application Platform Suite 5.1.1, do the following:

Export the Project

1. Before you export your Projects, save your current ICAN 5.0.x Projects to your Repository.
2. From the Project Explorer, right-click your Project and select **Export** from the shortcut menu. The Export Manager appears.
3. Select the Project that you want to export in the left pane of the Export Manager and move it to the Selected Projects field by clicking the **Add to Select Items** (arrow) button, or click **All** to include all of your Projects.
4. In the same manner, select the Environment that you want to export in the left pane of the Export Manager and move it to the Selected Environments field by clicking the **Add to Select Items** (arrow) button, or click **All** to include all of your Environments.
5. Browse to select a destination for your Project ZIP file and enter a name for your Project in the **ZIP file** field.

6 Click **Export** to create the Project ZIP file in the selected destination.

Install Java CAPS 5.1.1

1 Install **Java CAPS 5.1.1**, including all eWays, libraries, and other components used by your ICAN 5.0 Projects.

2 Start the Java CAPS 5.1.1 Enterprise Designer.

Import the Project

1 From the Java CAPS 5.1.1 Enterprise Designer's Project Explorer tree, right-click the Repository and select **Import Project** from the shortcut menu. The Import Manager appears.

2 Browse to and select your exported Project file.

3 Click **Import**. A warning message, **"Missing APIs from Target Repository**," may appear at this time. This occurs because various product APIs were installed on the ICAN 5.0 Repository when the Project was created that are not installed on the Java CAPS 5.1.1 Repository. These APIs may or may not apply to your Projects. You can ignore this message if you have already installed all of the components that correspond to your Projects. Click **Continue** to resume the Project import.

4 Close the Import Manager after the Project is successfully imported.

Deploy the Project

1 A new Deployment Profile must be created for each of your imported Projects. When a Project is exported, the Project's components are automatically *"checked in"* to Version Control to write-protected each component. These protected components appear in the Explorer tree with a red padlock in the bottom-left corner of each icon. Before you can deploy the imported Project, the Project's components must first be *"checked out"* of Version Control from both the Project Explorer and the Environment Explorer. To *"check out"* all of the Project's components, do the following:

   A From the Project Explorer, right-click the Project and select **Version Control > Check Out** from the shortcut menu. The Version Control - Check Out dialog box appears.

   B Select **Recurse Project** to specify all components, and click **OK**.

   C Select the Environment Explorer tab, and from the Environment Explorer, right-click the Project's Environment and select **Version Control > Check Out** from the shortcut menu.

   D Select **Recurse Environment** to specify all components, and click **OK**.

2 If your imported Project includes File eWays, these must be reconfigured in your Environment prior to deploying the Project.

   To reconfigure your File eWays, do the following:

   A From the Environment Explorer tree, right-click the File External System, and select **Properties** from the shortcut menu. The Properties Editor appears.

   B Set the inbound and outbound directory values, and click **OK**. The File External System can now accommodate both inbound and outbound eWays.

3 Deploy your Projects.

Note: *Only projects developed on ICAN 5.0.2 and later can be imported and migrated successfully into the Sun Java Composite Application Platform Suite.*

2.4 Installing Enterprise Manager eWay Plug-Ins

The **Sun SeeBeyond Enterprise Manager** is a Web-based interface you use to monitor and manage your Sun Java Composite Application Platform Suite applications. The Enterprise Manager requires an eWay specific "plug-in" for each eWay you install. These plug-ins enable the Enterprise Manager to target specific alert codes for each eWay type, as well as start and stop the inbound eWays.

The *Sun Java Composite Application Platform Suite Installation Guide* describes how to install Enterprise Manager. The *Sun SeeBeyond eGate Integrator System Administration Guide* describes how to monitor servers, Services, logs, and alerts using the Enterprise Manager and the command-line client.

The **eWay Enterprise Manager Plug-ins** are available from the **List of Components to Download** under the Sun Java Composite Application Platform Suite Installer's **Downloads** tab.

There are two ways to add eWay Enterprise Manager plug-ins:

- From the **Sun SeeBeyond Enterprise Manager**
- From the **Sun Java Composite Application Platform Suite Installer**

To add plug-ins from the Enterprise Manager

1 From the **Enterprise Manager**'s Explorer toolbar, click **configuration**.

2 Click the **Web Applications Manager** tab, go to the **Auto-Install from Repository** sub-tab, and connect to your Repository.

3 Select the application plug-ins you require, and click **Install**. The application plug-ins are installed and deployed.

To add plug-ins from the Sun Java Composite Application Platform Suite Installer

1 From the **Sun Java Composite Application Platform Suite Installer**'s **Downloads** tab, select the Plug-Ins you require and save them to a temporary directory.

2 From the **Enterprise Manager**'s Explorer toolbar, click **configuration**.

3 Click the **Web Applications Manager** tab and go to the **Manage Applications** sub-tab.

4 Browse for and select the WAR file for the application plug-in that you downloaded, and click **Deploy**. The plug-ins is installed and deployed.

2.4.1 Viewing Alert Codes

You can view and delete alerts using the Enterprise Manager. An alert is triggered when a specified condition occurs in a Project component. The purpose of the alert is to warn the administrator or user that a condition has occurred.

To View the eWay Alert Codes

1. Add the eWay Enterprise Manager plug-in for this eWay.
2. From the **Enterprise Manager**'s Explorer toolbar, click **configuration**.
3. Click the **Web Applications Manager** tab and go to the **Manage Alert Codes** sub-tab. Your installed eWay alert codes display under the **Results** section. If your eWay alert codes are not displayed under **Results**, do the following:
   A. From the **Install New Alert Codes** section, browse to and select the eWay alert properties file for the application plug-in that you added. The alert properties files are located in the **alertcodes** folder of your Sun Java Composite Application Platform Suite installation directory.
   B. Click **Deploy**. The available alert codes for your application are displayed under **Results**. A listing of the eWay's available alert codes is displayed in Table 2.

Table 2 WebLogic eWay Alert Codes

Alert Code	Description	User Action
EJB-ERRORENCOUNTERED000001	Error encountered in EJB eWay. Check debug logs for details.	An error has occured, such as: ▪ EJB has not been deployed. ▪ Configuration properties are not correct. ▪ JNDI name is invalid. ▪ OTD uses an incompatible version of the EJB. Refer to the log for more information.

For information on Managing and Monitoring alert codes and logs, as well as how to view the alert generated by the project component during runtime, see the *Sun SeeBeyond eGate™ Integrator System Administration Guide*.

Note: *An alert code is a warning that an error has occurred. It is not a diagnostic. The user actions noted above are just some possible corrective measures you may take. Refer to the log files for more information. For information on Managing and Monitoring alert codes and logs, see the Sun SeeBeyond eGate Integrator System Administration Guide.*

Chapter 3

Setting Properties of the WebLogic eWay

This chapter describes how to configure the WebLogic eWay properties, and provides a list of the eWay properties and their required values.

What's in This Chapter

3.1 Configuring the WebLogic eWay Properties

The WebLogic eWay includes a unique set of configuration parameters. After creating the eWays and the WebLogic External System in the Project's Environment, the property parameters can be modified for your specific system.

3.1.1 Selecting WebLogic as the External Application

To create a WebLogic eWay you must first create a WebLogic External Application in your Connectivity Map. WebLogic eWays are located between a WebLogic External Application and a Service. Services are containers for Java Collaborations, Business Processes, eTL processes, and so forth.

To create the WebLogic External Application

1. From the Connectivity Map toolbar, click the External Applications icon.
2. Select the **WebLogic External Application** from the menu. The selected WebLogic External Application icon appears on the Connectivity Map toolbar.

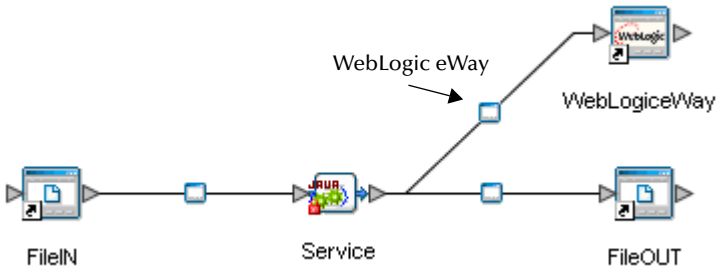
   The new External System can now be dragged and dropped onto the Connectivity Map canvas and incorporated into a Project.

3.1.2 Accessing the eWay Properties

When you connect an External Application to a Collaboration, the Enterprise Designer automatically assigns the appropriate eWay to the link (Figure 1). Each eWay is

supplied with a template containing default configuration properties that are accessible from the Connectivity Map and Environment Explorer Tree.

Figure 1 Connectivity Map with Components



3.1.3 Modifying the WebLogic eWay Properties

The eWay properties can be modified after the eWays have been created in the Connectivity Map and the Project's Environment has been created. WebLogic eWay properties are modified from two locations: from the Connectivity Map and from the Environment Explorer tree.

Modifying the eWay Connectivity Map Properties

The Connectivity Map parameters most commonly apply to a specific component eWay, and may vary from other eWays (of the same type) in the Project.

1. From the Connectivity Map, double click the eWay icon, located in the link between the associated External Application and the Service.
2. The eWay **Properties Editor** opens with the WebLogic eWay Connectivity Map properties. Make any necessary modifications and click **OK** to save the settings.

Modifying the eWay Environment Properties

These parameters are commonly global, applying to all eWays (of the same type) in the Project. The saved properties are shared by all eWays for the specified External System.

1. From the Environment Explorer tree, right-click the WebLogic External System. Select **Properties** from the shortcut menu. The **Properties Editor** opens with the WebLogic eWay Environment properties.
2. Make any necessary modifications to the Environment properties, and click **OK** to save the settings.

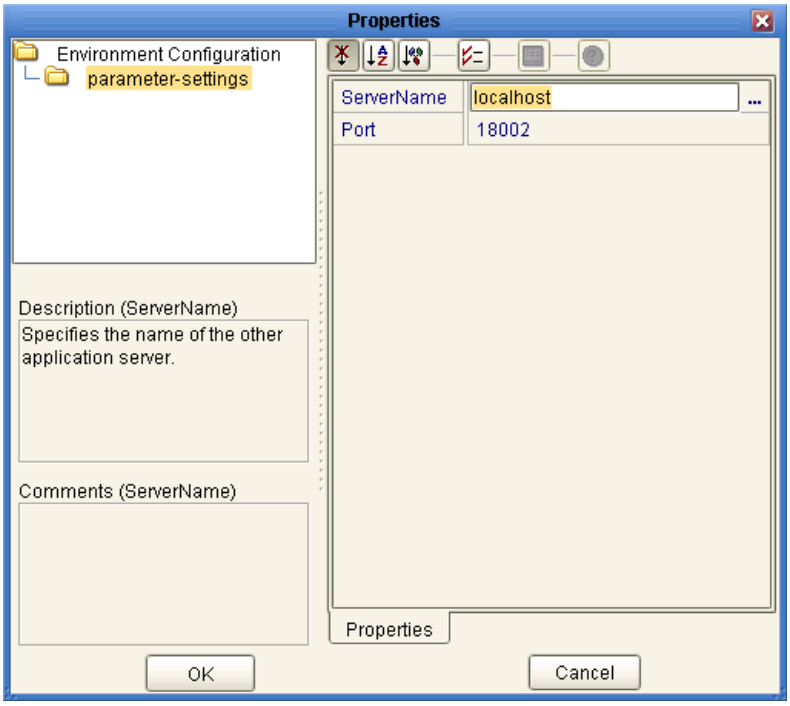
3.1.4 Using the Properties Editor

Modifications to the eWay properties are made using the WebLogic eWay Properties Editor.

Modifying the Default eWay Properties

1 From the upper-left pane of the Properties Editor, select a subdirectory of the Properties tree. The parameters contained in that subdirectory are now displayed in the right pane of the Properties Editor. For example, clicking on the **parameter-settings** subdirectory displays the editable parameters in the right pane, as shown in Figure 2

Figure 2 Properties Editor: WebLogic eWay



2 Click on any property field to make it editable. For example, click on the **ServerName** property to edit the properties settings. If a parameter's value is true/false or multiple choice, the field reveals a submenu of property options.

Click on the ellipsis (. . .) in the properties field (displayed when you click on the field). A separate configuration dialog box appears. This is helpful for large values that cannot be fully displayed in the parameter's property field. Enter the property value in the dialog box and click **OK**. The value now appears in the property field.

3 A description of each parameter is displayed in the **Description** pane when that parameter is selected, providing an explanation of any required settings or options.

4 The **Comments** pane provides an area for recording notes and information about the currently selected parameter. This is saved automatically for future referral.

5 Click **OK** to close the Properties Editor and save the changes.

3.2 Specifying JNDI Names

The Connectivity Map and Environment properties are used to specify JNDI names. There are two methods you can use to specify a JNDI name:

- Specify the **server name** and **port number** in the Environment properties, and specify a **JNDI name** in the Connectivity Map, **ejbJndiName** property. The eWay concatenates the server name and port number with the JNDI name and provides the qualified JNDI name.

  For example, if the specified server name is **MyServer**, the specified port number is **1111**, and the JNDI name is **ejb/MyStorageBin**, the eWay uses these property to construct the qualified JNDI name:

  ```
  corbaname:iiop:1.2@MyServer:1111#ejb/MyStorageBin
  ```

- Specify a qualified JNDI name in the Connectivity Map property, **ejbJndiName**. This name supercedes any values specified in the Environment properties. For example:

  ```
  corbaname:iiop:1.2@MyServer:1111#ejb/MyStorageBin
  ```

3.3 WebLogic eWay Properties

The WebLogic properties are organized into the following sections:

- **WebLogic eWay Connectivity Map Properties** on page 20
- **WebLogic eWay Environment Properties** on page 21

3.3.1 WebLogic eWay Connectivity Map Properties

The eWay properties, accessed from the Connectivity Map, are organized into the following sections:

parameter-settings (Connectivity Map) on page 20

parameter-settings (Connectivity Map)

The **parameter-settings** section of the Connectivity Map properties contains the top level parameters displayed in Table 3.

Table 3 Connectivity Map - parameter-settings

Name	Description	Required Value
EjbJndiName	Specifies the JNDI name of the EJB on the remote server including any JNDI prefixes for the EJB interoperability. If the EJB and your Project EAR file are deployed on the Sun Java System Application Server or the Sun SeeBeyond Integration Server, you can use the `localEJB:` prefix as the qualified JNDI name. For example: `localEJB:ejb/MyStorageBin` without the `corbaname/IIOP` syntax. If a qualified JNDI name is specified in this property, this supercedes any values specified in the Environment properties: ServerName and Port (see **"Specifying JNDI Names" on page 20**.	Enter a JNDI name (String) or a qualified JNDI name.

3.3.2 WebLogic eWay Environment Properties

The eWay properties, accessed from the Environment Explorer tree, are organized into the following sections:

parameter-settings (Environment) on page 21

parameter-settings (Environment)

The **parameter-settings** section of the Environment properties contains the top level parameters displayed in Table 4.

Table 4 Environment - **parameter-settings**

Name	Description	Required Value
ServerName	Specifies the name of the other application server. If a qualified JNDI name is specified in the Connectivity Map property, EjbJndiName, that value supercedes any values specified in the Environment properties: ServerName and Port (see **"Specifying JNDI Names" on page 20**.	A name of an application server. The default is **localhost**.

Table 4 Environment - **parameter-settings** (Continued)

Name	Description	Required Value
Port	Specifies the Corba Port of the other application server. See **"Specifying JNDI Names" on page 20**.	An integer indicating a port number. The default is **18002**.

Chapter 4

WebLogic Server Components

This chapter provides an overview of the various Sun Microsystem Java 2 Enterprise Edition (J2EE) Applications and WebLogic Server technologies employed in the WebLogic Server.

What's in This Chapter:

4.1 Java Naming and Directory Interface (JNDI)

The JNDI service is a set of APIs published by Sun that interface to a directory to locate named objects. APIs allow Java programs to store and lookup objects using multiple naming services in a standard manner. The naming service may be either LDAP, a file system, or a RMI registry. Each naming service has a corresponding provider implementation that can be used with JNDI. The ability for JNDI to "plug in" any implementation for any naming service (or span across naming services with a federated naming service) easily provides another level of programming abstraction. This level of abstraction allows Java code using JNDI to be portable against any naming service. For example, no code changes should be needed by the Java client code to run against an RMI registry or an LDAP server.

The WebLogic Naming Service

Any J2EE compliant application server, such as the WebLogic Server, has a JNDI subsystem. The JNDI subsystem is used in an Application Server as a directory for such objects as resource managers and Enterprise JavaBeans (EJBs). Objects managed by the WebLogic container have default environments for getting the JNDI **InitialContext** loaded when they use the default **InitialContext()** constructor. For a Collaboration using a WebLogic EJB Object Type Definition (OTD) to find the home interface of an EJB, the JNDI properties must be configured and associated with the OTD. However, for other external clients, accessing the WebLogic naming service requires a Java client program that sets up the appropriate JNDI environment when creating the JNDI Initial Context.

There are essentially two environments that have to be configured, **Context.PROVIDER_URL** and **Context.INITIAL_CONTEXT_FACTORY**. For WebLogic, the Context.PROVIDER_URL environment is

```
t3://<wlserverhost>:<port>/
```

where <wlserverhost> is the hostname on which the WebLogic Server instance is running and <port> is the port at which the Webserver instance is listening for connections. For example:

```
t3://localhost:7001/
```

The initial context factory class for the WebLogic JNDI is **weblogic.jndi.WLInitialContextFactory**. This class should be supplied to the **Context.INITIAL_CONTEXT_FACTORY** environment property when constructing the initial context. The overloaded **InitialContext(Map) constructor** must be used in this case.

Sample Code

The following code is an example of creating an initial context to WebLogic JNDI from a stand-alone client:

```
HashMap env = new HashMap();
env.put (Context.PROVIDER_URL, "t3://localhost:7001/");
env.put (Context.INITIAL_CONTEXT_FACTORY,
"weblogic.jndi.WLInitialContextFactory");
Context initContext = new InitialContext (env);
…
```

Once an initial context is created, sub-contexts can be created, objects can be bound, and objects can be retrieved using the initial context. For example the following segment of code retrieves a Topic object:

```
Topic topic
=(Topic)initContext.lookup("sbyn.inTopicToSunMicrosystemsTopic");
…
```

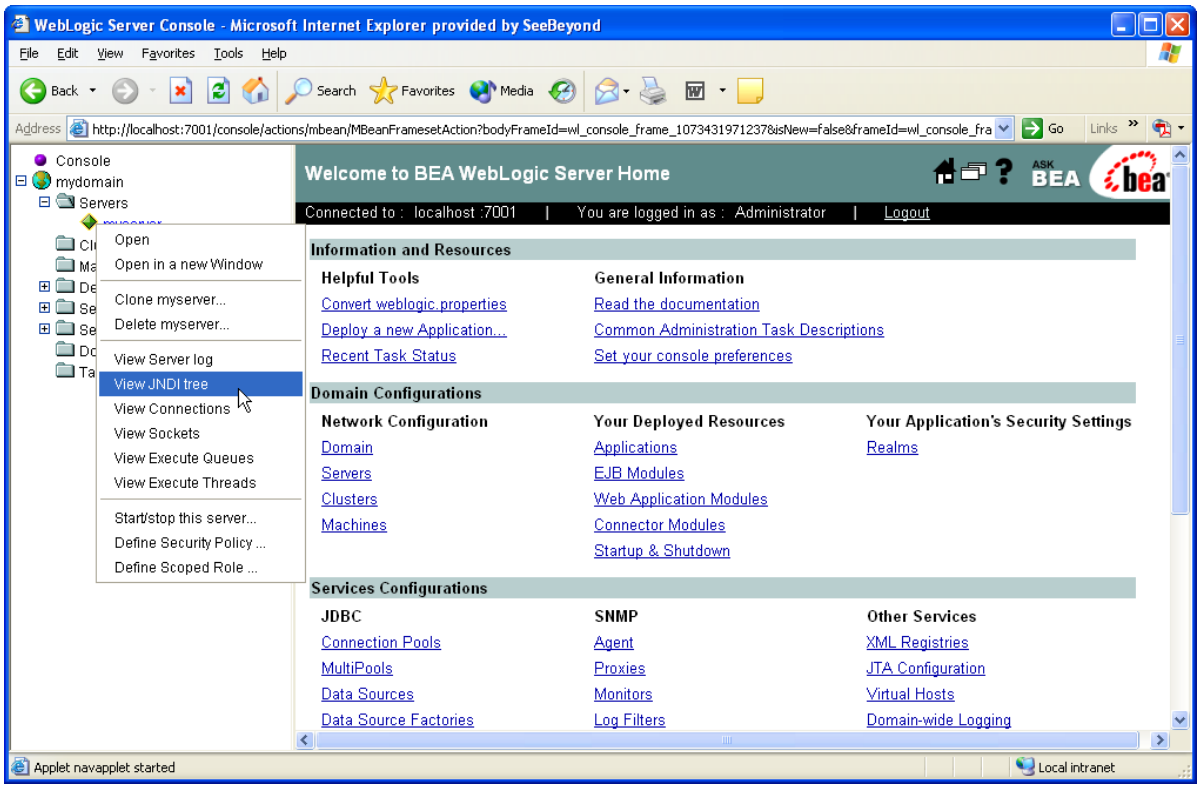
Here's an example of how to bind a Sun Microsystems Queue object:

```
Queue queue = null;
try {
   queue = new STCQueue("inQueueToSunMicrosystemsQueue");
   initContext.bind ("sbyn.ToSunMicrosystemsQueue", queue);
}
catch (NameAlreadyBoundException ex)
{
   try
   {
    if (queue != null)
      initContext.rebind ("sbyn.ToSunMicrosystemsQueue", queue);
   }
   catch (Exception ex)
   {
    throw ex;
   }
}
```

Viewing the WebLogic JNDI Tree

The WebLogic Administrative Console (Web Interface) allows a user to view the JNDI Tree associated with the server instance. To view the JNDI Tree (see Figure 3), log onto the Administrative console for the selected server (for example, the examplesServer), expand the **Servers** tab, right click on the server node, and select **View JNDI tree** from the pop up menu. Figure 3 displays the WebLogic Administrative Console for WebLogic 7.0.

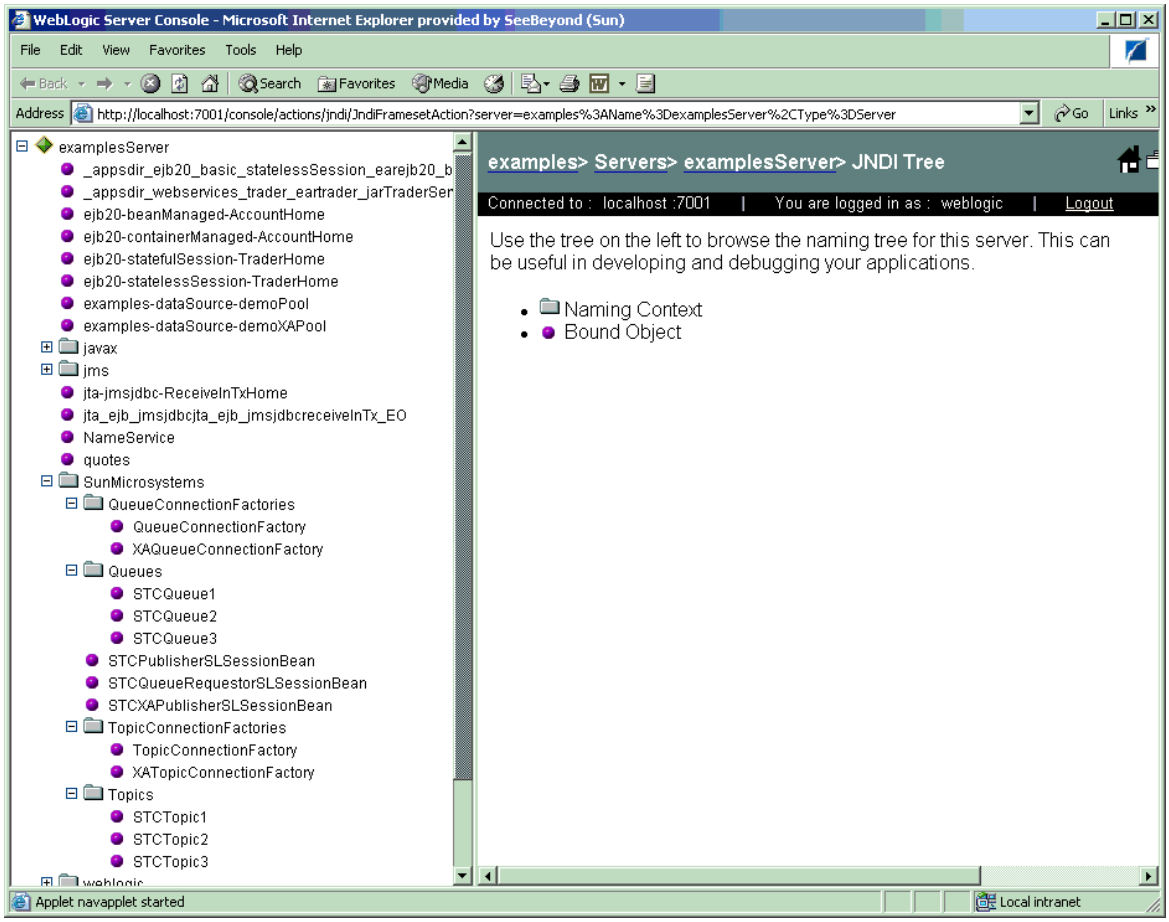
Figure 3 Administrative Console - View JNDI Tree



In the following example, (see Figure 4) the JNDI tree Web page shows that the **Sun Microsystems** subcontext was expanded in order to view the Sun Microsystems JMS objects that were bound to the WebLogic JNDI. These objects are bound when the **WeblogicStartup** class is loaded and run by the WebLogic Server. (See **"Sun Microsystems WebLogic Startup Class" on page 43** for more details about this startup class.)

Additionally, when EJBs are deployed on the application server they are registered in the JNDI. This JNDI name is used by the EJB OTD to look up the home interface of the EJB.

Figure 4 Administrative Console - The JNDI Tree Web Page



4.2 Java Messaging Service (JMS)

The Java Messaging Service is a messaging oriented middleware API designed by Sun. The client makes use of these APIs, allowing portability with any JMS implementation. JMS allows clients to be de-coupled from one another. The clients do not communicate with each other directly, but rather by send messages to each other via middleware. Each client in a JMS environment connects to a messaging server. The messaging server facilitates the flow of messages among all clients. The messaging server guarantees that all messages arrive at the appropriate destinations. The messaging server also guarantees quality of services as transactions (local or XA), persistence, durability, and others.

Clients send messages to or receive messages from **Topics** or **Queues** (see Figure 5 and Figure 6). The difference between a Topic and a Queue is that all subscribers to a Topic receive the same message when the message is published and only one subscriber to a Queue receives a message when the message is sent (see **"Sun Microsystems JMS" on page 35**).

Figure 5 Topic - The Publish-Subscribe Model

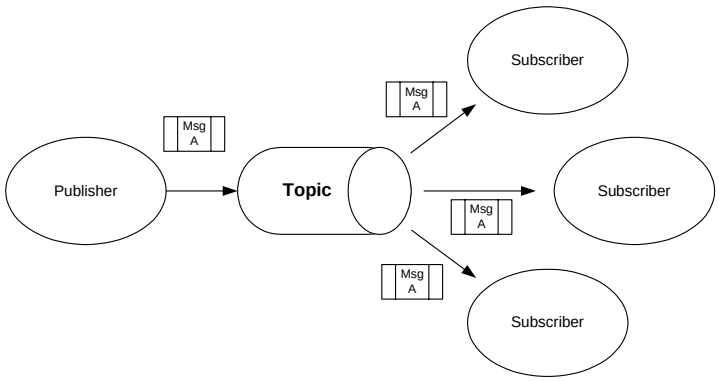
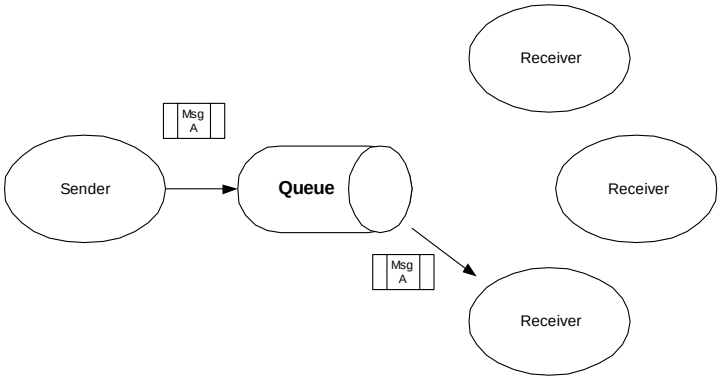


Figure 5 shows multiple subscribers receiving the same messages when the publisher publishes the message to a Topic. This is the pubsub (publish-subscribe) model.

Figure 6 Queue - The Point-to-Point Model



The Point-to-Point model (Figure 6), on the other hand, allows for only one receiver to get the message when a sender sends a message to a Queue.

4.3 Enterprise JavaBeans (EJBs)

Enterprise JavaBeans are reusable software programs that you can develop and assemble easily to create sophisticated applications. Developers use EJBs to design and develop customized, reusable business logic. EJBs are the units of work that an application server is responsible for and exposes to the external world. The WebLogic Application Server provides the architecture for writing business logic components, allowing Web servers to easily access data.

There are three types of Enterprise JavaBeans:

- Session Beans
- Entity Beans
- Message Driven Beans

Session Beans

Session Beans are business process objects that perform actions. An action may be opening an account, transferring funds, or performing a calculation. Session Beans consist of the remote, home, and bean classes. A client gets a reference to the Session Bean's home interface in order to create the Session Bean remote object, which is essentially the bean's factory. The Session Bean is exposed to the client with the remote interface. The client uses the remote interface to invoke the bean's methods. The actual implementation of the Session Bean is done with the bean class. (See **"Accessing Session Beans" on page 95**.)

Entity Beans

Entity Beans are data objects that represent the real-life objects on which Session Beans perform actions. Objects may include items such as accounts, employees, or inventory. An Entity Bean, like a Session Bean, consists of the remote, home, and bean classes. The client references the Entity Bean's home interface in order to create the Entity Bean remote object (essentially the bean's factory). The Entity Bean is exposed to the client with the remote interface, which the client uses to invoke the bean's methods. The implementation of the Entity Bean is done with the bean class. (See **"Entity Beans" on page 28**.)

Message Driven Beans

Message Driven Beans (MDBs) are messaging objects designed to route messages from clients to other Enterprise Java Beans. In the WebLogic eWay, MDBs are only supported with asynchronous communication with JMS. However, Message Driven Beans deal with asynchronous subscription/publication of JMS messages in a different manner than Entity and Session Beans (EJB 2.0 specification). Message Driven Beans are often compared to a Stateless Session Bean in that it does not have any state context. A Message Driven Bean differs from Session and Entity Beans in that it has no local/remote or localhome/home interfaces. An MDB is not exposed to a client at all. The MDB simply subscribes to a Topic or a Queue, receives messages from the container via the Topic or Queue, and then process the messages it receives from the container.

An MDB implements two interfaces: **javax.ejb.MessageBean** and **javax.jms.MessageListener**. Minimally, the MDB must implement the **setMessageDrivenContext**, **ejbCreate**, and **ejbRemove** methods from the javax.ejb.MessageBean interface. In addition, the MDB must implement the **onMessage** method of the **javax.jms.MessageListener** interface. The container calls the **onMessage** method, passing in a **javax.jms.Message**, when a message is available for the MDB.

4.4 XA Transactions

XA is a two-phase commit protocol that is natively supported by many databases and transaction monitors. It ensures data integrity by coordinating single transactions accessing multiple relational databases. XA guarantees that transactional updates are

committed in all of the participating databases, or are fully rolled back out of all of the databases, reverting to the state prior to the start of the transaction.

The **X/Open XA** specification defines the interactions between the Transaction Manager (TM) and the Resource Manager. The Transaction Manager, also known as the XA Coordinator, manages the XA or global transactions. The Resource Manager manages a particular resource such as a database or a JMS system. In addition, an XA Resource exposes a set of methods or functions for managing the resource.

In order to be involved in an XA transaction, the XA Resource must make itself known to the Transaction Manager. This process is called enlistment. Once an XA Resource is enlisted, the Transaction Manager ensures that the XA Resource takes part in a transaction and makes the appropriate method calls on the XA Resource during the lifetime of the transaction. For an XA transaction to complete, all the Resource Managers participate in a two-phase commit (2pc). A commit in an XA transaction is called a two-phase commit because there are two passes made in the committing process. In the first pass, the Transaction Manager asks each of the Resource Managers (via the enlisted XA Resource) whether they will encounter any problems committing the transaction. If any Resource Manager objects to committing the transaction, then all work done by any party on any resource involved in the XA transaction must all be rolled back. The Transaction Manager calls the **rollback()** method on each of the enlisted XA Resources. However, if no resource Managers object to committing, then the second pass involves the Transaction Manager actually calling **commit()** on each of the enlisted XA Resources. This process guarantees the ACID (atomicity, consistency, isolation, and durability) properties of a transaction that can span multiple resources.

Both Sun Microsystems JMS and BEA WebLogic Server implement the X/Open XA interface specifications. Because both systems support XA, the EJBs running inside the WebLogic container can subscribe or publish messages to Sun Microsystems JMS in XA mode. When running in XA mode, the EJBs subscribing or publishing to Sun Microsystems JMS can also participate in a global transaction involving other EJBs. For the "example" EJBs running in XA mode, Container Managed Transactions (CMTs) are used. In other words, we define the transactional attributes of the EJBs through their deployment descriptors and allow the container to transparently handle the XA transactions on behalf of the EJBs. The WebLogic Transaction Manager coordinates the XA transactions. The Sun Microsystems JMS XA Resource is enlisted to a transaction so that the WebLogic Transaction Manager is aware of the Sun Microsystems JMS XA Resource involved in the XA transaction. The WebLogic container interacts closely with the Transaction Manager in CMT such that transactions are almost transparent to an EJB developer. (See **"Sun Microsystems Sample XA Session Beans" on page 105**.)

Chapter 5

WebLogic eWay Component Communication

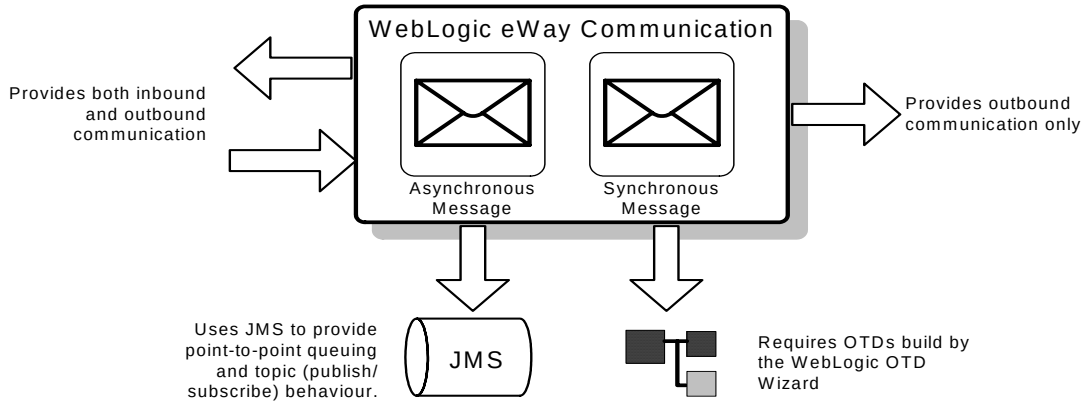
This chapter provides an overview of how components of the WebLogic eWay Adapter communicate with the WebLogic Application Server.

What's in This Chapter:

5.1 Synchronous and Asynchronous Communication

WebLogic eWay takes advantage of both Synchronous and Asynchronous communication in message delivery. Asynchronous messages provide both inbound and outbound communication between eGate and WebLogic, using the WebLogic JMS. Synchronous messages only provide outbound communication and require OTDs to hold the data structure and define rules referenced in the EJB.

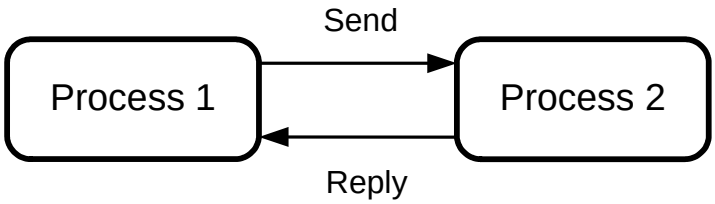
Figure 7 WebLogic Synchronous and Asynchronous Communication



5.1.1 Synchronous Communication

Synchronous communication is considered an unbuffered process, requiring either complete data transmission and reply or confirmation of message transmission failure before continuing with the process. This can be comparable to a phone call in which the caller makes the call and waits for a response before attempting to make another call. An example of synchronous communication is displayed in Figure 8.

Figure 8 Synchronous Communication



Synchronous Communication in eGate Includes:

- **eGate to WebLogic Transactions** – an outbound transaction, where eGate makes a request to WebLogic and waits for a response. For more information, see **"Synchronous Communication in eGate" on page 33**.

Associated Sample Projects:

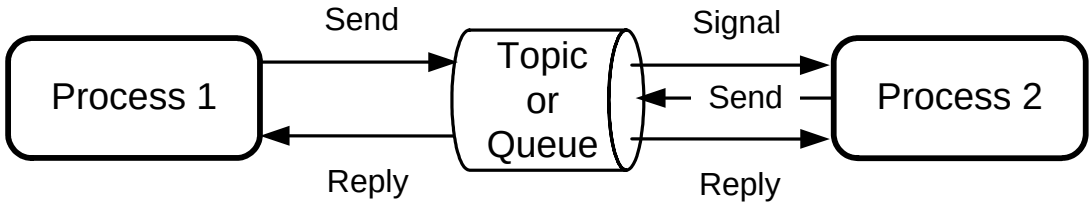
Two sample projects—**prjWebLogic_Sample_BPEL** and **prjWebLogic_Sample_JCD**—are included with the WebLogic eWay to demonstrate synchronous message interactions.

- **prjWebLogic_Sample_BPEL** – demonstrates how to deploy an eGate component as an Activity in an eInsight Business Process. For more information, see **"Implementing the WebLogic eWay Sample Projects" on page 66**
- **prjWebLogic_Sample_JCD** – demonstrates how to deploy and eGate component using java collaborations. For more information, see **"Using the Sample Projects in eGate" on page 77**.

5.1.2 Asynchronous Communication

Asynchronous communication is considered a buffered process, since the sender never waits after sending data and the receiver only waits when the buffer is empty. The buffer or queue is a service that temporarily holds messages until the receiver is ready to process them. This can be comparable to a mail message in which mail is sent and forgotten until sometime later when a response is received.

Figure 9 Asynchronous Communication



Asynchronous Communication in eGate Includes:

- **WebLogic EJB to eGate (JMS) Transactions** – an inbound transaction, where the JMS dictates how client applications talk to a Queue, and the WebLogic EJBs publish to the eGate JMS IQ Manager. For more information, see **"Asynchronous Communication in eGate" on page 34**.
- **eGate (JMS) to WebLogic Message Driven Bean Transactions** – an outbound transaction, where the eGate JMS publishes to a WebLogic Application Server Message Driven Bean. A Message Driven Bean (MDB) is a specialized EJB that acts like a trigger which executes whenever there is activity on a specific Queue. A message published to eGate's JMS causes an MDB stored in WebLogic to execute. For more information, see **"Asynchronous Communication in eGate" on page 34**.

Associated Sample Projects:

Six sub projects are included in the WebLogic eWay **WebLogicJMS.zip** file to demonstrate Asynchronous message interactions.

- **JMSQueueRequestor** – an inbound example that demonstrates how a remote client requests and receives messages asynchronously from a JMS queue.
- **JMSQueueSend** – an outbound example that demonstrates how to pass messages into a JMS queue asynchronously, before ultimately passing into a WebLogic container.
- **JMSTopicPublish** – an outbound example that demonstrates how messages are read, subscribed and published to a JMS topic asynchronously, before passing into a WebLogic container.
- **JMSTopicSubscribe** – an inbound example that demonstrates how a remote client is used to send a messages to eGate asynchronously through a JMS topic.
- **JMSXAQueueSend** – an outbound example that demonstrates how to asynchronously pass two-phase commit protocol (XA) messages into a JMS queue, before ultimately passing into a WebLogic container.
- **JMSXATopicSubscribe** – an inbound example that demonstrates how a remote client is used to asynchronously pass two-phase commit protocol (XA) messages into a JMS topic.

5.2 Synchronous Communication in eGate

Synchronous communication is carried out by the WebLogic eWay, and requires the creation of an OTD using the WebLogic OTD Wizard. WebLogic OTDs are created using WebLogic's Session and Entity Beans (not Message Driven Beans) EJB interface classes, that represent the methods of the EJB.

Once created, these methods are called from within a Collaboration, making them accessible to the user. The OTD queries the JNDI directory services and locates a home interface, uses the home interface to acquire remote interfaces, applies Iterator methods for managing multiple remote interface instances, and provides access to the remote interface methods. Collaborations can then be built between the OTD and OTDs for other applications, making the EJB methods available to that application.

5.2.1 The WebLogic OTD

The WebLogic OTD contains EJB methods that are callable from inside a Collaboration.

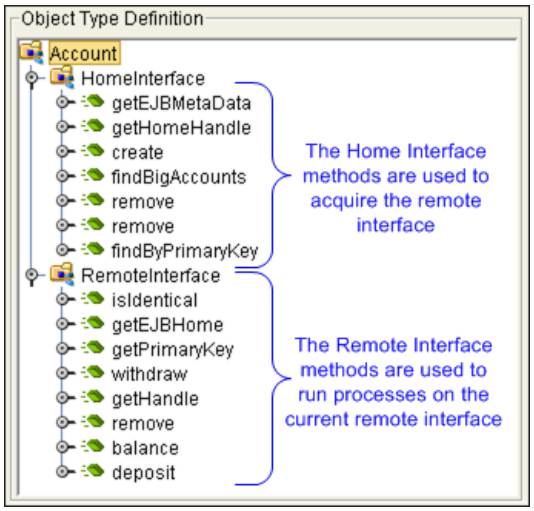
The OTD is divided into two portions:

- **Home Interface Methods** – used to acquire the Remote Interface, allowing OTDs to find and invoke EJB instances.

  As an example, the home interface method **findBigAccounts()**, seen in Figure 10, could use the argument "balanceGreaterThan (100,000)" to find all account EJBs with a balance over 100,000 and assign their remote interface to the Remote Instances OTD node.

- **Remote Interface Methods** – contains remote interface methods that allow processes to be run on the current remote interface.

Figure 10 EJB OTD nodes represent both Home and Remote Interface methods



5.3 Asynchronous Communication in eGate

The following section describes how Sun Microsystems' implementation of JMS applies to asynchronous interaction between the eWay Adapter for WebLogic and WebLogic Server.

The eWay incorporates the Sun Microsystems JMS IQ Manager into the WebLogic environment, allowing EJBs in the WebLogic container to receive messages from or send messages to eGate.

Two messaging procedures are used to facilitate interaction:

- **Message Driven Beans subscribing to Sun Microsystems JMS**
- **Session Beans publishing/sending to Sun Microsystems JMS**

Note: *Installation of the eGate API Kit is required for asynchronous communication in eGate.*

5.3.1 Additional Messaging Service Requirements

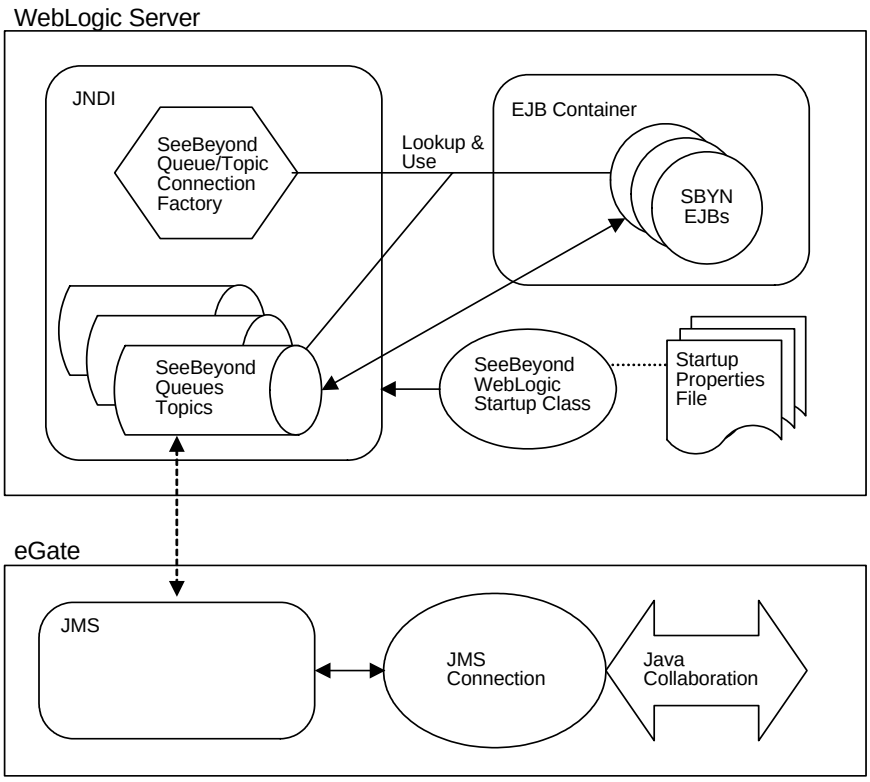
Other WebLogic subsystems required to facilitate messaging services include:

- **EJB Containers** – contains and provides persistence, distributed objects, concurrency, security, and transactions to all EJBs.
- **Naming Services** – required to locate distributed objects, the Java Naming and Directory Interface™ (JNDI) enables servers to host objects at specific times. The naming service allows you to "bind" the following Sun Microsystems JMS objects:
  - **TopicConnectionFactory**
  - **QueueConnectionFactory**
  - **Topic**(s)
  - **Queue**(s)

  By binding instances of these objects, any EJB can get a hold of the references to these objects by looking them up in the naming service using JNDI. The Message Driven Beans (MDBs) are used for asynchronous subscription of messages from a JMS Topic or Queue. This scenario corresponds to the Sun Microsystems JMS provider driving MDBs running in WebLogic. Session Beans are used for publishing and sending Topic/Queue messages through the Sun Microsystems JMS provider as well.

  The following architectural diagram (Figure 11) illustrates the components involved:

Figure 11 WebLogic Server and WebLogic eWay Components



5.3.2 Sun Microsystems JMS

As part of the WebLogic eWay installation, Sun Microsystems supplies startup classes for JMS objects to install into the naming service. Four JMS ConnectionFactory objects are bound to the naming service, including:

- **MyTopicConnectionFactory**
- **XATopicConnectionFactory**
- **MyQueueConnectionFactory**
- **XAQueueConnectionFactory**

Moreover, installing the Sun Microsystems supplied Session Beans and Message Driven Beans installs Topic and Queue objects into the naming service.

Message Flow from eGate to WebLogic Using JMS Objects

To enable message flow from eGate to WebLogic, WebLogic uses the Sun Microsystems**TopicConnectionFactory** to create the necessary JMS TopicConnection(s) and TopicSession(s) and uses the Sun Microsystems **QueueConnectionFactory** to create the JMS QueueConnection(s) and QueueSession(s). Likewise, **XATopicConnectionFactory** is used to create the necessary JMS XATopicConnection(s) and XATopicSession(s) and the Sun Microsystems **XAQueueConnectionFactory** is

used to create the JMS XAQueueConnection(s) and XAQueueSession(s). The **weblogic-ejb-jar.xml** deployment descriptor allows the configuration of Sun Microsystems JMS as a foreign JMS to which the MDBs subscribe. The diagram in Figure 12 shows the components involved in eGate to WebLogic mode. The arrows represent message flow.

Figure 12 Message Flow from eGate to WebLogic

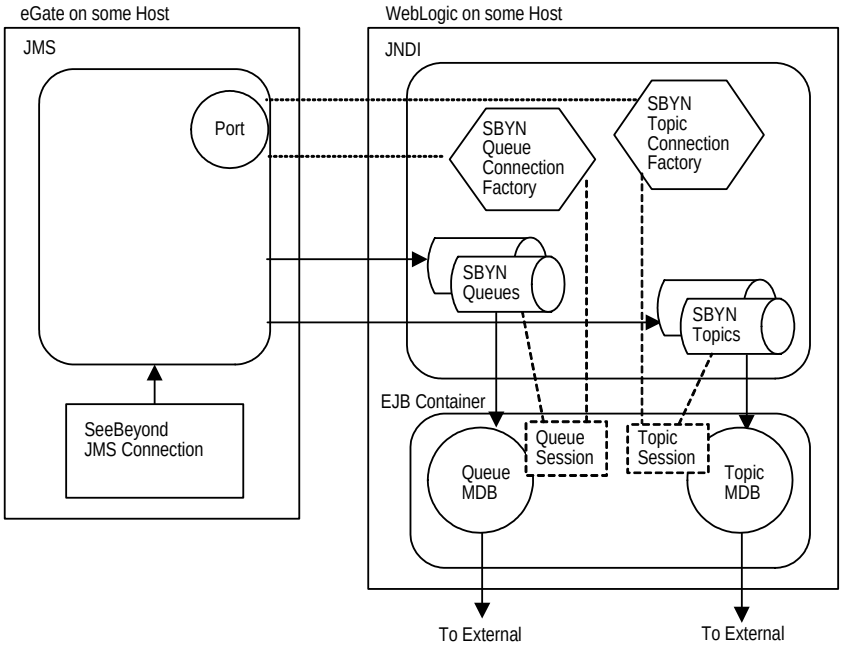
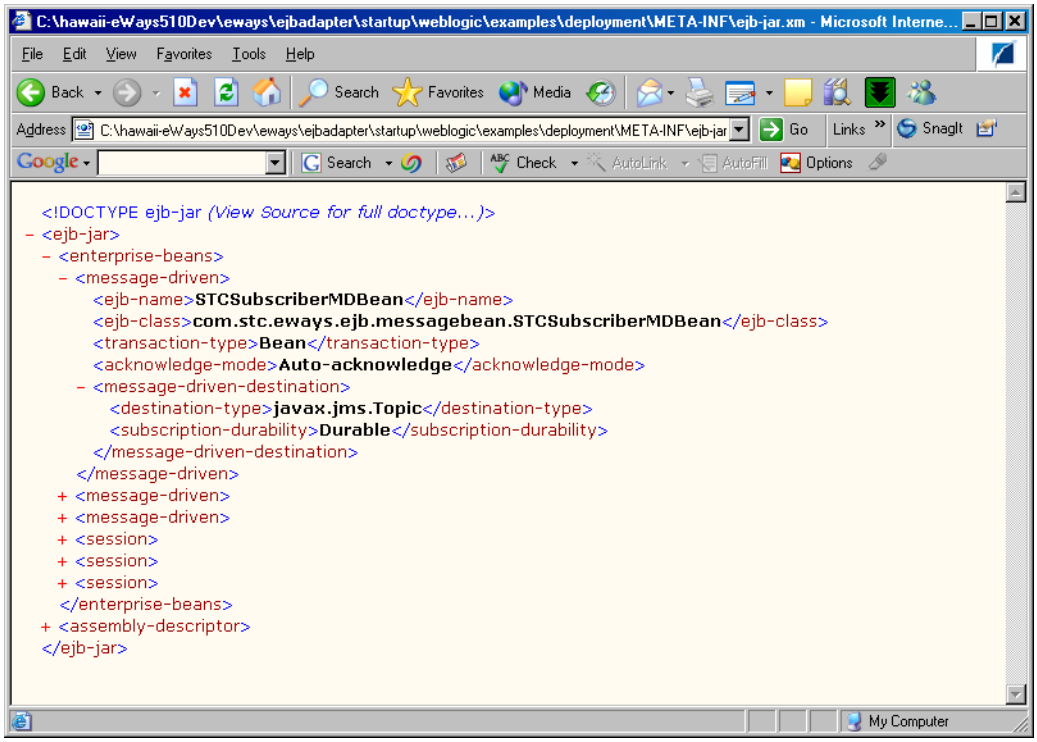


Figure 13 displays an example of the **ejb-jar.xml** for the Topic MDB which receives messages from a Sun Microsystems JMS Topic.

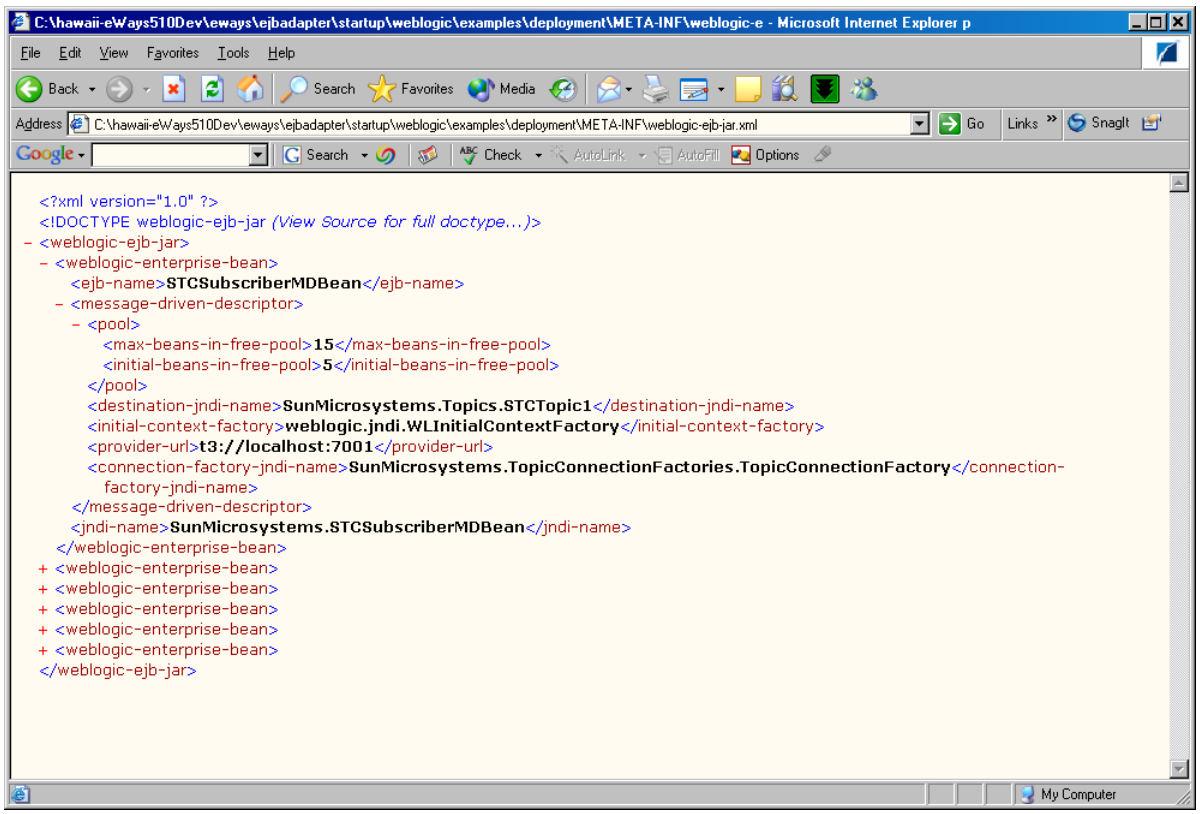
Figure 13 ejb-jar.xml - Topic MDB



Updating the WebLogic JMS

An updated WebLogic JMS is required to ensure communication between eGate and WebLogic. Figure 14 displays an example of the **weblogic-ejb-jar.xml** for the Topic MDB which receives messages from a Sun Microsystems JMS Topic.

Figure 14 weblogic-ejb-jar.xml - Topic MDB



In the above figure, the <destination-jndi-name> tag of the Topic is **SunMicrosystems.Topics.STCTopic1**; this is a Sun Microsystems JMS Topic. Using the WebLogic naming service, the two entries *initial-context-factory* and *provider-url* are **weblogic.jndi.WLInitialContextFactory** and **t3://localhost:7001** respectively. Since the container needs to use the Sun Microsystems **JMS TopicConnectionFactory**, we specify the Sun Microsystems **TopicConnectionFactory** with the <connection-factory-jndi-name> tag as **SunMicrosystems.TopicConnectionFactories.TopicConnectionFactory**. The JNDI bound objects **SunMicrosystems.Topics.STCTopic1** and **SunMicrosystems.TopicConnectionFactories.TopicConnectionFactory** must be created and bound to the WebLogic JNDI for this server instance before deploying and using the MDB. The WebLogic Administrative Console (for version 7.1 and earlier) does NOT allow creation of any foreign JMS objects. This must be done outside of the Administrative Console.

The task of creating the Sun Microsystems JMS objects is done by the Sun Microsystems WebLogic startup class called **WeblogicStartup**. (See the section **"Sun Microsystems WebLogic Startup Class" on page 43** to see how the startup class works and how to configure and deploy it.) The three tag entries <initial-context-factory>, <provider-url>, and <connection-factory-jndi-name> are necessary because Sun Microsystems JMS is being used as a foreign JMS into WebLogic.

The same entries can be added for subscribing to a Sun Microsystems Queue (using the Sun Microsystems **QueueConnectionFactory** as the connection factory and Sun Microsystems Queue as the destination).

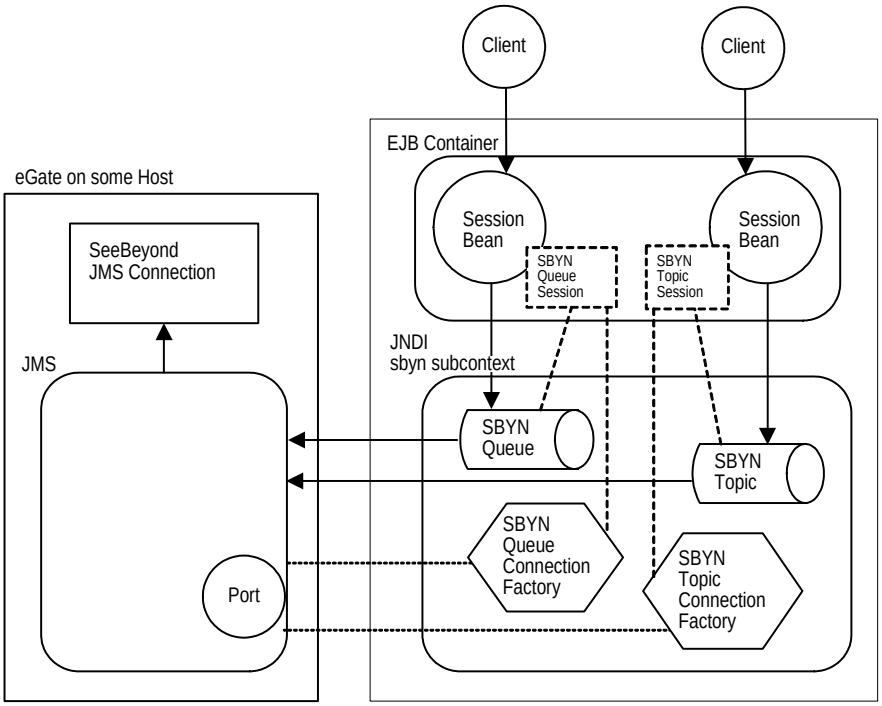
Message Flow from WebLogic to eGate Using JMS Objects

For message flow from WebLogic to eGate, Session Beans can publish/send JMS messages to Sun Microsystems JMS Topics/Queues.

In addition to the connection factories, the Topic and Queue destinations are also bound to the naming service before they are referenced by the Session Beans. Creating these Sun Microsystems JMS objects and JNDI bindings is done through the Sun Microsystems WebLogic startup class, **WeblogicStartup**. (See **"Sun Microsystems WebLogic Startup Class" on page 43** for details.) With access to these JMS objects via JNDI, the Session Beans use the JMS API's to send the JMS message to eGate.

Figure 15 displays a diagram of the components involved for the WebLogic to eGate mode. The arrows represent the message flow.

Figure 15 Message Flow from WebLogic to eGate



Every bean automatically has access to a special naming system called the **Environment Naming Context** (ENC). The ENC is managed by the container and accessed by beans using JNDI. The JNDI ENC allows a bean to access resources like JDBC connections, other enterprise beans, and properties specific to that bean. Each Session Bean uses the ENC to specify the **TopicConnectionFactory** or **QueueConnectonFactory** with the <resource-ref> element in the **ejb-jar.xml** file.

Additionally, the Session Bean uses the ENC to specify the destination via the <resource-env-ref> element in the **ejb-jar.xml**. The **weblogic-ejb-jar.xml** also has these corresponding elements defined with the <resource-description> and <resource-env-description> elements.

Figure 16 displays the Session Bean **ejb-jar.xml** deployment descriptor.

Figure 16 Session Bean ejb-jar.xml deployment descriptor

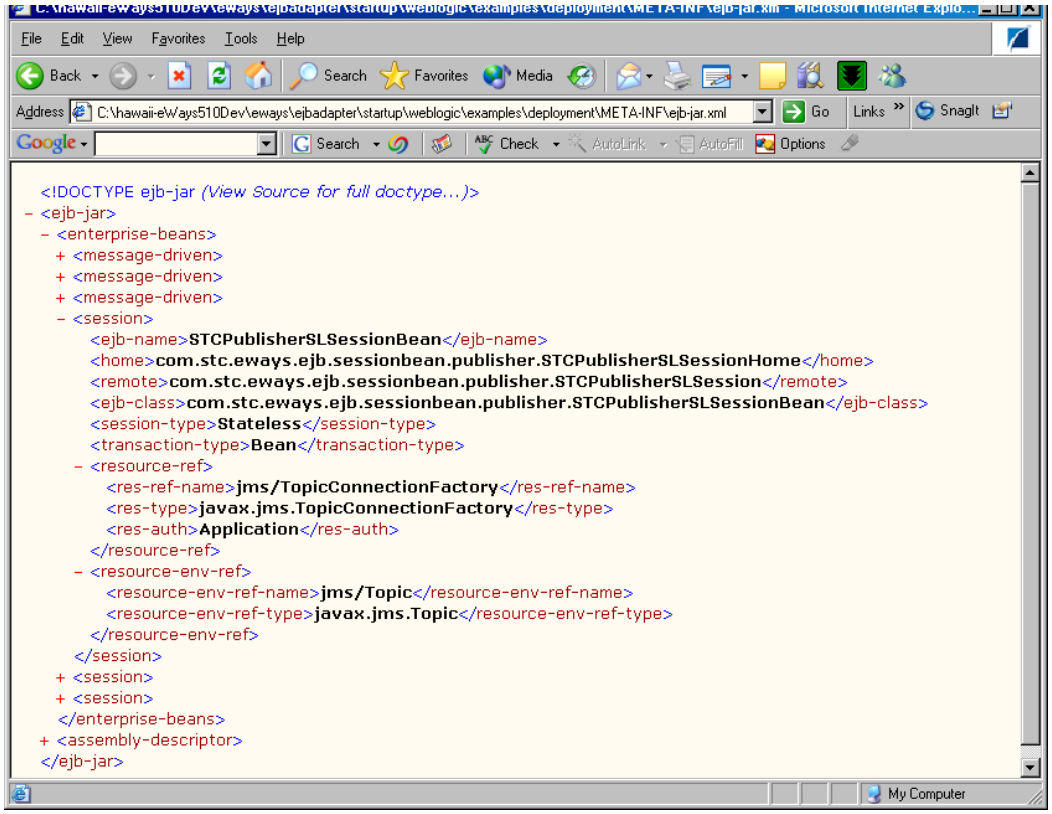


Figure 17 displays the Session Bean **weblogic-ejb-jar.xml** deployment descriptor.

Figure 17 Session Bean weblogic-ejb-jar.xml deployment descriptor

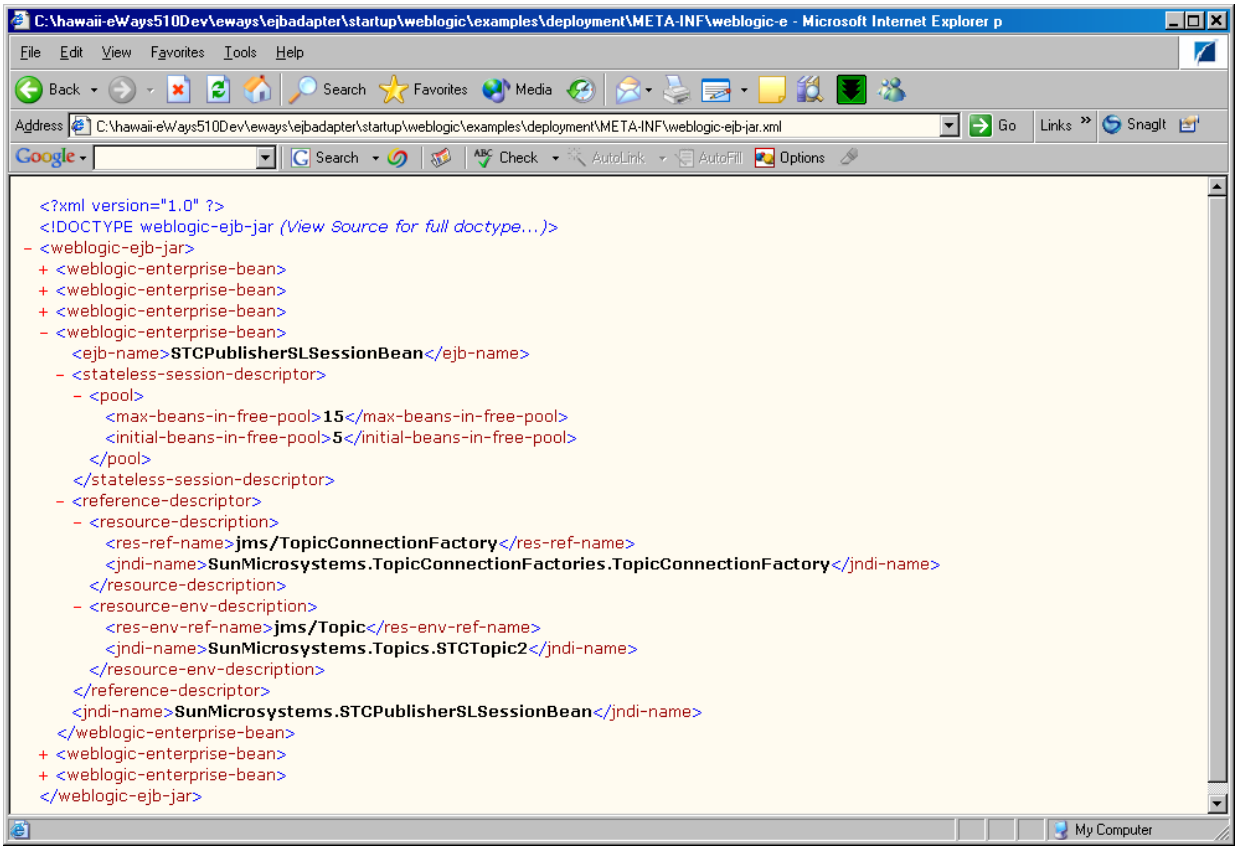
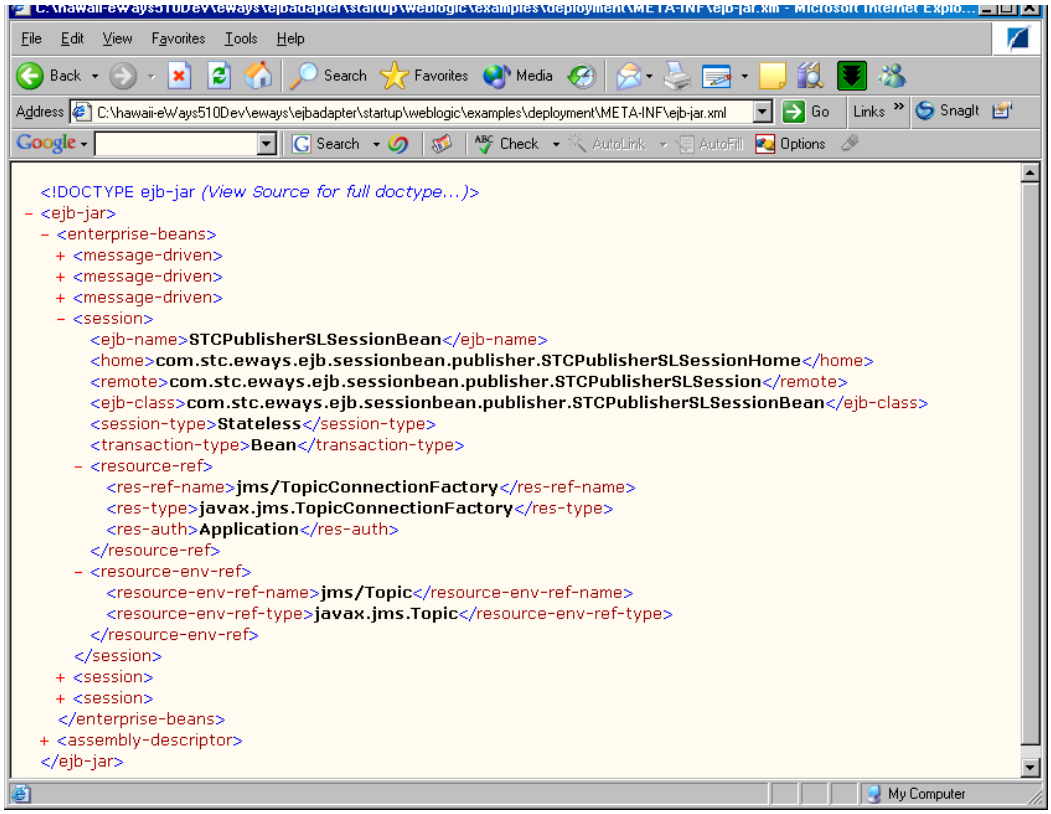


Figure 18 displays an example of the **ejb-jar.xml** deployment descriptor for the Session Bean publishing to a Sun Microsystems JMS Topic:

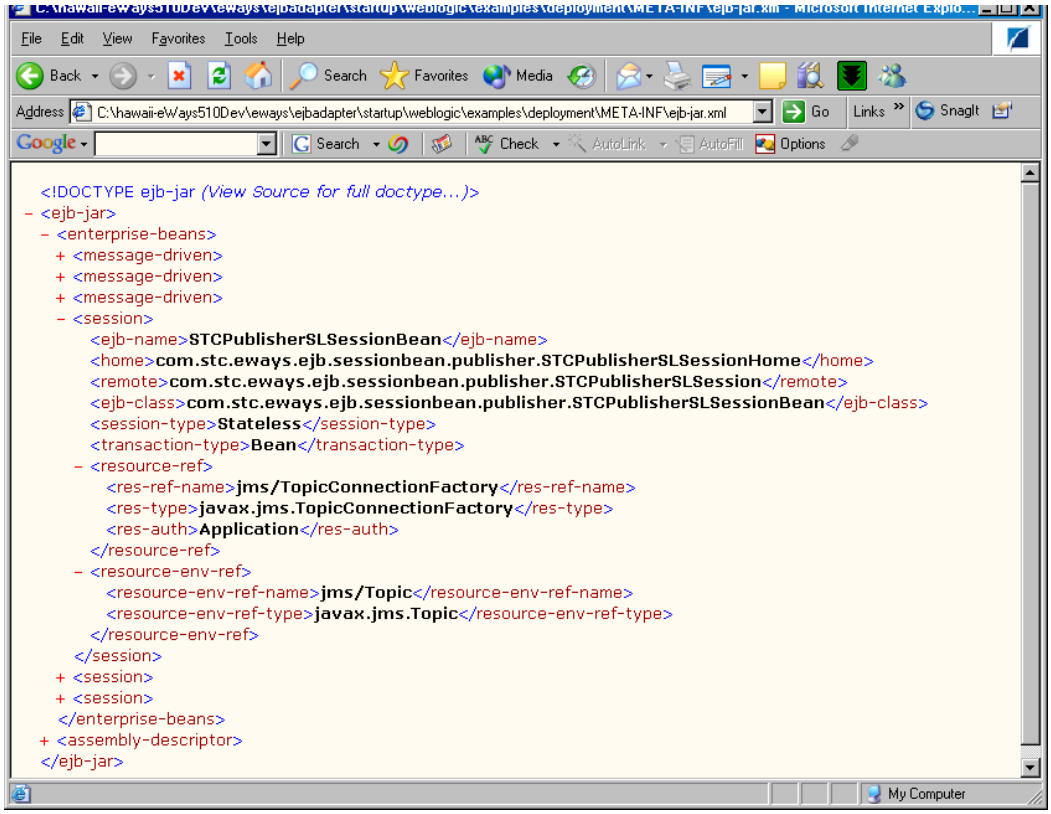
Figure 18 ejb-jar.xml deployment descriptor - Session Bean to Sun Microsystems JMS Topic



The value for the <res-ref-name> tag is **jms/TopicConnectionFactory** and the value for the <resource-env-ref-name> environment entry tag is **jms/Topic**. They are specified as **javax.jms.TopicConnectionFactory** and **javax.jms.Topic** for the resource type respectively. These resource references are another level of JNDI indirection. They don't specify the actual JNDI names of the JMS objects, but instead reference the JNDI name. Additionally, the EJB can reference **jms/TopicConnectionFactory** but does not really care what the actual JNDI name is. The actual JNDI names for these references are defined in the **weblogic-ejb-jar.xml** file.

The **weblogic-ejb-jar.xml** defines the actual JNDI name of the resource references defined in **ejb-jar.xml** for the Session Bean as seen in Figure 19.

Figure 19 weblogic-ejb-jar.xml defines the actual JNDI name



The value for the jndi-name tag for the resource name **jms/TopicConnectionFactory** is **SunMicrosystems.TopicConnectionFactories.TopicConnectionFactory** and the value for the **jndi-name** tag for the **jms/Topic** entry is **SunMicrosystems.Topics.STCTopic2**. These define the resource reference name to JNDI name mappings. As mentioned earlier, these JNDI bound objects need to be created by the startup class.

5.4 Sun Microsystems WebLogic Startup Class

To bind the Sun Microsystems JMS objects into the WebLogic naming service, a Sun Microsystems startup class is installed on the WebLogic Server. The startup class is loaded by the WebLogic Server when the server is booted and the startup method of the class is invoked.

Upon invocation of the startup method, the following objects are instantiated and bound to WebLogic's naming service:

- **A Sun Microsystems MyTopicConnectionFactory**
- **A Sun Microsystems MyQueueConnectionFactory**
- **All Configured Topics**
- **All Configured JMS Queues**

The configuration file for the startup class is in the form of a Java properties file. Before describing the format of this file, let's look at the implementation of the startup class.

5.4.1 Startup Class Implementation

The startup class is called **WeblogicStartup.class**. It implements the **weblogic.common.T3StartupDef interface**. The **WeblogicStartup.class** only needs to implement two methods:

- **setServices()**
- **startup()**

setServices() method

The **setServices()** method is trivial; the server passes in an instance of **T3ServicesDef** which can be saved by the startup class as an attribute. (See the WebLogic documentation on **T3ServicesDef** for more information on this interface.)

startup() method

The **startup()** method is where the crux of the work is done. This method is invoked by the server and this is where the Sun Microsystems JMS objects are created and bound to the naming service.

The **startup()** method takes two parameters that are provided by the server:

- **name** – which is of type **java.lang.String**, is the name of the startup class.
- **args** – which is of type **HashTable,** contains name/value pairs that are passed to the startup as program "arguments."

Both the **name** and **args** program arguments are defined when the startup class is deployed in the server using the WebLogic Administrative Console.

5.4.2 Startup Properties File

The startup properties file, **weblogic.startup.properties,** is read by the startup class when the **startup()** method is invoked by the WebLogic Server and is used to configure information about the Sun Microsystems JMS specific information.

This file consists of name/value pairs. There are seven sections to this properties file. Each name and value in the different sections have different meanings. Each section of the default **weblogic.startup.properties** file in detail. Comment lines in the properties file start with either a '#' or a '!' character.

Any changes to the startup configuration (properties) file does not take effect right away. The WebLogic Server must be restarted in order for the startup class to get reloaded and for the startup class to read the changes to the configuration file. For example, if a new Topic or Queue is added, the WebLogic Server needs to be restarted.

weblogic.startup.properties File

Sun Microsystems JNDI Sub-context

The first section allows the user to specify the JNDI sub-context for Sun Microsystems.

```
#-------------------------------------------------------------------
#
# JNDI subcontext for Sun Microsystems objects.
# This section configures the JNDI subcontext to which all the Sun
# Microsystems JMS objects will bind.
#
# WARNING : Only the property value can be changed here.
#-------------------------------------------------------------------
Subcontext.SunMicrosystems=SunMicrosystems
```

The user should not have to change this.

Sun Microsystems JMS TopicConnectionFactory Sub-context

The next section allows the user to specify the JNDI sub-context where all instances of Sun Microsystems **JMS TopicConnectionFactory** are bound. This sub-context is under the Sun Microsystems sub-context.

```
#-------------------------------------------------------------------
#
# JNDI subcontext for Sun Microsystems JMS Topic connection
# factories. This section configures the JNDI subcontext to which all
# the Sun Microsystems JMS TopicConnectionFactory objects will bind.
#
# WARNING : Only the property value can be changed here.
#-------------------------------------------------------------------
Subcontext.TopicConnectionFactory=TopicConnectionFactories
```

The user should not have to change this.

SunMicrosystems JMS QueueConnectionFactory Sub-context

The next section allows the user to specify the JNDI sub-context where all instances of Sun Microsystems **JMS QueueConnectionFactory** are bound. This sub-context is under the Sun Microsystems sub-context configured.

```
#-------------------------------------------------------------------
#
# JNDI subcontext for Sun Microsystems JMS Queue connection
# factories. This section configures the JNDI subcontext to which all
# the Sun Microsystems JMS QueueConnectionFactory objects will bind.
#
# WARNING : Only the property value can be changed here.
#-------------------------------------------------------------------
Subcontext.QueueConnectionFactory=QueueConnectionFactories
```

The user should not have to change this.

SunMicrosystems JMS Topic Sub-context

The next section allows the user to specify the JNDI sub-context where all instances of Sun Microsystems JMS Topic destinations are bound. This sub-context is under the Sun Microsystems sub-context configured.

```
#-------------------------------------------------------------------
#
# JNDI subcontext for SunMicrosystems JMS Topics.
# This section configures the JNDI subcontext to which all the Sun
# Microsystems JMS Topic objects will bind.
#
# WARNING : Only the property value can be changed here.
#-------------------------------------------------------------------
Subcontext.Topic=Topics
```

The user should not have to change this.

SunMicrosystemsJMS Queue Sub-context

The next section allows the user to specify the JNDI sub-context where all instances of Sun MicrosystemsJMS Queue destinations are bound. This sub-context is under the Sun Microsystems sub-context configured.

```
#-------------------------------------------------------------------
#
# JNDI subcontext for Sun Microsystems JMS Queues.
# This section configures the JNDI subcontext to which all the Sun
# Microsystems JMS Queues objects will bind.
#
# WARNING : Only the property value can be changed here.
#-------------------------------------------------------------------
Subcontext.Queue=Queues
```

The user should not have to change this.

Sun Microsystems JMS Server Names List

The next section allows the user to specify the logical names of each JMS server instances to configure for registration to WebLogic JNDI:

```
#-------------------------------------------------------------------
#
# JMS Server Names
# Define all the logical JMS Server Names in this section.
# Each Server Name must be separated by a '&' character.
#
# WARNING: Only the property value can be changed here.
# Example: SunMicrosystemsJMS&MyJMS&JMSOnHostA
#-------------------------------------------------------------------

#JMSServerNames=SunMicrosystemsJMS&MyJMS
JMSServerNames=SunMicrosystemsJMS
```

The server names are separated by the '&' character. The server names used here are referenced in another section for configuring the JMS host, port, and the connection factories.

Sun Microsystems JMS Servers Configuration

For each server name listed in the **JMSServerNames** property value, the user is required to specify the hostname and port of the JMS server. In addition, the user can configure one or more of the types of JMS connection factories (TopicConnectionFactory, QueueConnectionFactory, and so forth.).

```
#-------------------------------------------------------------------
#
# JMS Servers Configuration
# For each of the Servers define in the JMS Server Names section,
# define the JMS configurations in this section.
# The following JMS information must be defined for each Server:
#   Host, Port
# The following JMS information is optional for each Server:
#   PingTimeout
# The following are used to configure JMS Connection Factories:
#   TopicConnectionFactory, QueueConnectionFactory
#   XATopicConnectionFactory, XAQueueConnectionFactory
#
#-------------------------------------------------------------------

! SunMicrosystemsJMS Server configuration
! Notice that "SunMicrosystemsJMS" is in the JMS Server Names list.
```

```
SunMicrosystemsJMS.Host=localhost
SunMicrosystemsJMS.Port=18007
SunMicrosystemsJMS.TopicConnectionFactory=TopicConnectionFactory
SunMicrosystemsJMS.QueueConnectionFactory=QueueConnectionFactory
SunMicrosystemsJMS.XATopicConnectionFactory=XATopicConnectionFactory
SunMicrosystemsJMS.XAQueueConnectionFactory=XAQueueConnectionFactory
#SunMicrosystemsJMS.PingTimeout=10


! MyJMS Server configuration
! Notice that "MyJMS" is in the JMS Server Names list.
#MyJMS.Host=localhost
#MyJMS.Port=9876
#MyJMS.TopicConnectionFactory=MyTopicConnectionFactory
#MyJMS.QueueConnectionFactory=MyQueueConnectionFactory
#MyJMS.PingTimeout=10
```

The optional property **PingTimeOut** is added to the sample property file. If the socket is inactive for longer than the set PingTimeOut (in milliseconds), the socket will close and the client can detect that the connection is lost.

Note: *The sample above demonstrates how two JMS server instances are configured on two different ports.*

There are four possible connection factories that can be configured:

- **TopicConnectionFactory**
- **QueueConnectionFactory**
- **XATopicConnectionFactory**
- **XAQueueConnectionFactory**

For the connection factories, the property value is used as the JNDI name of the factory object created. In the example above, we are telling the startup to create a **TopicConnectionFactory** with **SunMicrosystems.TopicConnectionFactories.TopicConnectionFactory** as the JNDI name for the **TopicConnectionFactory**. Notice that the Sun Microsystems sub-context and the **TopicConnectionFactories** sub-context are pre-pended.

Sun Microsystems JMS Topic Destinations

The next section allows the user to specify the Topics to create and bind to JNDI:

```
#--------------------------------------------------------------------
#
# Sun Microsystems JMS Topics
# This section configures the Sun Microsystems JMS Topics.
# The property name for each Topic entry must start with "Topic.".
# For each Topic entry, the property name will be used as the JMS
# Topic name and the property value will be used as the JNDI name for
# the Topic.
#
#--------------------------------------------------------------------

! A sample JMS Topic with name "Topic.Sample1" and JNDI name
"STCTopic1"
Topic.Sample1=STCTopic1
```

```
! Another sample JMS Topic with name "Topic.Sample2" and JNDI name
"STCTopic2"
Topic.Sample2=STCTopic2

! Another sample JMS Topic with name "Topic.Sample3" and JNDI name
"STCTopic3"
Topic.Sample3=STCTopic3
```

For each Topic to configure, the property name must start with "Topic". The startup class uses the property name as the Topic name when creating the Sun Microsystems Topic. This Topic name is the name to be used in the eGate environment (the name of the event created with the Enterprise Manager). The property value for the Topic is used as the JNDI name for the Topic. The JNDI name is used by the EJB (via the EJB's deployment descriptor). See the section **"Message Flow from eGate to WebLogic Using JMS Objects" on page 35** and **"Message Flow from WebLogic to eGate Using JMS Objects" on page 39** for more information on the EJB deployment descriptors.

Sun Microsystems JMS Queue Destinations

The next section allows the user to specify the Queues to create and bind to JNDI:

```
#-------------------------------------------------------------------
#
# Sun Microsystems JMS Queues
# This section configures the Sun Microsystems JMS Queues.
# The property name for each Queue entry must start with "Queue.".
# For each Topic entry, the property name will be used as the JMS
# Queue name and the property value will be used as the JNDI name for
# the Queue.
#
#-------------------------------------------------------------------

! A sample JMS Queue with name "Queue.Sample1" and JNDI name
"STCQueue1"
Queue.Sample1=STCQueue1

! Another sample JMS Queue with name "Queue.Sample2" and JNDI name
"STCQueue2"
Queue.Sample2=STCQueue2

! Another sample JMS Queue with name "Queue.Sample3" and JNDI name
"STCQueue3"
Queue.Sample3=STCQueue3
```

For each Queue to configure, the property name must start with "Queue". The startup class uses the property name as the Queue name when creating the Sun Microsystems Queue. This Queue name is the name to be used in the eGate environment (the name of the event created with Enterprise Manager). The property value for the Queue is used as the JNDI name for the Queue. The JNDI name is used by the EJB (via the EJB's deployment descriptor). See the section **"Message Flow from eGate to WebLogic Using JMS Objects" on page 35** and **"Message Flow from WebLogic to eGate Using JMS Objects" on page 39** for more information on the EJB deployment descriptors.

Chapter 6

Configuring WebLogic Server for Asynchronous Communication

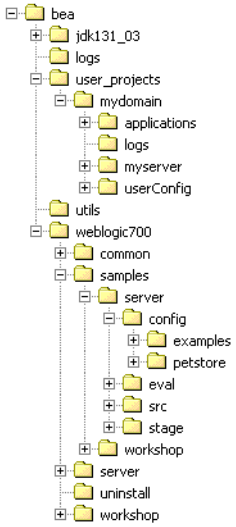
The following chapter provides directions for configuring WebLogic Server for asynchronous interaction with eGate. Setup directions are provided for both WebLogic versions 7.0 and 8.1.

What's in This Chapter:

6.1 Configuration for WebLogic 7.0

WebLogic Server 7.0 installation creates a home or root directory named "**bea**" by default (this name may be changed during installation). Sample servers are located in the **<BEA-HOME>\weblogic700\samples\server\config** directory. Servers created by the user are located under **<BEA-HOME>\user_projects\<domain name>** (see Figure 20).

Figure 20 WebLogic Server File Structure



1. Verify that the system classpath contains ejb.jar, weblogic.jar (with ejb.jar proceeding weblogic.jar in order), and weblogic.ejb.example.jar.
2. Copy the following files to the <BEA-HOME>\weblogic700\server\lib directory.
   - com.stc.jms.stcjms.jar
   - weblogic.ejb.example.jar
   - weblogic.startup.jar
   - weblogic.startup.properties

   com.stc.jms.stcjms.jar can be found in the Java CAPS Repository at:

   ```
   repository\data\files\InstallManager\50Base\stcas\common\lib\com.stc.jms.stcjms.jar
   ```

   The required JAR files can be downloaded from the link **WebLogic eWay - Runtime Asynchronous Communication samples** under the **Downloads** tab in the Sun Java Composite Application Platform Suite Installer.
3. Copy the JSSE.jar file from: `repository\jre\1.4.2\lib` and place into `bea\weblogic700\server\lib`.
4. Modify **startExamplesServer.cmd** and **setExamplesServer.cmd** located at <BEA-HOME>\user_projects\<domain name>, appending com.stc.jms.stcjms.jar and weblogic.startup.jar to the classpath for each. For example:

   For startExamplesServer.cmd

   ```
   CLASSPATH=C:\bea\jdk131_03\lib\tools.jar;%POINTBASE_HOME%\lib\pbserver42ECF183.jar;%POINTBASE_HOME%\lib\pbclient42ECF183.jar;%CLIENT_CLASSES%;%SERVER_CLASSES%;%COMMON_CLASSES%;%CLIENT_CLASSES%\utils_common.jar;C:\bea\weblogic700\server\lib\com.stc.jms.stcjms.jar;C:\bea\weblogic700\server\lib\weblogic.startup.jar
   ```

   For setExampleEnv.cmd

   ```
   CLASSPATH=%CLIENT_CLASSES%;%SERVER_CLASSES%;%SAMPLES_HOME%\server\eval\pointbase\lib\pbserver42ECF183.jar;%SAMPLES_HOME%\server\eval\pointbase\lib\pbclient42ECF183.jar;%WL_HOME%\server\lib\classes12
   ```

```
.zip;%COMMON_CLASSES%;C:\bea\weblogic700\server\lib\com.stc.jms.st
cjms.jar;C:\bea\weblogic700\server\lib\weblogic.startup.jar
```

5 The sample EJBs have been configured to reference the T3 naming service that is running on the localhost at port 7001. By default, each WebLogic Server instance is installed to listen on port 7001. If your server instance is running, listening on port 7003 for example, then you do not need to modify the deployment descriptors for the EJBs. Otherwise, modify the deployment descriptors by completing the following steps:

   A Extract weblogic.ejb.example.jar to a temporary file and edit **META-INF\weblogic-ejb-jar.xml**.

   B For each Bean that is run, find the Provider_URL tag of the deployment descriptor, change the port number from **7001** to **7003**, and if necessary, change l**ocalhost** to the name of your specific computer.

   C Save, re-jar (zip), and replace weblogic.ejb.example.jar.

6 Start an instance of the application server (in this case, the user defined domain/server).

7 When the server has finished booting, start the Administration Console. Go to Deployments, Startup & Shutdown, and click on **Configure a New Startup Class** (see Figure 21.) Enter the following Values:

   Name: SunMicrosystems_Startup

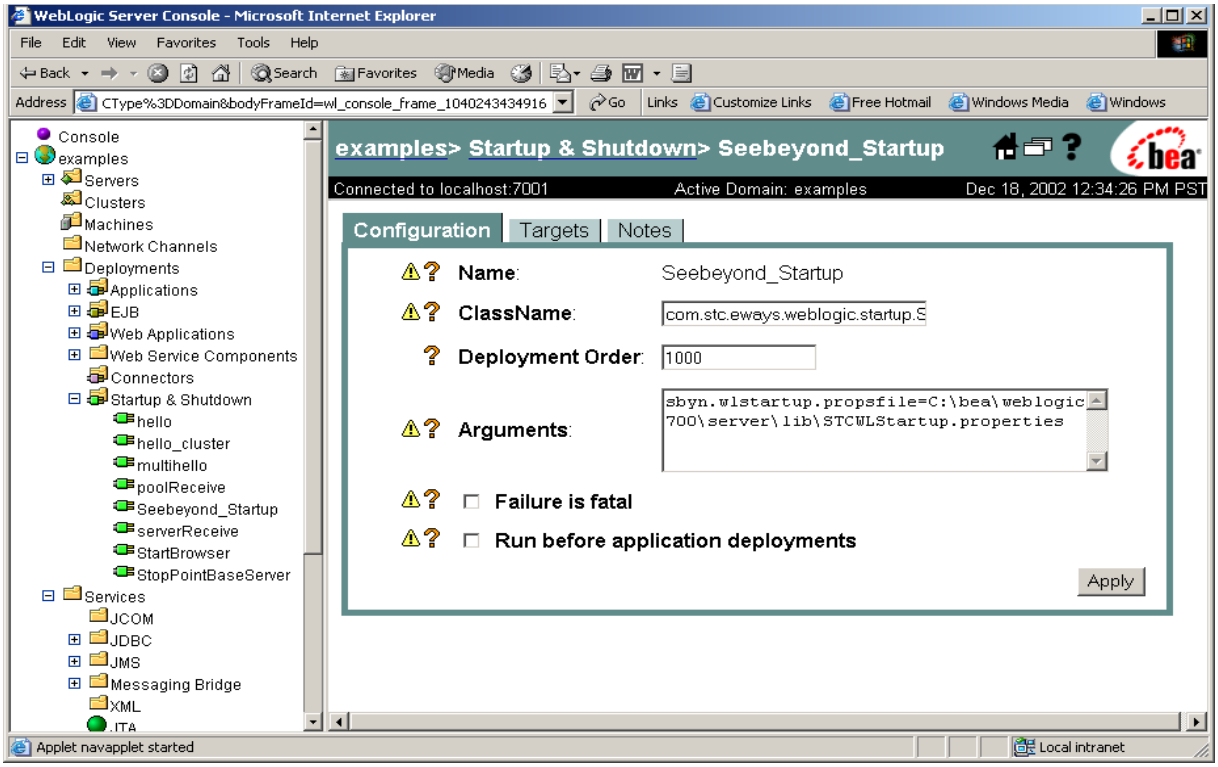
   CLASSNAME: com.stc.eways.weblogic.startup.weblogicstartup

   Deployment Order: 1000 (default)

   Arguments: sbyn.wlstartup.propsfile=<*BEA-HOME*>\weblogic700\server\lib\weblogicstartup
   .properties (where <*BEA-HOME*> is the home directory of the WebLogic Server.)

   Click **Create** and **Apply**.

Figure 21 WebLogic Server Console - Create a New StartupClass



8 Click on the **Targets** tab and move the new server instance from Available to Chosen using the arrow button. Click **Apply**.

9 Stop and restart the server by completing the following steps:

A From the navigator pane on the left, go to <mydomain>, Servers, and right-click on <myserver> (or the new server instance). Click on **Start/stop this server**.

B In the pane on the right, under the Start/Stop tab, click on **Shutdown this server,** then click **Yes**. The server shuts down.

C To restart the server, from the Windows Programs menu, select BEA WebLogic Platform 7.0, User Projects, <mydomain>, Start Server.

D When prompted, enter user name and password.

If the startup class is successfully invoked, you should see the following text in the Start Server command window:

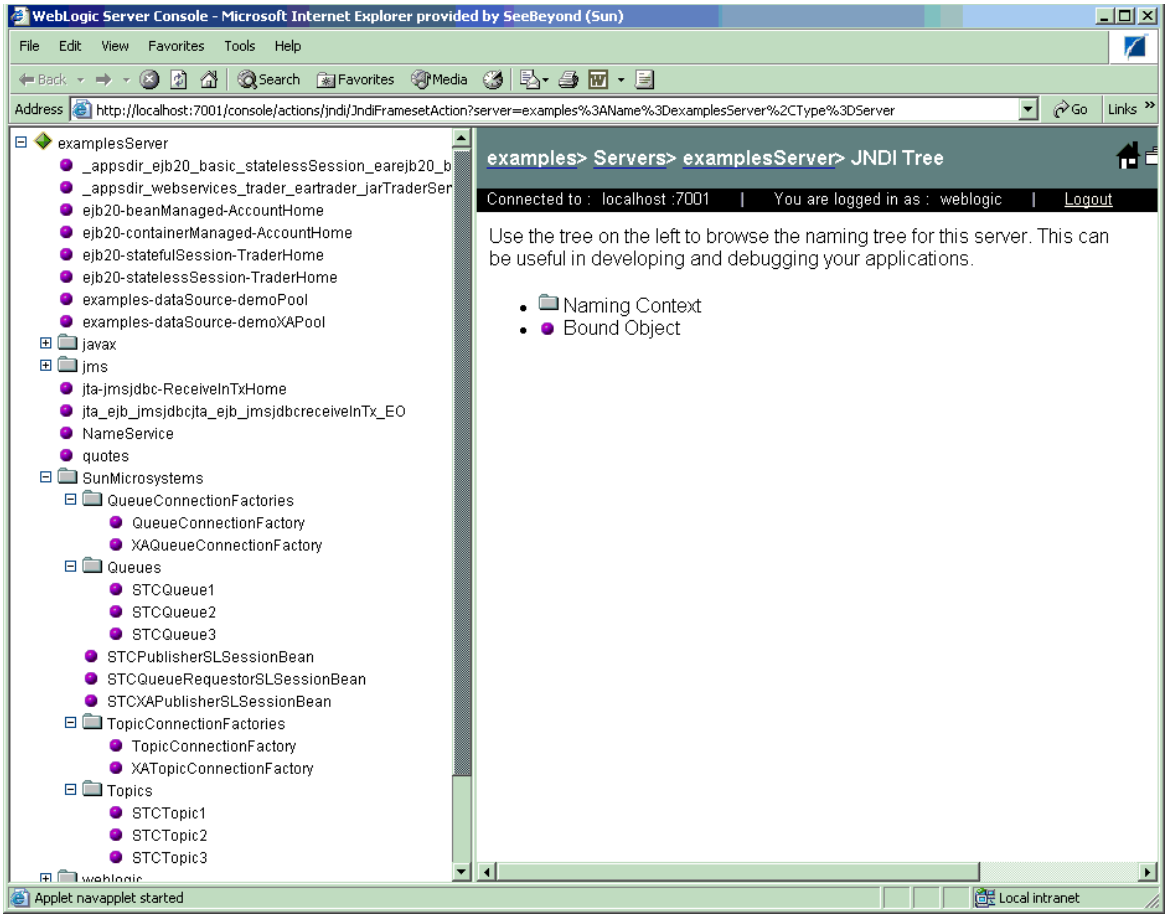
```
weblogicstartup - Sun Microsystems startup class invoked -
SunMicrosystems_Startup
weblogicstartup - Topic name: Topic.Sample3
weblogicstartup - Topic name: Topic.Sample2
weblogicstartup - Topic name: Topic.Sample1
weblogicstartup - Queue name: Queue.Sample3
weblogicstartup - Queue name: Queue.Sample2
weblogicstartup - Queue name: Queue.Sample1
weblogicstartup - Successfully invoked Sun Microsystems startup.
```

10 Start the Administration Console.

11 In the Console, go to Servers, <myserver> (or the new server instance). Right-click exampleServer and select View JNDI Tree to open the JNDI Tree window. Expand

the Sun Microsystems node to verify that all Sun Microsystems JMS objects are now available (see Figure 22).

Figure 22 View the JNDI Tree



12 From the Navigator pane on the left, click on Examples, Deployments, EJB. Click on **Configure a new EJB**.

Note: *Before deploying the EJB, make sure that the JMS IQ Manager is running. It is only necessary to start the JMS IQ Manager*

A Under **Step 1**, click on **upload it through your browser**. Click **Browse** and select <BEA-Home>\weblogic700\server\lib**weblogic.ejb.example.jar**. With the file selected, click **Upload**.

B Under **Step 2**, find **weblogic.ejb.example.jar** and click **select** (left of the name).

C Under **Step 3**, select the server instance under **Available Servers**. Click the **right-arrow** to move the new server instance to **Target Servers**.

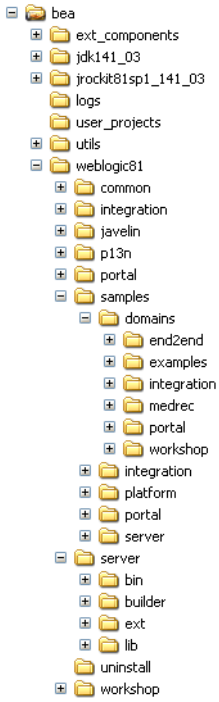
D Under **Step 4**, enter **weblogic.ejb.example** as the name for this application (EJB).

E Under **Step 5**, click the **Configure and Deploy** button. This installs the EJB on the WebLogic Administration Server.

6.2 Configuration for WebLogic 8.1

WebLogic Server 8.1 installation creates a home or root directory named "**bea**" by default (this name may be changed during installation). Sample servers are located in the **<BEA-HOME>\weblogic81\sample\server\lib** directory. Servers created by the user are located under **<BEA-HOME>\user_projects\<domain name>** (see Figure 23).

Figure 23 WebLogic Server File Structure



1. Verify that the system classpath contains ejb.jar, weblogic.jar (with ejb.jar preceding weblogic.jar in order), and weblogic.ejb.example.jar.
2. Copy the following files to the **<BEA-HOME>\weblogic81\server\lib** directory:
   - com.stc.jms.stcjms.jar
   - weblogic.ejb.example.jar
   - weblogic.startup.jar
   - weblogic.startup.properties

com.stc.jms.stcjms.jar can be found in the Java CAPS Repository at:

```
repository\data\files\InstallManager\50Base\stcas\common\lib\com.stc.jms.stcjms.jar
```

The required JAR files can be downloaded from the link **WebLogic eWay - Runtime Asynchronous Communication samples** under the **Downloads** tab in the Sun Java Composite Application Platform Suite Installer.

3. Copy the **stcjms.jar** file to the **<BEA-HOME>\weblogic81\server\lib** directory.

   This file is located in the following directory:

<eGate-Home>\repository\data\files\InstallManager\STCMA\common\lib

4 Modify **startExamplesServer.cmd** and **setExamplesServer.cmd** located at <BEA-HOME>\<user_projects>\<domain name>, appending com.stc.jms.stcjms.jar and weblogic.startup.jar to the classpath for each. For example:

For startExamplesServer.cmd

```
CLASSPATH=C:\bea\weblogic81\server\lib\webservices.jar;%POINTBASE_
CLASSPATH%;%CLIENT_CLASSES%;%SERVER_CLASSES%;%COMMON_CLASSES%;%CLI
ENT_CLASSES%\utils_common.jar;C:\bea\weblogic81\server\lib\com.stc
.jms.stcjms.jar;C:\bea\weblogic81\server\lib\weblogic.startup.jar
```

For setExampleEnv.cmd

```
CLASSPATH=%WL_HOME%\server\lib\webservices.jar;%CLIENT_CLASSES%;%S
ERVER_CLASSES%;%POINTBASE_CLASSPATH%;%POINTBASE_TOOLS%;%COMMON_CLA
SSES%;%CLIENT_CLASSES%\utils_common.jar;%WEBLOGIC_CLASSPATH%;%WL_H
OME%\server\lib\com.stc.jms.stcjms.jar;%WL_HOME%\server\lib\weblog
ic.startup.jar
```

5 The sample EJBs have been configured to reference the T3 naming service that is running on the localhost at port 7001. By default, each WebLogic Server instance is installed to listen on port 7001. If your server instance is running on a different port, then you should modify the deployment descriptors for the EJBs to match this port.

If you need to modify the deployment descriptors, do the following:

A Extract weblogic.ejb.example.jar to a temporary file and edit **META-INF\weblogic-ejb-jar.xml**.

B For each Bean that is run, find the Provider_URL tag of the deployment descriptor, change the port number from the current port number from **7001** to **7003**, and if necessary, change **localhost** to the name of your specific computer.

C Save, re-jar (zip), and replace weblogic.ejb.example.jar.

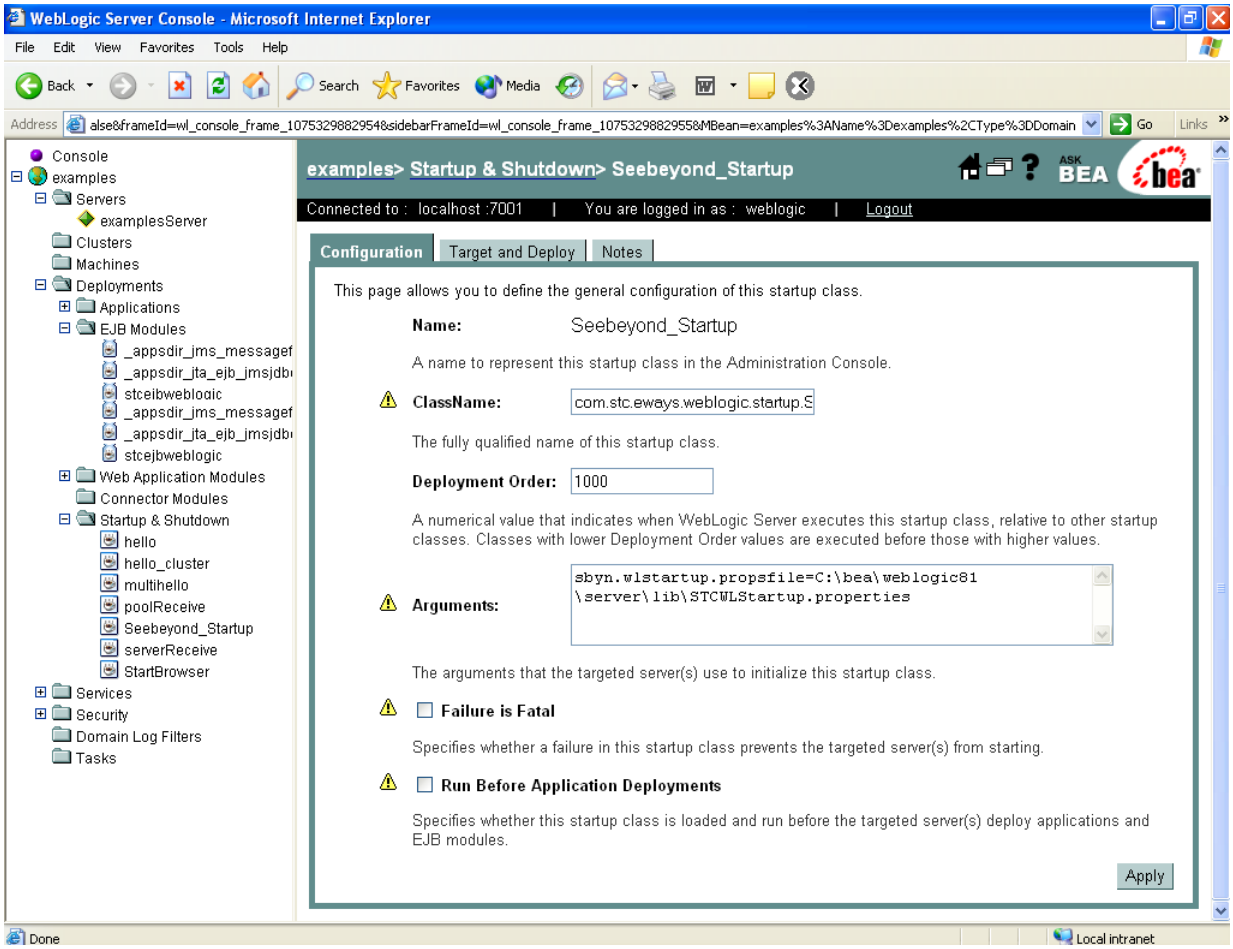
6 Start an instance of the application server (in this case, the user defined domain/server).

7 When the server has finished booting, start the Administration Console. Go to Deployments, Startup & Shutdown, and click on **Configure a New Startup Class** (see Figure 21.) Enter the following Values:

- **Name:** SunMicrosystems_Startup
- **CLASSNAME:** com.stc.eways.weblogic.startup.weblogicstartup
- **Deployment Order**: 1000 (default)
- **Arguments:** sbyn.wlstartup.propsfile=<*BEA-HOME*>\weblogic81\server\lib\weblogicstartup .properties (where <*BEA-HOME*> is the home directory of the WebLogic Server.)

8 Click **Create** and **Apply**.

Figure 24 WebLogic Server Console - Create a New StartupClass



9 Click on the **Target and Deploy** tab and move the new server instance from Available to Chosen using the arrow button. Click **Apply**.

10 Stop and restart the server by completing the following steps:

A From the navigator pane on the left, go to <mydomain>, Servers, and right-click on <myserver> (or the new server instance). Click on **Start/stop this server**.

B In the pane on the right, under the Start/Stop tab, click on **Graceful shutdown of this server** and **Apply**. The server shuts down.

C To restart the server, from the Windows Programs menu, select BEA WebLogic Platform 8.1, Examples, WebLogic Server Examples, Launch WebLogic server Examples.

D When prompted, enter user name and password.

If the startup class is successfully invoked, you should see the following text in the Start Server command window:

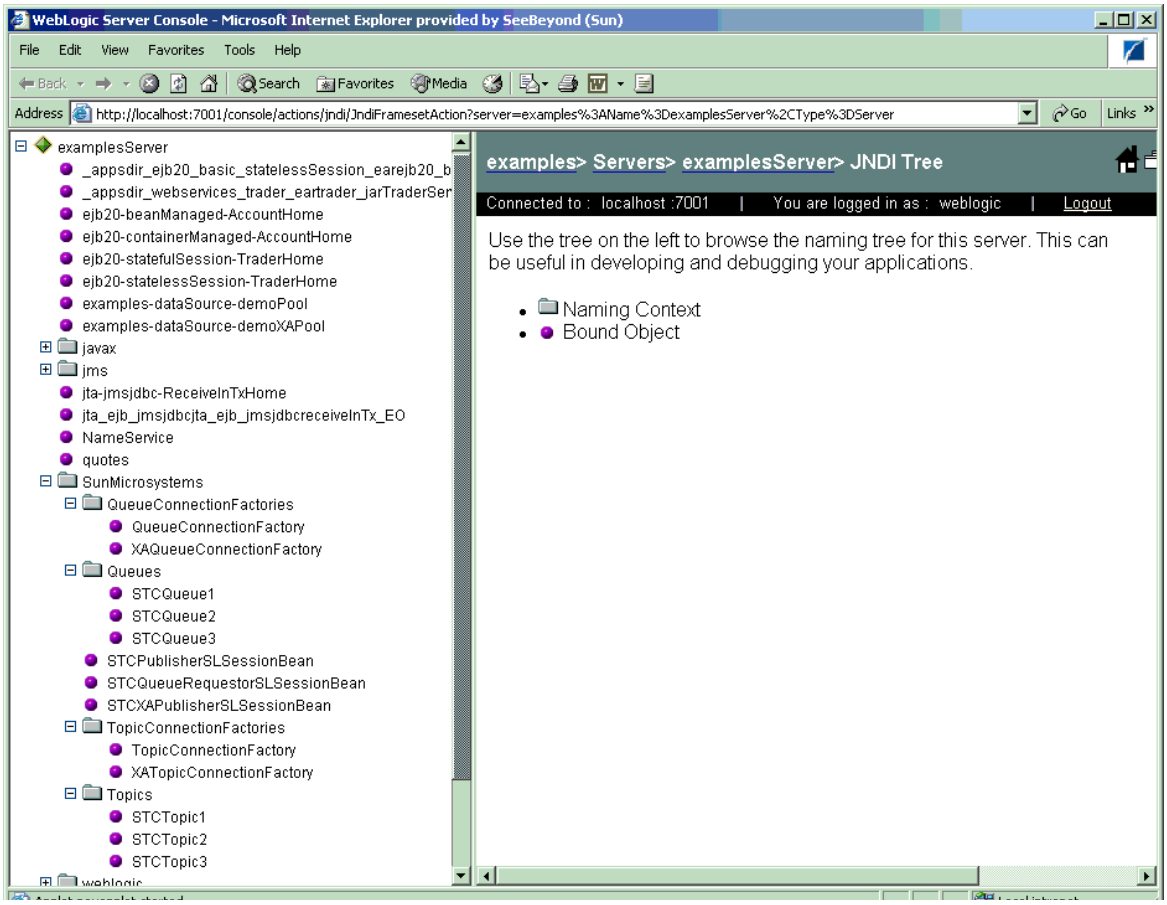
```
weblogicstartup - Sun Microsystems startup class invoked -
SunMicrosystems_Startup
weblogicstartup - Topic name: Topic.Sample3
weblogicstartup - Topic name: Topic.Sample2
weblogicstartup - Topic name: Topic.Sample1
```

```
weblogicstartup - Queue name: Queue.Sample3
weblogicstartup - Queue name: Queue.Sample2
weblogicstartup - Queue name: Queue.Sample1
weblogicstartup - Successfully invoked Sun Microsystems startup.
```

11 Start the Administration Console.

12 In the Console, go to Servers, <myserver> (or the new server instance). Right-click exampleServer and select View JNDI Tree to open the JNDI Tree window. Expand the Sun Microsystems node to verify that all Sun Microsystems JMS objects are now available (see Figure 25).

Figure 25 View the JNDI Tree



13 From the Navigator pane on the left, click on Examples, Deployments, EJB. Click on **Configure a new EJB**.

Note: *Before deploying the EJB, make sure that the JMS IQ Manager is running in Enterprise Manager.*

To Deploy the EJB:

1 In the left pane of the WebLogic Server Home, click open the Deployments node.

2 Right-click the **EJB Deployments** node and select **Deploy a new EJB Module** from the menu.

3 Select <BEA-Home>\weblogic81\server\lib

4 Click the **upload your file(s)** link, then click **Browse** and select <BEA-Home>\weblogic81\server\lib**weblogic.ejb.example.jar**. With the file selected, click **Upload**.

5 Select the uploaded **weblogic.ejb.example.jar** and click **Target Module**.

6 Select the server instance under **Available Servers**. Click the **right-arrow** to move the new server instance to **Target Servers**.

7 Enter **weblogic.ejb.example** as the name for this application (EJB).

8 Click the **Deploy** button. This installs the EJB on the WebLogic Administration Server.

Chapter 7

Using the WebLogic OTD Wizard

Object Type Definitions define external data formats that characterize the input and output data structures in a Collaboration Definition. This chapter describes how to build and use Object Type Definitions (OTDs) using the WebLogic OTD Wizard.

What's in This Chapter:

Note: *Java classes provided in the WebLogic OTD Wizard can contain APIs created using standard Sun JDK 1.3.x, JDK 1.4.x, or JDK 1.5.x, but they must be compatible with either versions of the JVM. For example, java code that is dependent on the JDK 1.3.x characteristic for java.util.TimeZone and java.util.SimpleTimeZone might not work with the same behavior or load correctly for JVM 1.4.x.*

Note: *JNI methods and inner classes are not supported.*

Important: *If the home or the remote interface class or their dependent class(es) contain(s) recursive reference(s), the support for the corresponding EJB methods will be limited.*

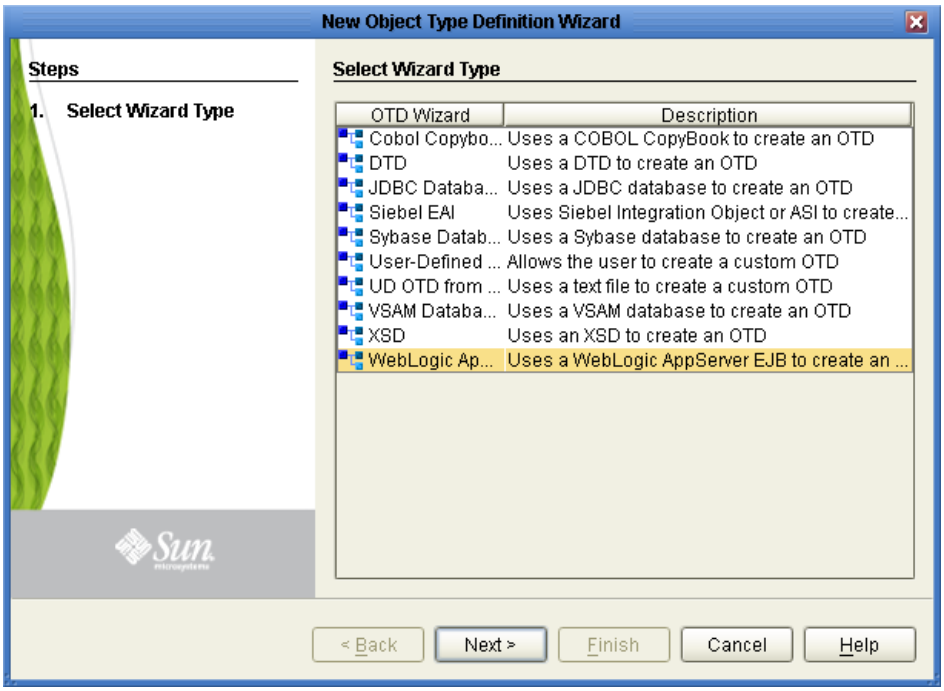
7.1 Creating a WebLogic OTD

Steps required to create an OTD include:

Select Wizard Type

1 From the Project Explorer tree, right click the Project and select **New > Object Type Definition** from the menu. The **Select Wizard Type** page appears, displaying the available **OTD** wizards. See Figure 26.

Figure 26 OTD Wizard Selection

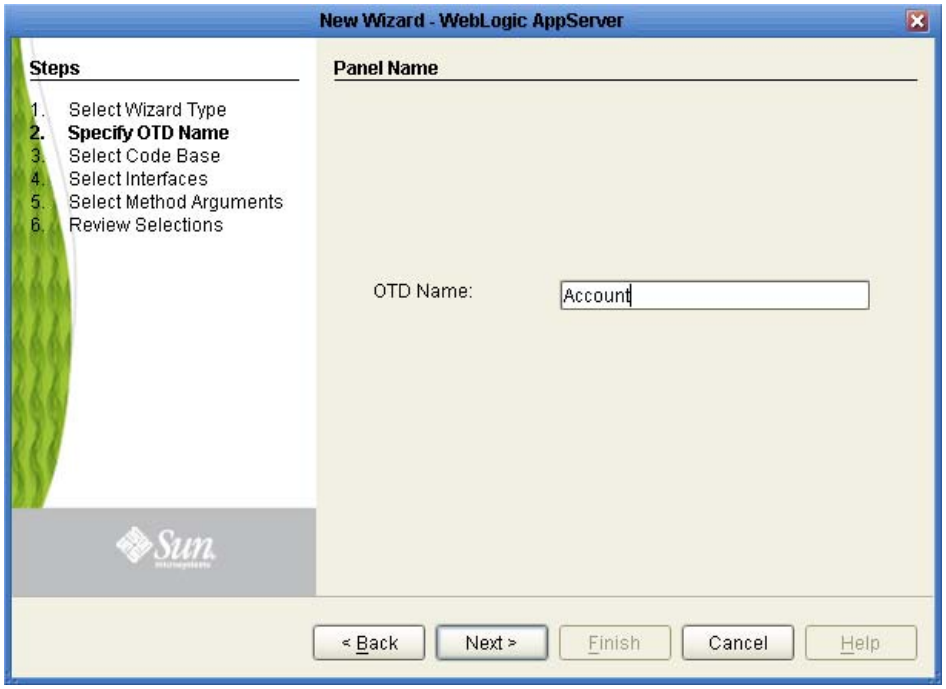


2 From the list, select the **WebLogic AppServer** and click **Next**. The **Specify OTD Name** page appears.

Specify OTD Name

3 Enter a name for the new OTD (see Figure 27).

Figure 27 Specify OTD Name



4 Click **Next**, the Select Home and Remote Interfaces page appears.

Select Code Base

From the **Select Code Base** page, you can select the directory that contains your EJB class files by selecting the root directory above the top-level Java package, or you can select a specific JAR file containing the EJB class files.

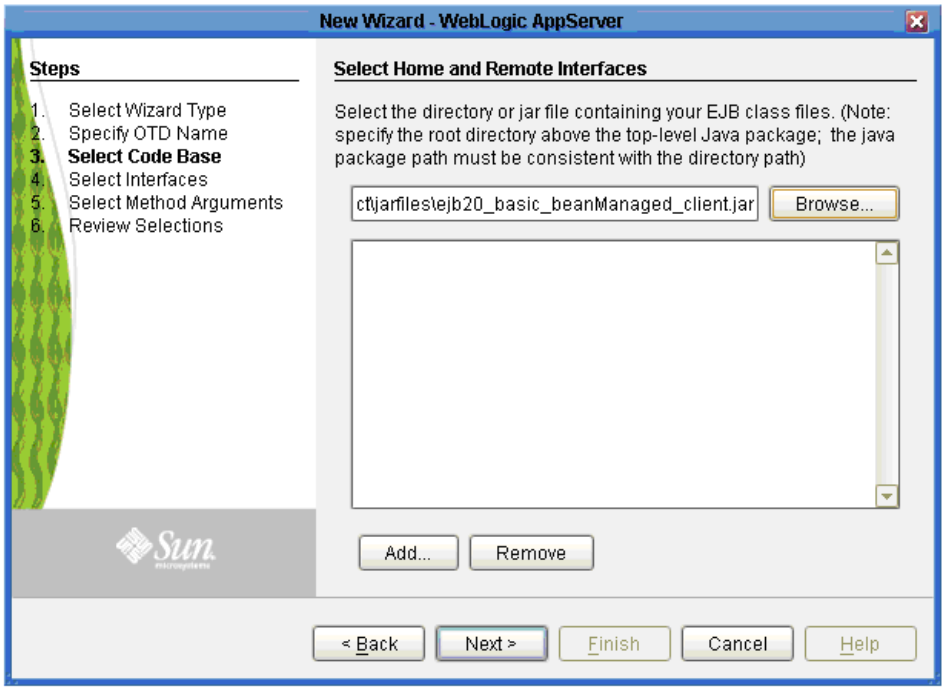
5 To select the root directory or archive file that contains the EJB class files, click **Browse** and navigate to and select the specific root directory or archive file. The EJB archive file must contain, at a minimum,both the Home and Remote Interface class pair for the EJB. Click **Add or Remove** to select or unselect any additional archive files to be available for design time (that is, in the JCD) and runtime (that is, in the Project EAR file). With the file or directory selected (at a minimum) click **Next** (see Figure 28).

If you do not specify a JAR file, the program searches through the entire directory looking for all Java Archive files. Searches on top level drives or directories can significantly increase search times. Only recognized Class File Root file names are accepted in the File Name field.

EJB class files must contain a package name for the EJB Home and Remote Interface Classes. The wizard does not support EJB class files that do not provide a package name.

Note: *EJB Class files and their dependent files must be located in a directory to be used with the OTD Wizard. The Wizard does not support EJB classes imbedded in archived files, such as EAR, WAR, or SAR files. These files must be extracted to a directory before they can be used to create an OTD.*

Figure 28 Select Code Base page

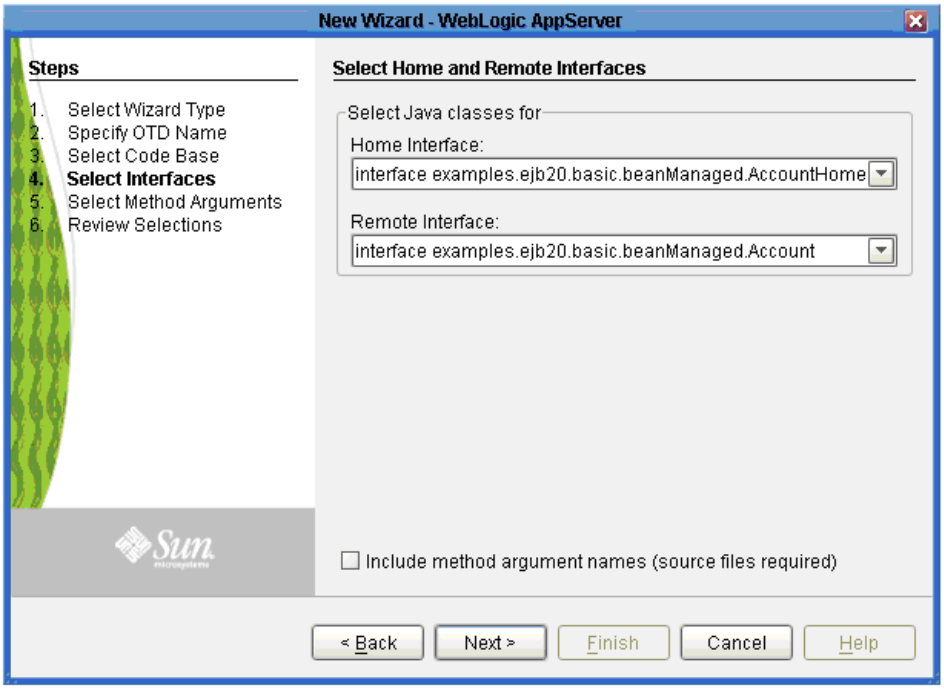


Select Home and Remote Interfaces

The **Select Home and Remote Interfaces** page displays the selected Java Home and Remote Interfaces. These fields are automatically populated. Both fields include a drop-down list that allows you to select the appropriate home and remote interface (if more than one choice is available).

6 Review the selected Home and Remote Interface fields (see Figure 29).

Figure 29 Select Home and Remote Interfaces page



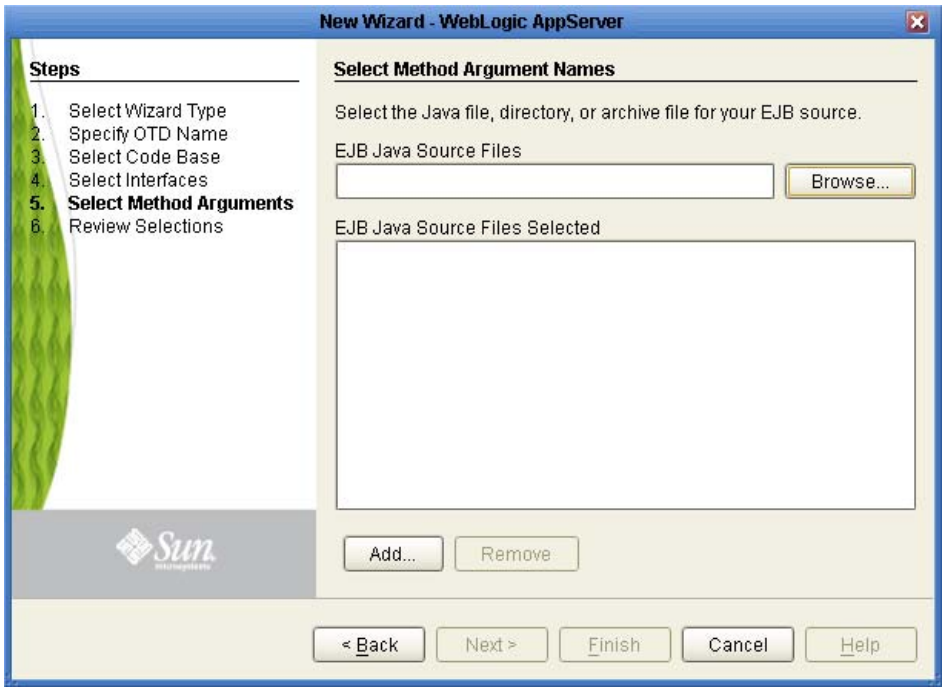
7 To add the method argument names to the OTD, select the Include method argument names option. Do not select this option if you do not have the EJB source code. Click **Next**. The Wizard advances to the **Select Methods Arguments** page.

If the selected EJB file's interfaces are valid and the Select Method argument names option was not selected, the wizard advances to the **Review Selections** page. The **Select Method Arguments Names** page appears.

Select Method Arguments

8 Enter the Java source file or click **Browse** to locate the java source files for the EJB archive supplied. Only a JAVA file or an archive file (containing JAVA files) will be accepted. If a directory is supplied, then only JAVA files are searched. See Figure 30.

Figure 30 Select Method Argument Names page



9 Click **Add** to locate and add Java source files to the **EJB Java Source Files Selected** field. Remove unwanted files from the **EJB Java Source Files Selected** field by selecting the file and clicking **Remove**. At least two files must be supplied - one for the home interface and one for the remote interface (or one EJB bean implementation source). Once the field contains all the necessary files, click **Next**.

Caution: *It is the user's responsibility to match the correct source files to the EJB.*

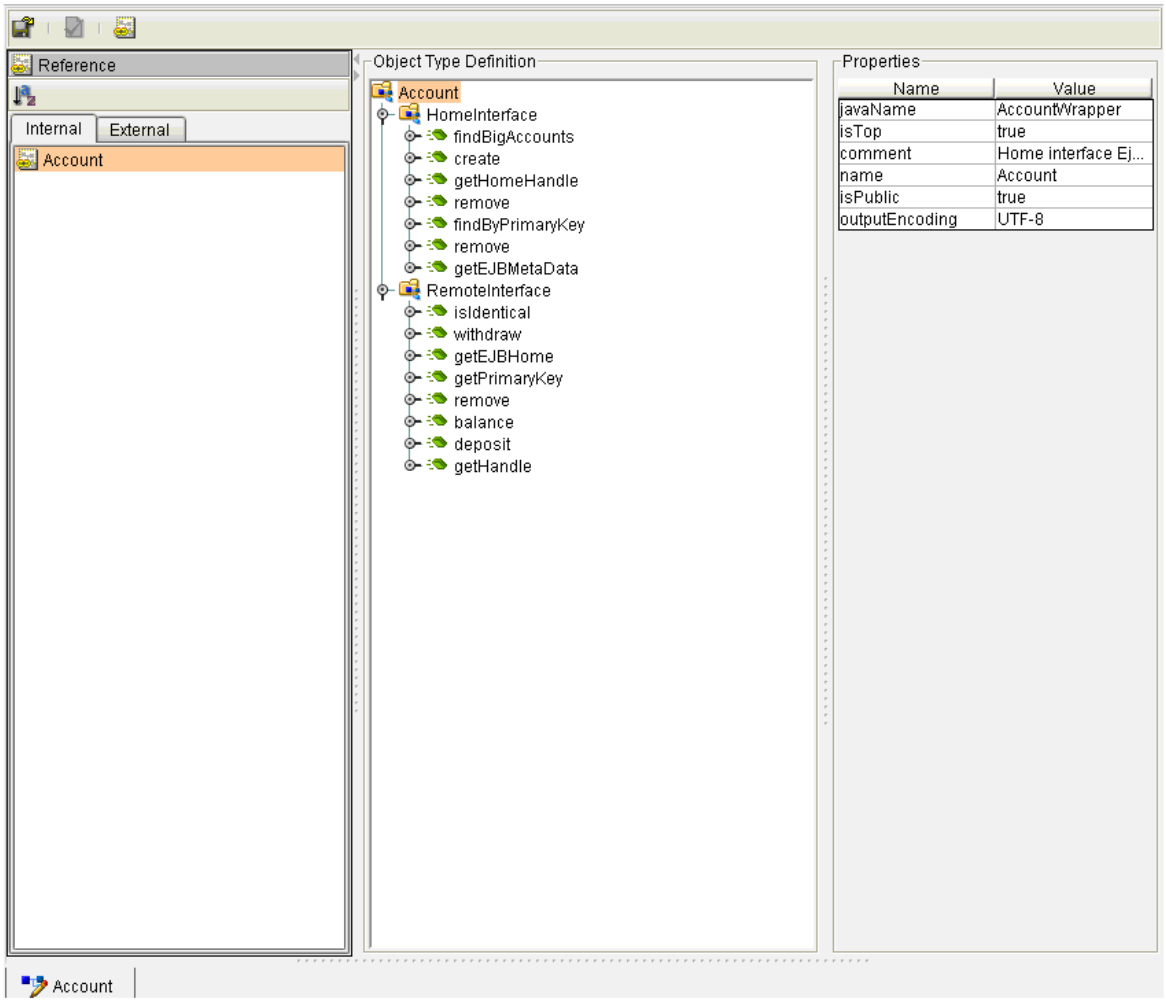
Review Selections

10 The Review Selections page appears. Review your selection information in the right pane of the page. To change any entries, click **Back** and return to the appropriate step.

Generate the OTD

11 Once you are satisfied with your selection information, click **Finish**. The OTD Editor appears with the generated OTD (see Figure 31).

Figure 31 OTD Editor - New OTD



Chapter 8

Implementing the WebLogic eWay Sample Projects

This chapter describes how to use the sample projects included in the installation CD-ROM package.

What's in This Chapter:

Note: *Sample projects mentioned in this chapter were created using WebLogic 8.1.*

8.1 Sample Projects Overview

Sample projects are designed to provide an overview of the basic functionality of the WebLogic eWay by identifying how to synchronously and asynchronously pass data to and from a WebLogic Server.

Types of synchronous and asynchronous communication include:

8.1.1 Synchronous Communication—eGate to WebLogic Server

Sample projects using synchronous communication between eGate and the WebLogic Server, require the creation of OTDs using WebLogic's Session and Entity Beans EJB interface classes.

Sample projects using synchronous communication in this manner include:

- **The prjWebLogic_Sample_JCD Sample Project on page 77**

8.1.2 Asynchronous Communication—WebLogic EJB to eGate JMS

Asychronous Communication from the WebLogic EJB to eGate JMS requires the implementation of Sun Microsystems' JMS IQ Manager into the WebLogic environment, allowing EJBs in the WebLogic container to receive messages from or send messages to eGate.

Sample projects using synchronous communication in this manner include:

- **The JMSQueueRequestor Sample Project** on page 83
- **The JMSTopicSubscribe Sample on page 84**
- **The JMSXATopicSubscribe Sample** on page 85

Note: *Installation of the eGate API Kit is required for asynchronous communication in eGate.*

Note: *WebLogic 9 is not supported for asynchronous communication.*

Preparing WebLogic

The following steps required to prepare WebLogic EJB for interaction with the eGate JMS:

1. Configure WebLogic to create JNDI entries for Sun Microsystems JMS on WebLogic Server instance startup.
2. Create an EJB that can publish to Sun Microsystems JMS. The basic sample Session Beans STCPublisherSLSession and STCQueueRequrestorSLSession are provided so that when instantiated, they publish to the Queue name listed in their parameters. Use these samples as models to build Session Beans.
3. Create a new Deployment Descriptor. An EJB is a Java class is written following the protocols of the application server. A deployment tool—a XML file similar to a configuration file for an eWay—is then used to make the EJBs available to other programs from the directory. An EJB in itself does not have parameters. Parameters that direct the behavior of the EJB—port number, class names for the JMS provider, and so on—are provided and stored in the Deployment Descriptor.
4. Take the Session Bean and Deployment Descriptor and use the WebLogic GUI to make the EJB available for external applications to call and publish to the Sun Microsystems JMS.

8.1.3 Asynchronous Communication—eGate JMS to a WebLogic Message Driven Bean

Asychronous Communication between the eGate JMS to a WebLogic Message Driven Bean also requires the implementation of Sun Microsystems' JMS IQ Manager into the WebLogic environment.

Sample projects using synchronous communication in this manner include:

- **The JMSQueueSend Sample on page 87**
- **The JMSTopicPublish Sample on page 88**
- **The JMSXAQueueSend Sample on page 89**

Note: *Installation of the eGate API Kit is required for asynchronous communication in eGate.*

Note: *WebLogic 9 is not supported for asynchronous communication.*

Preparing WebLogic

The following steps are required to prepare the WebLogic MDB:

1. Configure WebLogic to create JNDI entries for Sun Microsystems JMS on WebLogic Server instance startup. Responsibility for building the JNDI tree lies with the startup classes. Install these classes in the startup area of the Console and specify the name of the properties file.
2. Build the EJB and implement the business logic. Implementation uses JNDI to lookup TopicConnectionFactory.
3. Create a new Deployment Descriptor. An EJB is a Java class that is written following the protocols of the application server. A deployment tool is then used to make the EJBs available to other programs from the directory. The Deployment Descriptor comes in two parts:
   - **General EJB parameters (ejb-jar.xml)**— defines the session type (stateless, statefull), registers the Home and Remote classes with JNDI, and defines the JNDI name.
   - **Application Server vendor-specific parameters (weblogic-ejb-jar.xml)**— defines Pooling parameters and Reference Resource parameters.
4. Take the Bean class files and Deployment Descriptors then place these in a Jar file. Upload Jar files using the WebLogic Console. Classes are available to other applications once the EJB is deployed.

8.2 Using Sample Projects in eInsight

This section describes how to use sample projects with the Java CAPS eInsight Business Process Manager and the Web Services interface. This section does not provide an explanation of how to *create* a project that uses a Business Process Extension Language (BPEL). For these instructions, you should refer to the *eInsight Enterprise Service Bus User's Guide*.

8.2.1 Importing a Sample Project

Sample eWay Projects are included as part of the installation package. To import a sample eWay Project to the Enterprise Designer do the following:

1. The sample files are uploaded with the eWay's documentation SAR file and downloaded from the Sun Java Composite Application Platform Suite Installer's **Documentation** tab. The **WebLogic_eWay_Sample.zip** file contains the various sample Project ZIP files. Extract the samples to a local file.
2. Save all unsaved work before importing a Project.
3. From the Enterprise Designer's Project Explorer pane, right-click the Repository and select **Import** from the shortcut menu. The **Import Manager** appears.
4. Browse to the directory that contains the sample Project zip file. Select the sample file and click **Import**. After the sample Project is successfully imported, click **Close**.
5. Before an imported sample Project can be run you must do the following:
   - Configure the eWays for your specific system (see **"Setting the Properties" on page 72**)
   - Configure the Integration Server (see **"Configuring the Integration Server" on page 72**)
   - Create an **Environment** (see **"Creating an Environment" on page 73**)
   - Create a **Deployment Profile** (see **"Creating the Deployment Profile" on page 73**)
   - Create and start a domain (see **"Creating and Starting the Domain" on page 74**)
   - Build and deploy the Project (see **"Building and Deploying the Project" on page 75**)

8.2.2 The eInsight Engine and Components

eGate components can be deployed as Activities in eInsight Business Processes. Once a component is associated with an Activity, eInsight invokes it using a Web Services interface. eGate components that can interface with eInsight in this way include the following:

- Object Type Definitions (OTDs)
- eWays (using default receive and write operators of the File eWay)
- Collaborations

Using the Enterprise Designer and eInsight, you can add an Activity to a Business Process, then associate that Activity with an eGate component (for example, an eWay). Then, when eInsight runs the Business Process, it automatically invokes that component via its Web Services interface.

8.2.3 The prjWebLogic_Sample_BPEL Sample Project

The eInsight sample project **prjWebLogic_Sample_BPEL** demonstrates a synchronous interaction between the WebLogic EJB and the eGate JMS. As mentioned previously, synchronous communication is considered an unbuffered process, requiring complete data transmission and reply or confirmation of message transmission failure before enacting additional communication processes. The nature of the sample project is dependent on the services invoked through the methods and properties of the EJB that is used to create the OTD. In the **prjWebLogic_Sample_BPEL** sample project, these services are used to detail the creation and subsequent funding of an account.

The sample project includes the following business processes:

bpCreateAccount

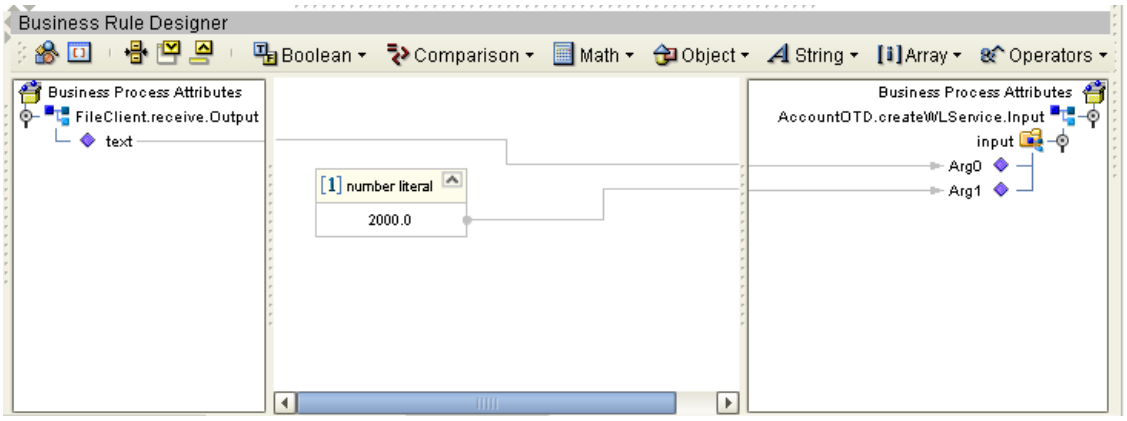
This business process describes the account creation process seen in Figure 32.

Figure 32 Account Creation Business Process

Business Process:

1. The File eWay subscribes to an external directory and picks up a text request for specific account data.
2. Account data is copied to the input container FileClient.receive.Output, using the AccountOTD.createWLService BPEL service. A number literal of 2000 is also passed into the OTD (see Figure 33) before sending the combined data to the FileClient.write container.

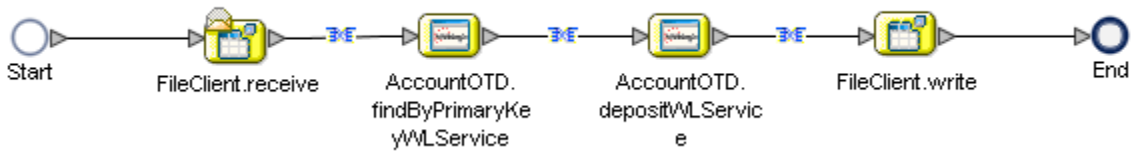
Figure 33 Account Creation



bpDepositAmount

This business process describes the account deposit process as seen in Figure 34.

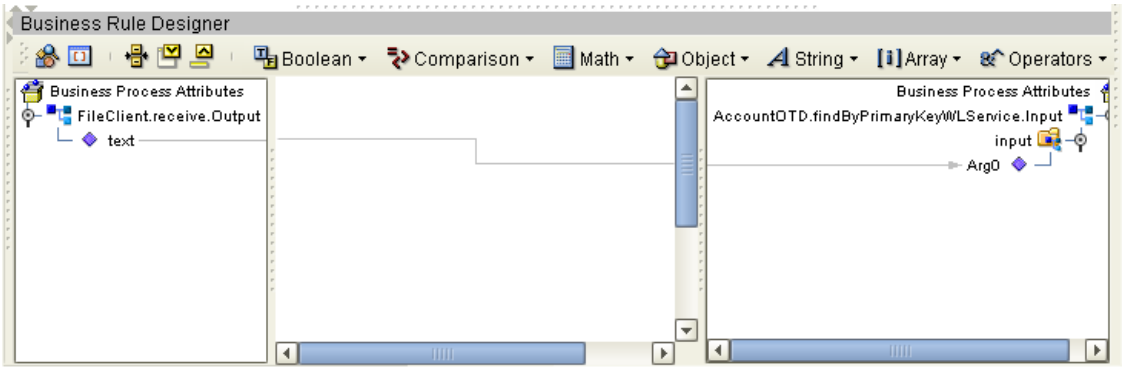
Figure 34 Account Deposit Business Process



Business Process:

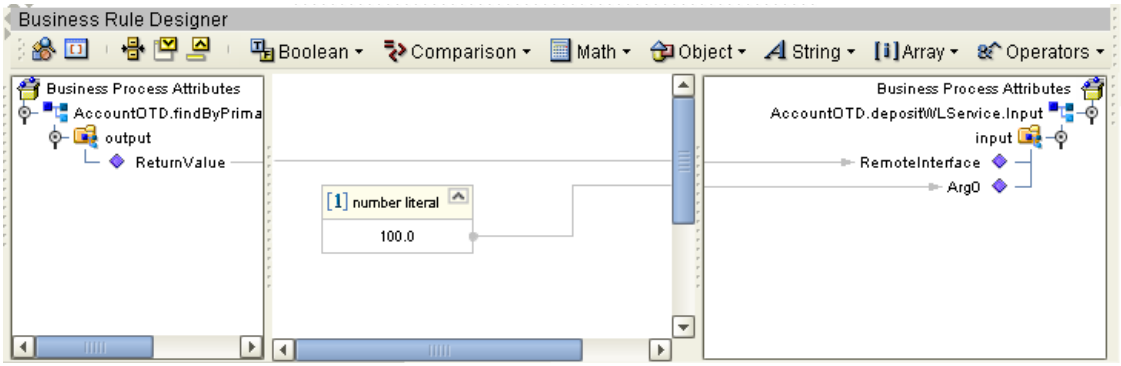
1. The file eWay subscribes to an external directory and picks up a text request for specific account data.
2. Account data is copied to the input container FileClient.receive.Output, using the AccountOTD.findByPrimaryKeyWLService BPEL service. This service also requests the specific data (Account ID and amount) using the primary key from WebLogic Server, as seen in Figure 35.

Figure 35 Retrieving Account Data



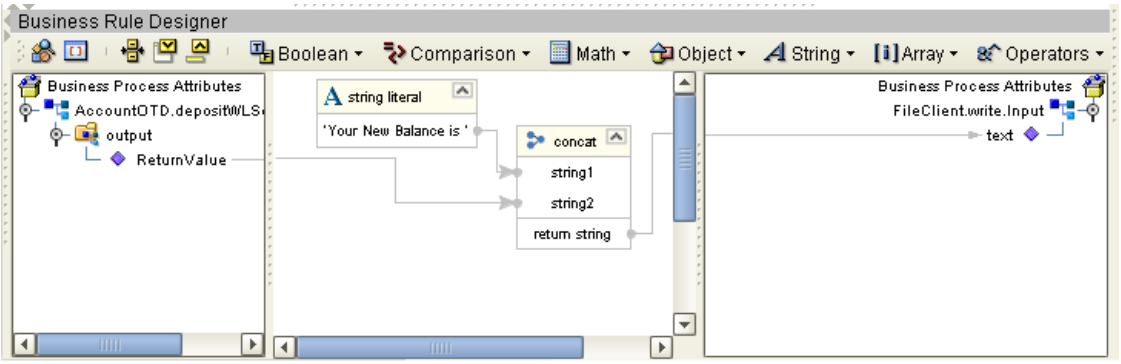
3. The AccountOTD.depositWLService OTD is then invoked, adding the number literal of 100 to the account balance listed in WebLogic Server (see Figure 36).

Figure 36 Making the Deposit



4 Combined data is sent to the FileClient.write container, as seen in Figure 37.

Figure 37 Returning the Response



8.2.4 Setting the Properties

Each sample project contains properties accessible either through the File or WebLogic eWay, located on the Project Explorer Connectivity Map, or from the WebLogic eWay External System, located in the Environment. To configure the WebLogic eWay properties for your project, see **"Setting Properties of the WebLogic eWay" on page 17**.

8.2.5 Configuring the Integration Server

You must set your SeeBeyond Integration Server Password property before deploying your Project.

1 From the Environment Explorer, right-click **IntegrationSvr1** under your **Logical Host**, and select **Properties** from the shortcut menu. The Integration Server Properties Editor appears.

2 Click the **Password** property field under **Sun SeeBeyond Integration Server Configuration**. An ellipsis appears in the property field.

3 Click the ellipsis. The **Password Settings** dialog box appears.

4 Enter **STC** as the **Specific Value** and as the **Confirm Password**, and click **OK**.

5 Click **OK** to accept the new property and close the Properties Editor.

For more information on deploying a Project see the *Sun SeeBeyond Java™ Composite Application Platform Suite Deployment Guide*.

8.2.6. Creating an Environment

Environments include the external systems, Logical Hosts, integration servers and message servers used by a Project and contain the configuration information for these components. Environments are created using the Environment Editor.

1 From the Enterprise Explorer, click the **Environment Explorer** tab.

2 Right-click the Repository and select **New Environment**. A new Environment is added to the Environment Explorer tree.

3 Rename the new Environment to **envWebLogic**.

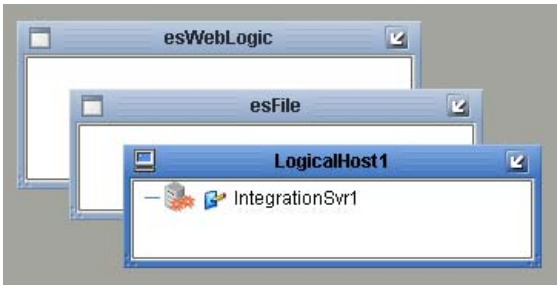
4 Right-click **envWebLogic** and select **New > WebLogic External System**. Name the External System **esWebLogic**. Click **OK**. **esWebLogic** is added to the Environment Editor.

5 Right-click **envWebLogic** and select **New > File External System**. Name the External System **esFile**. Click **OK**. **esFile** is added to the Environment Editor.

6 Right-click **envWebLogic** and select **New > Logical Host**. **LogicalHost1** is added to the Environment Editor.

7 From the Environment Explorer tree, right-click **LogicalHost1** and select **New > Sun SeeBeyond Integration Server**. A new Integration Server (**IntegrationSvr1**) is added to the Environment Explorer tree under LogicalHost1 (seeFigure 38).

Figure 38 Environment Editor



8.2.7 Creating the Deployment Profile

Deployment Profiles are specific instances of a Project in a particular Environment. A Deployment Profile is created using the Enterprise Designer's Deployment Editor.

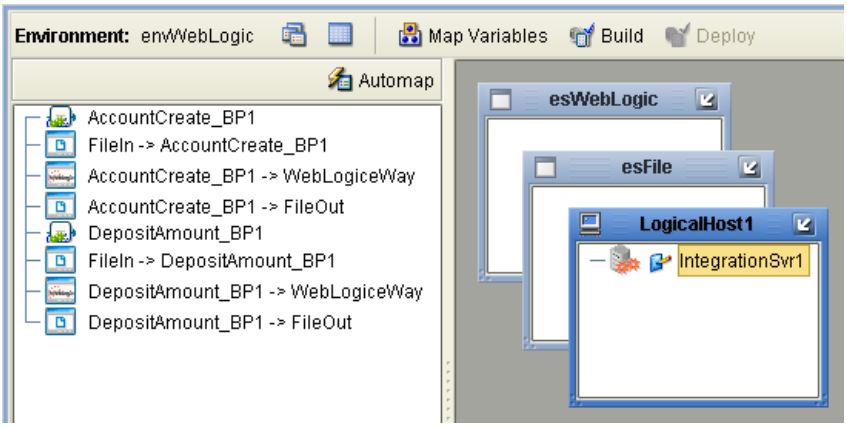
To create a Deployment Profile, do the following:

1 From the Enterprise Explorer's Project Explorer, right-click the Project and select **New** > **Deployment Profile**.

2 From the **Create Deployment Profile** dialog box, enter a name for the Deployment Profile (for this example, **dpWebLogic_Sample_BPEL**). Select the appropriate Environment (**envWebLogic)** and click **OK**.

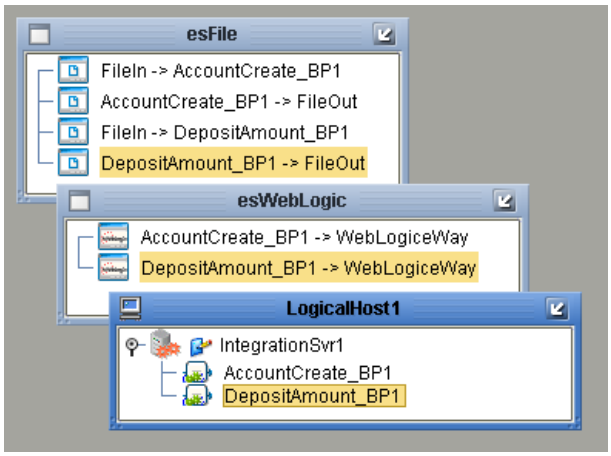
3 Click the **Auto Map** icon. as displayed in Figure 39.

Figure 39 Deployment Profile - Auto Map



4 The Project's components are automatically mapped to their system windows as seen in Figure 40.

Figure 40 Deployment Profile



8.2.8 Creating and Starting the Domain

To deploy your Project, you must first create a domain. A domain is an instance of a Logical Host.

Create and Start the Domain

1. Navigate to your **<*JavaCAPS51*>\logicalhost** directory (where <*JavaCAPS51*> is the location of your Sun Java Composite Application Platform Suite installation.
2. Double-click the **domainmgr.bat** file. The **Domain Manager** appears.
3. If you have already created a domain, select your domain in the Domain Manager and click the **Start an Existing Domain** button. Once your domain is started, a green check mark indicates that the domain is running.
4. If there are no existing domains, a dialog box indicates that you can create a domain now. Click **Yes**. The **Create Domain** dialog box appears.
5. Make any necessary changes to the **Create Domain** dialog box and click **Create**. The new domain is added to the Domain Manager. Select the domain and click the **Start an Existing Domain** button. Once your domain is started, a green check mark indicates that the domain is running.
6. For more information about creating and managing domains see the *Sun SeeBeyond eGate Integrator System Administration Guide*.

8.2.9 Building and Deploying the Project

The Build process compiles and validates the Project's Java files and creates the Project EAR file.

Note: *If your EJB was compiled with JDK 1.5, see* **"Build the Deployment (EAR) File Using the Commandline Codegen Tool" on page 75**.

Build the Project

1. From the Deployment Editor toolbar, click the **Build** icon.
2. If there are any validation errors, a **Validation Errors** pane will appear at the bottom of the Deployment Editor and displays information regarding the errors. Make any necessary corrections and click **Build** again.
3. After the Build has succeeded you are ready to deploy your Project.

Deploy the Project

1. From the Deployment Editor toolbar, click the **Deploy** icon. Click **Yes** when the **Deploy** prompt appears.
2. A message appears when the project is successfully deployed.

8.2.10 Build the Deployment (EAR) File Using the Commandline Codegen Tool

The Sun SeeBeyond Enterprise Designer is packaged with **JDK 1.4**. If your EJB is compiled with **JDK version 1.5 or above**, you must use **Command Line Codegen** to build the Deployment (EAR) file. Please refer to the *Sun SeeBeyond eGate Integrator User's Guide (Building an Application File From the Command Line)* for details.

For a simple description of how to use the Commandline Codegen Tool to create an EAR file, do the follow:

1. Download the CommandLineCodegen tool from the Sun Java Composite Application Platform Suite Installer.
2. Extract the files into a local directory.
3. Download and install **JDK 1.5** from Sun website, or make a note of the JDK that was used to compile the EJB.
4. Download and install **ANT** (version 1.6.2 or above) from Apache Software Foundation at **http://ant.apache.org**.
5. Change the **build.properties** file in the commandlinecodegen directory. At a minimum, you must change the following parameters (with sample data). Make sure there is no trailing space after each parameter. The parameters below are usually different for each specific environment.

```
commandline.rep.url=http://localhost:12000/rep
commandline.rep.user=Administrator
commandline.rep.pass=STC
commandline.rep.dir=localrepository
commandline.rep.projectName=prjWebLogice_Sample_JCD
commandline.rep.projectDeployName=dpWebLogic_Sample_JCD
```

6. Open a Command Prompt to the **commandlinecodegen** directory.
7. Make sure **JAVA_HOME** and **ANT_HOME** are pointing to the proper directory (this should be consistent with step 3 and 4 above).
8. Run commandline codegen by issuing the following command from the Command Prompt in your commandlinecodegen directory:

```
ant -propertyfile build.properties
```

9. The resulting EAR file is be located in the following directory:

```
commandlinecodegen\localrepository\DEST
```

   This is where the EAR file for the next step is located.

10. Deploy the EAR file into Sun SeeBeyond eGate Integrator using the Sun SeeBeyond Enterprise Manager. See the *Sun SeeBeyond eGate™ System Administration Guide* (Chapter 3) for details.

8.2.11. Running the Project

To run your deployed sample Project do the following

1. From your configured input directory, paste (or rename) the sample input file to trigger the eWay.

From your output directory, verify the output data.

8.3 Using the Sample Projects in eGate

This section describes how the sample projects included with the WebLogic eWay are implemented using eGate Integrator.

8.3.1 Importing a Sample Project

Sample eWay Projects are included as part of the installation package. To import a sample eWay Project to the Enterprise Designer do the following:

1. The sample files are uploaded with the eWay's documentation SAR file and downloaded from the Sun Java Composite Application Platform Suite Installer's **Documentation** tab. The **WebLogic_eWay_Sample.zip** file contains the various sample Project ZIP files. Extract the samples to a local file.
2. Save all unsaved work before importing a Project.
3. From the Enterprise Designer's Project Explorer pane, right-click the Repository and select **Import** from the shortcut menu. The **Import Manager** appears.
4. Browse to the directory that contains the sample Project zip file. Select the sample file and click **Import**. After the sample Project is successfully imported, click **Close**.
5. Before an imported sample Project can be run you must do the following:
   - Configure the eWays for your specific system (see **"Setting the Properties" on page 78**)
   - Configure the Integration Server (see **"Configuring the Integration Server" on page 78**)
   - Create an **Environment** (see **"Creating an Environment" on page 79**)
   - Create a **Deployment Profile** (see **"Creating the Deployment Profile" on page 79**)
   - Create and start a domain (see **"Creating and Starting the Domain" on page 80**)
   - Build and deploy the Project (see **"Building and Deploying the Project" on page 81**)

8.3.2 The prjWebLogic_Sample_JCD Sample Project

The **prjWebLogic_Sample_JCD** sample project demonstrates synchronous communication from eGate to WebLogic. The sample Project is similar to the **prjWebLogic_Sample_JCD** sample project described in **"The prjWebLogic_Sample_BPEL Sample Project" on page 70**, with the exception that there is no eInsight Business Process.

The Connectivity Maps for this sample appear as follows:

Figure 41 prjWebLogic_Sample_JCD Connectivity Map—Create Account

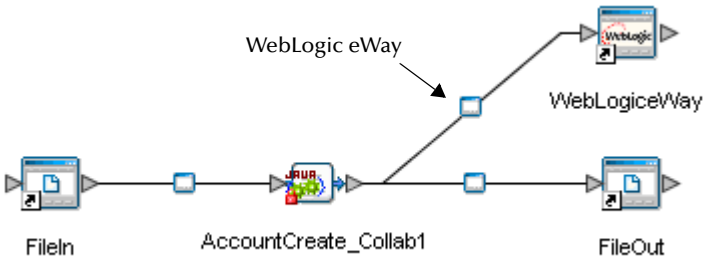
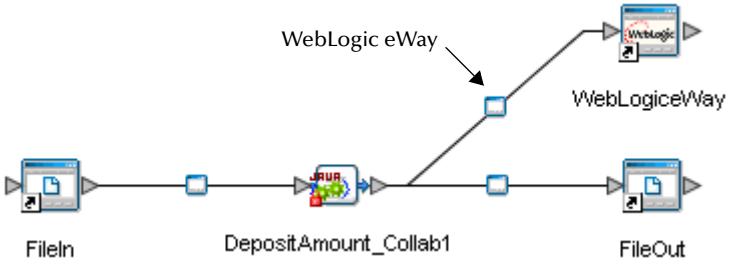


Figure 42 prjWebLogic_Sample_JCD Connectivity Map—Deposit



This sample project demonstrates creating and then depositing funds to an account using two collaborations (**jcdAccountCreate** and **jcdDepositAmount**). The first collaboration subscribes to the FileIn (File eWay), and then picks up an account name (String). Both the account name and a string literal value of 1000.00 (double) are concatenated and converted to text before passing into the FileOut (File eWay). During the second collaboration, a new deposit of 100.00 is added to the previous balance and a response is written to the FileOut (File eWay) revealing the updated balance.

8.3.3 Setting the Properties

Each sample project contains properties accessible either through the File or WebLogic eWay, located on the Project Explorer Connectivity Map, or from the WebLogic eWay External System, located in the Environment. To configure the WebLogic eWay properties for your project, see **"Setting Properties of the WebLogic eWay" on page 17**.

8.3.4 Configuring the Integration Server

You must set your SeeBeyond Integration Server Password property before deploying your Project.

1. From the Environment Explorer, right-click **IntegrationSvr1** under your **Logical Host**, and select **Properties** from the shortcut menu. The Integration Server Properties Editor appears.
2. Click the **Password** property field under **Sun SeeBeyond Integration Server Configuration**. An ellipsis appears in the property field.
3. Click the ellipsis. The **Password Settings** dialog box appears.

4 Enter **STC** as the **Specific Value** and as the **Confirm Password**, and click **OK**.

5 Click **OK** to accept the new property and close the Properties Editor.

For more information on deploying a Project see the *Sun SeeBeyond Java™ Composite Application Platform Suite Deployment Guide*.

8.3.5. Creating an Environment

Environments include the external systems, Logical Hosts, integration servers and message servers used by a Project and contain the configuration information for these components. Environments are created using the Environment Editor.

1 From the Enterprise Explorer, click the **Environment Explorer** tab.

2 Right-click the Repository and select **New Environment**. A new Environment is added to the Environment Explorer tree.

3 Rename the new Environment to **envWebLogic**.

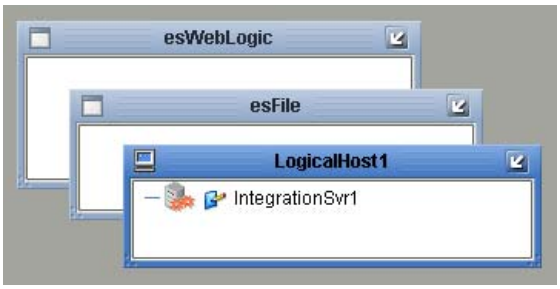
4 Right-click **envWebLogic** and select **New > WebLogic External System**. Name the External System **esWebLogic**. Click **OK**. **esWebLogic** is added to the Environment Editor.

5 Right-click **envWebLogic** and select **New > File External System**. Name the External System **esFile**. Click **OK**. **esFile** is added to the Environment Editor.

6 Right-click **envWebLogic** and select **New > Logical Host**. **LogicalHost1** is added to the Environment Editor.

7 From the Environment Explorer tree, right-click **LogicalHost1** and select **New > Sun SeeBeyond Integration Server**. A new Integration Server (**IntegrationSvr1**) is added to the Environment Explorer tree under LogicalHost1 (seeFigure 43).

Figure 43 Environment Editor



8.3.6 Creating the Deployment Profile

Deployment Profiles are specific instances of a Project in a particular Environment. A Deployment Profile is created using the Enterprise Designer's Deployment Editor.

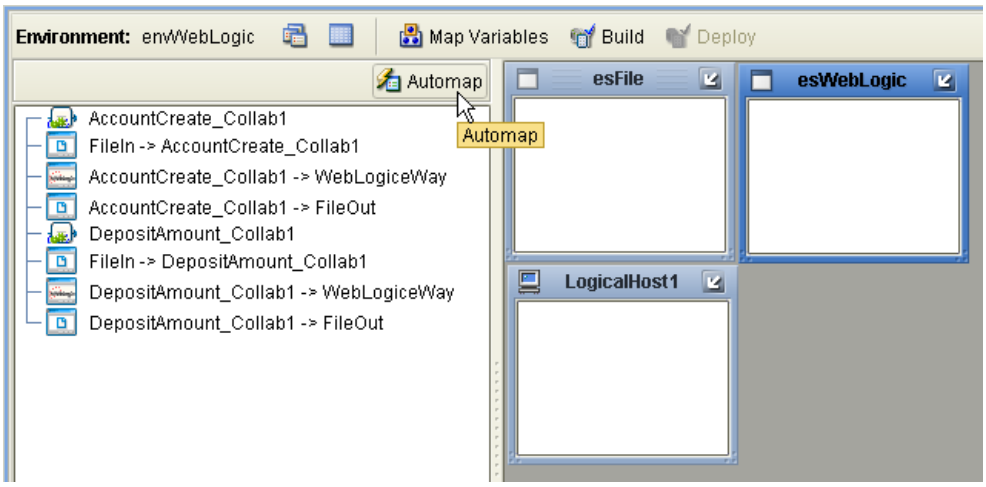
To create a Deployment Profile, do the following:

1 From the Enterprise Explorer's Project Explorer, right-click the Project and select **New** > **Deployment Profile**.

2 From the **Create Deployment Profile** dialog box, enter a name for the Deployment Profile (for this example, **dpWebLogic_Sample_JCD**). Select the appropriate Environment (**envWebLogic)** and click **OK**.

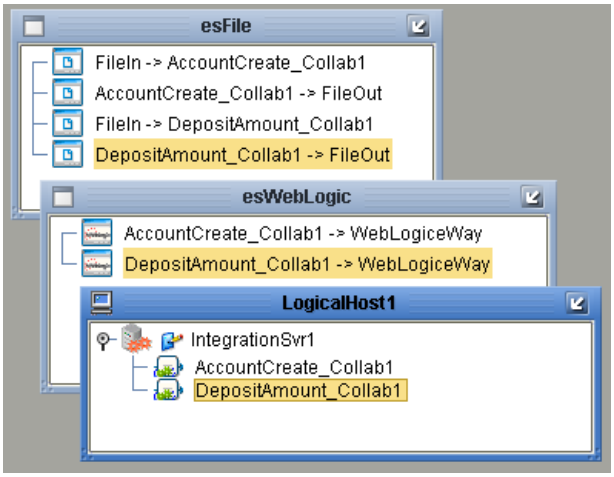
3 Click the **Auto Map** icon. as displayed in Figure 44.

Figure 44 Deployment Profile - Auto Map



4 The Project's components are automatically mapped to their system windows as seen in Figure 45.

Figure 45 Deployment Profile



8.3.7 Creating and Starting the Domain

To deploy your Project, you must first create a domain. A domain is an instance of a Logical Host.

Create and Start the Domain

1 Navigate to your **<*JavaCAPS51*>\logicalhost** directory (where <*JavaCAPS51*> is the location of your Sun Java Composite Application Platform Suite installation.

2 Double-click the **domainmgr.bat** file. The **Domain Manager** appears.

3 If you have already created a domain, select your domain in the Domain Manager and click the **Start an Existing Domain** button. Once your domain is started, a green check mark indicates that the domain is running.

4 If there are no existing domains, a dialog box indicates that you can create a domain now. Click **Yes**. The **Create Domain** dialog box appears.

5 Make any necessary changes to the **Create Domain** dialog box and click **Create**. The new domain is added to the Domain Manager. Select the domain and click the **Start an Existing Domain** button. Once your domain is started, a green check mark indicates that the domain is running.

6 For more information about creating and managing domains see the *Sun SeeBeyond eGate Integrator System Administration Guide*.

8.3.8 Building and Deploying the Project

The Build process compiles and validates the Project's Java files and creates the Project EAR file.

Note: *If your EJB was compiled with JDK 1.5, see* **"Build the Deployment (EAR) File Using the Commandline Codegen Tool" on page 75**.

Build the Project

1 From the Deployment Editor toolbar, click the **Build** icon.

2 If there are any validation errors, a **Validation Errors** pane will appear at the bottom of the Deployment Editor and displays information regarding the errors. Make any necessary corrections and click **Build** again.

3 After the Build has succeeded you are ready to deploy your Project.

Deploy the Project

1 From the Deployment Editor toolbar, click the **Deploy** icon. Click **Yes** when the **Deploy** prompt appears.

2 A message appears when the project is successfully deployed.

8.3.9 Build the Deployment (EAR) File Using the Commandline Codegen Tool

The Sun SeeBeyond Enterprise Designer is packaged with **JDK 1.4**. If your EJB is compiled with **JDK version 1.5 or above**, you must use **Command Line Codegen** to build the Deployment (EAR) file. Please refer to the *Sun SeeBeyond eGate Integrator User's Guide (Building an Application File From the Command Line)* for details.

For a simple description of how to use the Commandline Codegen Tool to create an EAR file, do the follow:

1 Download the CommandLineCodegen tool from the Sun Java Composite Application Platform Suite Installer.

2 Extract the files into a local directory.

3 Download and install **JDK 1.5** from Sun website, or make a note of the JDK that was used to compile the EJB.

4 Download and install **ANT** (version 1.6.2 or above) from Apache Software Foundation at **http://ant.apache.org**.

5 Change the **build.properties** file in the commandlinecodegen directory. At a minimum, you must change the following parameters (with sample data). Make sure there is no trailing space after each parameter. The parameters below are usually different for each specific environment.

```
commandline.rep.url=http://localhost:12000/rep
commandline.rep.user=Administrator
commandline.rep.pass=STC
commandline.rep.dir=localrepository
commandline.rep.projectName=prjWebLogiceWay
commandline.rep.projectDeployName=dpBmpSimple
```

6 Open a Command Prompt to the **commandlinecodegen** directory.

7 Make sure **JAVA_HOME** and **ANT_HOME** are pointing to the proper directory (this should be consistent with step 3 and 4 above).

8 Run commandline codegen by issuing the following command from the Command Prompt in your commandlinecodegen directory:

```
ant -propertyfile build.properties
```

9 The resulting EAR file is be located in the following directory:

```
commandlinecodegen\localrepository\DEST
```

This is where the EAR file for the next step is located.

10 Deploy the EAR file into Sun SeeBeyond eGate Integrator using the Sun SeeBeyond Enterprise Manager. See the *Sun SeeBeyond eGate™ System Administration Guide* (Chapter 3) for details.

8.3.10. Running the Project

To run your deployed sample Project do the following

1 From your configured input directory, paste (or rename) the sample input file to trigger the eWay.

From your output directory, verify the output data.

8.4 Using the JMS Sample Projects in eGate

This section describes how the JMS sample projects included with the WebLogic eWay are implemented using eGate Integrator.

8.4.1 The JMSQueueRequestor Sample Project

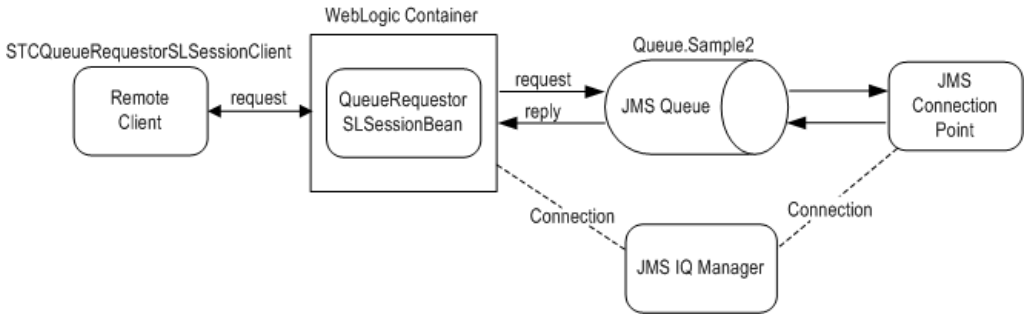
JMSQueueRequestor is an inbound sample project that demonstrates how a remote client requests and receives messages asynchronously from a JMS queue.

The Connectivity Map for this sample project appears as follows:

Figure 46 JMSQueueRequrestor Connectivity Map

In this sample, the Collaboration (crJMSQueueRequestor) subscribes to the Queue (Queue.Sample2), picks up messages, and then publishes messages to a second Queue (DummyQueue1). The Collaboration is configured to use the internal Sun Microsystems JMS IQ Manager as the JMS server. The Collaboration constructs a reply string, by prepending the String "This is a text message" to the message it received from the Queue and manually publishes the reply back to the Session Bean. In this case, the STCQueueRequestorSLSessionBean Session Bean acts as the sender to the Queue.Sample2 Queue and waits for the reply from eGate. Essentially, this demonstrates a request/reply usage of the QueueRequestor JMS object by the STCQueueRequestorSLSessionBean.

Figure 47 JMSQueueRequestor Sample Components



As seen in Figure 47, the stand-alone remote client, com.stc.eways.ejb.sessionbean.queuerequestor.STCQueueRequestorSLSessionClient, is used to invoke the request() method of the STCQueueRequestorSLSessionBean and wait for a reply from the Session Bean. As parameters, the client takes the provider URL of the WebLogic JNDI where the Session Bean is bound, the JNDI name of the Session Bean (SunMicrosystems.STCQueueRequestorSLSessionBean), a text message or a file name, and the option specifying whether the third parameter is a file or a text message (msg). For example, the following command sends the message “This is a text message”:

```
java com.stc.eways.ejb.sessionbean.queuerequestor.STCQueueRequestorSLSessionClient t3://
localhost:7001 SunMicrosystems.STCQueueRequestorSLSessionBean "This is a text message." msg
```

whereas the following command sends the message contained in the file c:\temp\testfile.txt:

```
java com.stc.eways.ejb.sessionbean.queuerequestor.STCQueueRequestorSLSessionClient t3://
localhost:7001 SunMicrosystems.STCQueueRequestorSLSessionBean c:\temp\testfile.txt file
```

8.4.2 The JMSTopicSubscribe Sample

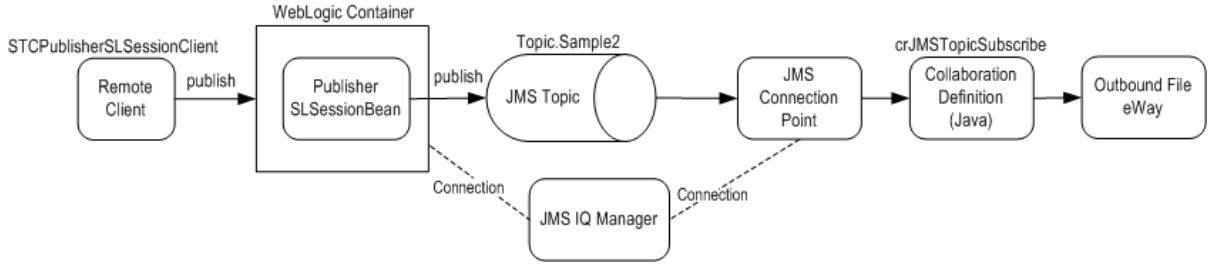
JMSTopicSubscribe is an inbound example that demonstrates how a remote client is used to send a messages to eGate asynchronously through a JMS topic.

The Connectivity Map for this sample project appears as follows:

Figure 48 JMSTopicSubscribe Connectivity Map

In this sample, the STCPublisherSLSessionBean Session Bean acts as publisher to the JMS Topic. This demonstrates how messages are published asynchronously from an EJB running in WebLogic to a Sun Microsystems JMS Topic. The Collaboration (crJMSTopicSubscribe) seen on the Connectivity Map is configured to use the internal Sun Microsystems JMS IQ Manager as the JMS server. It subscribes to the JMS client properties on the Topic.Sample2 Topic and sends data received to the Inbound File eWay.

Figure 49 JMSTopicSubscribe Sample Components



As seen in Figure 49, the stand-alone remote client, com.stc.eways.ejb.sessionbean.publisher.STCPublisherSLSessionClient, can be used to invoke the publish() method of the STCPublisherSLSessionBean to send a message to eGate asynchronously. The parameters taken by the client are:

- The provider URL of the WebLogic JNDI where the Session Bean is bound
- The JNDI name of the Session Bean (SunMicrosystems.STCPublisherSLSessionBean)
- A text message or a file name
- The option specifying whether the third parameter is a file or a text message (msg).

For example, the following command sends the message “This is a text message”:

```
java com.stc.eways.ejb.sessionbean.publisher.STCPublisherSLSessionClient t3://localhost:7001
SunMicrosystems.STCPublisherSLSessionBean "This is a text message." msg
```

Whereas the following command sends the message contained in the file c:\temp\testfile.txt:

```
java com.stc.eways.ejb.sessionbean.publisher.STCPublisherSLSessionClient t3://localhost:7001
SunMicrosystems.STCPublisherSLSessionBean c:\temp\testfile.txt file
```

Note: *Before running this client, make sure that the system classpath includes ejb.jar, weblogic.jar (with ejb.jar proceeding weblogic.jar in order), and weblogic.ejb.example.jar.*

The result of the test is that eGate sees the message that the remote client sent to the STCPublisherSLSessionBean. The message is written to an output file.

Note: *For more information on eWay Connection Configuration Parameters for JMS see* **"WebLogic eWay Properties" on page 20***.*

8.4.3 The JMSXATopicSubscribe Sample

JMSXATopicSubscribe is an inbound example that demonstrates how a remote client is used to asynchronously pass two-phase commit protocol (XA) messages into a JMS topic.

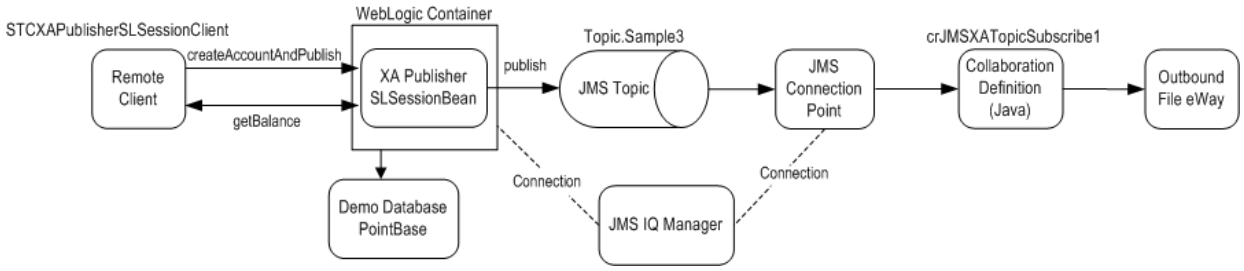
The Connectivity Map for this sample appears as follows:

Figure 50 JMSXATopicSubscribe Sample

In this sample, the Inbound File eWay consumes messages coming from the Topic (Topic.Sample3). The Collaboration (crJMSXATopicSubscribe) subscribes to the JMS client properties on the Topic.Sample3 Topic. The JMS client properties is configured to use the internal Sun Microsystems JMS IQ Manager as the JMS server, and is responsible for displaying the message received to standard output, and then publishing the message to the external file (Inbound File).

In this case, the STCXAPublisherSLSessionBean acts as publisher to the Topic.Sample3 topic. This demonstrates transactionally publishing asynchronous messages from an EJB running in WebLogic to a Sun Microsystems JMS Topic.

Figure 51 JMSXATopicSubscribe Sample Components



The stand-alone remote client, com.stc.eways.ejb.sessionbean.xapublisher.STCXAPublisherSLSessionClient, is used to invoke the **createAccountAndPublish()** method of the STCXAPublisherSLSessionBean. This method takes two parameters:

- An account ID of type java.lang.String
- A balance of type double

The XA Session Bean inserts a record into the demo database and publishes to the topic with a message indicating that the record is successfully inserted into the database.

The parameters taken by the client are:

- The provider URL of the WebLogic JNDI where the Session Bean is bound
- The JNDI name of the Session Bean (SunMicrosystems.STCXAPublisherSLSessionBean)
- An account ID
- A balance for the account to create in the database

For example, the following command inserts a record into the database with the ID "JohnDoe" and a balance of 8888.99:

```
java com.stc.eways.ejb.sessionbean.xapublisher.STCXAPublisherSLSessionClient t3://localhost:7001
SunMicrosystems.STCXAPublisherSLSessionBean John 9888.99
```

Note: *Before running this client, make sure that the system classpath includes ejb.jar, weblogic.jar (with ejb.jar proceeding weblogic.jar in order), and weblogic.ejb.example.jar.*

After successfully inserting the record into the database and publishing to the topic, the remote client invokes the **getBalance()** method of the Session Bean and confirms that the record is successfully inserted. Note that getBalance does NOT confirm occurrence of a two phase commit. To see that both the database and Sun Microsystems JMS XA Resources are being used, look at the weblogic.log and Sun Microsystems JMS IQ Manager log. In addition, upon successfully publishing to the topic, the inbound File eWay writes a confirmation message to the file.

To simulate a rollback, pass an account ID of "rollback" in the command line for the remote client. For more details on the demoXAPool resource see **"examples-dataSource-demoXAPool" on page 111**.

Important: *XA transactions for the WebLogic eWay are managed by the WebLogic TransactionManager, NOT the eGate TransactionManager or in the eWay*

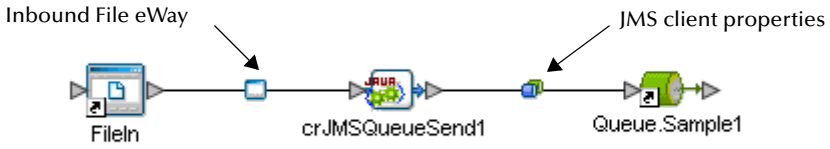
Connection parameters. For XA transactions make sure that the XAConnectionFactory(ies) are configured for the startup class.

8.4.4 The JMSQueueSend Sample

JMSQueueSend is an outbound sample project that demonstrates how to pass messages into a JMS queue asynchronously, before ultimately passing into a WebLogic container.

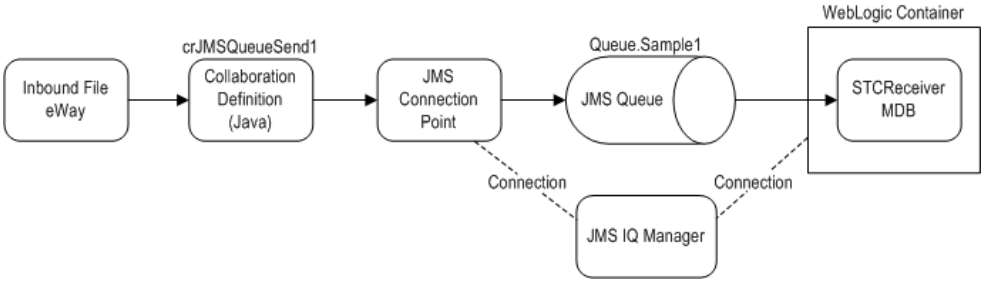
The Connectivity Map for this sample project appears as follows:

Figure 52 JMSQueSend Connectivity Map



In this sample, the Inbound File eWay feeds messages to the Queue (Queue.Sample1). The eWay looks for files with extension ".qfin" as input files (the input directory configured is C:\temp). The crJMSQueueSend1 Collaboration subscribes to the external data source and publishes to the Queue. The Collaboration is configured to use the internal Sun Microsystems JMS IQ Manager as the JMS server, and is responsible for copying data from the source event to the output event. The STCReceiverMDBean MDB receives messages from the Queue and displays the receiving message to the WebLogic console.

Figure 53 JMSQueueSend Sample Components



As seen in Figure 53, the Inbound File reads a file containing the input message event. A Collaboration subscribes to the external data source and publishes the input message to the JMS Queue. The JMS Connection is configured to use a JMS Queue and acts as a QueueSender. Both the JMS Connection and the MDB on WebLogic are configured to connect to the JMS IQ Manager as the JMS server. When WebLogic intercepts a JMS message, it delegates and dispatches the message to the MDB.

Note: *For more information on how to configure/deploy the MDB to use the Sun Microsystems JMS IQ Manager to drive the MDB, see* **"Sun Microsystems JMS" on page 35**.

8.4.5 The JMSTopicPublish Sample

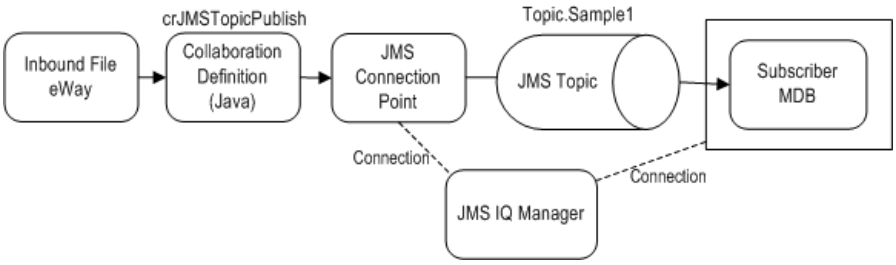
JMSTopicPublish is an outbound example that demonstrates how messages are read, subscribed and published to a JMS topic asynchronously, before passing into a WebLogic container.

The Connectivity Map for this sample appears as follows:

Figure 54 JMSTopicPublish Connectivity Map

In this sample, the Outbound File eWay publishes messages to the JMS Topic (Topic.Sample1). The Inbound File eWay looks for input files with the extension .tfin within the input directory C:\InputData. The Collaboration (crJMSTopicPublish) subscribes to this external data source and then publishes to the JMS Topic. The JMS client properties is configured to use the internal Sun Microsystems JMS IQ Manager as the JMS server, and is responsible for copying data from the source event to the output event. The STCSubscriberMDBean receives messages from the Topic.Sample1 Topic and displays the message it receives to the WebLogic console.

Figure 55 JMSTopicPublish Sample Components



As seen in Figure 55, the Outbound File eWay reads a file containing the input message event. Collaboration subscribes to the external data source and then publishes the input message, as a Topic.Sample1 event, to the JMS client properties. The JMS eWay Connection is configured to use a JMS Topic, acting as a TopicPublisher. Both the JMS Connection and the MDB are configured to connect to the JMS IQ Manager as the JMS server. The STCSubscriberMDB then receives the message, passed to it by the container, and displays the message in standard output (the WebLogic console).

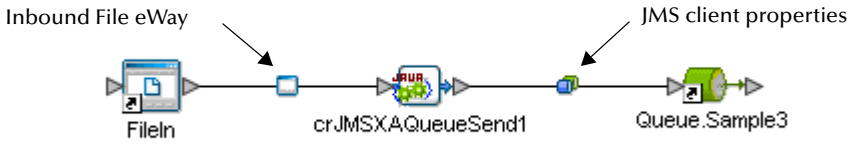
Note: *For more information on how to configure/deploy the MDB to use the Sun Microsystems JMS IQ Manager to drive the MDB, see* **"Sun Microsystems JMS" on page 35**.

8.4.6. The JMSXAQueueSend Sample

JMSXAQueueSend is an outbound sample project that demonstrates how to asynchronously pass two-phase commit protocol (XA) messages into a JMS queue, before ultimately passing into a WebLogic container.

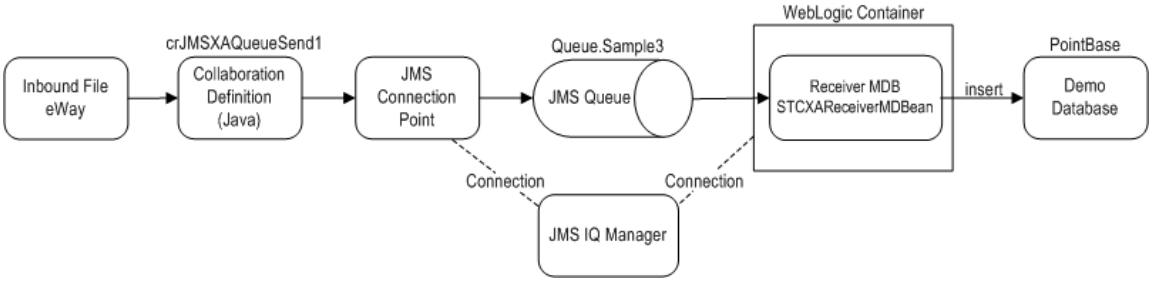
The Connectivity Map for tis sample project appears as follows:

Figure 56 JMSXAQueueSend Connectivity Map



In this sample, the Outbound File eWay feeds messages to the Queue (Queue.Sample3). The eWay looks for input files with the extension .xaqfin within the input directory C:\InputData. The Collaboration (crJMSXAQueueSend1) subscribes to this external data and publishes to the JMS Queue. The JMS client properties is configured to use the internal Sun Microsystems JMS IQ Manager as the JMS server, and is responsible for copying data from the source event to the output event.

Figure 57 JMSXAQueueSend Sample Components



As seen in Figure 57, the Outbound File eWay reads a file containing the input message event. The Collaboration subscribes to this external file and publishes the input message to the JMS Queue. The JMS client properties is configured to use a JMS Queue and therefore acts as a QueueSender. Both the JMS client properties and the MDB are configured to connect to the JMS IQ Manager as the JMS server. (For more information on how to configure/deploy the MDB to use the Sun Microsystems JMS IQ Manager to drive the MDB, see **"Sun Microsystems JMS" on page 35**.)

The STCXARecieverMDBean MDB receives the message in the format "accountID|balance," where accountID is a String account ID and balance is a numerical balance amount. The STCXAReceiverMDBean is configured to use the Sun Microsystems JMS XAResource and the PointBase sample demoXAPool to receive messages from Sun Microsystems JMS and write database records into the sample PointBase database table. Checking the database to see that the record is there does not necessarily confirm occurrence of the two stage commit.

Verify XA functionality by looking into the weblogic.log file for the examples domain, and also the Sun Microsystems IQ Manager log. For more information on how to effect proper logging, and to see XA at work, see **"Verifying XA At Work" on page 109**. XA

prepares and commits should be called on both database and Sun Microsystems JMS XA Resource. To simulate a rollback, pass an account ID of "rollback." For more details on the demoXAPool resource see **"examples-dataSource-demoXAPool" on page 111**. For details on the format of the input message for the feeder eWay see **"Sun Microsystems Sample Message Driven Beans" on page 91**.

Note: *Before running this client, be sure that the system classpath includes ejb.jar, weblogic.jar (with ejb.jar proceeding weblogic.jar in order), and weblogic.ejb.example.jar.*

The result of the test is that eGate sees the message that the remote client sent to the STCQueueRequestorSLSessionBean and the remote client sees the reply message constructed by the Java Collaboration from eGate.

Important: *XA transactions for the WebLogic eWay are managed by the WebLogic TransactionManager, NOT the eGate TransactionManager or in the eWay Connection parameters. For XA transactions make sure that the XAConnectionFactory(ies) are configured for the startup class.*

Note: *WebLogic will create a warning message, that XA is not supported, if a combination of XA and non-XA EJBs are loaded in the weblogic.ejb.example.jar file and the associated deployment descriptor files.*

Chapter 9

Sun Microsystems Sample Message Driven Beans

The previous sections, **"Java Naming and Directory Interface (JNDI)" on page 23** and **"Java Messaging Service (JMS)" on page 26** describe the JNDI and JMS subsystems. This chapter relates the concepts that were discussed in the previous sections with those regarding the Sun Microsystems Message Driven Beans (MDBs).

There are two MDBs that are deployed in WebLogic:

- MDB Subscribing to Sun Microsystems Topic
- MDB Subscribing to Sun Microsystems Queue.

In the following sections, there are references to two XML files. These files are used as the MDB's deployment descriptor. These are **ejb-jar.xml** and **weblogic-ejb-jar.xml**. The **ejb-jar.xml** deployment descriptor is specified by the EJB 2.0 specification. The **weblogic-ejb-jar.xml** is proprietary to WebLogic. Both are defined in order to deploy the MDB.

What's in This Chapter:

9.1 MDB Subscribing to a Sun Microsystems Topic

This MDB subscribes to a Sun Microsystems JMS Topic. It receives from ONLY ONE Sun Microsystems Topic. The MDB simply receives and displays the JMS messages.

ejb-jar.xml

The following is the deployment descriptor for this MDB (ejb-jar.xml):

```
<ejb-jar>
    <enterprise-beans>
        <message-driven>
            <ejb-name>STCSubscriberMDBean</ejb-name>
```

```
                <ejb-
class>com.stc.eways.ejb.messagebean.STCSubscriberMDBean</ejb-class>
                <transaction-type>Bean</transaction-type>
                <acknowledge-mode>Auto-acknowledge</acknowledge-mode>
                <message-driven-destination>
                    <destination-type>javax.jms.Topic</destination-type>
                    <subscription-durability>Durable</subscription-
durability>
                </message-driven-destination>
            </message-driven>
        ...

        </enterprise-beans>

        <assembly-descriptor>
            <container-transaction>
                <method>
                    <ejb-name>STCXAReceiverMDBean</ejb-name>
                    <method-name>*</method-name>
                </method>
                <trans-attribute>Required</trans-attribute>
            </container-transaction>
            <container-transaction>
                <method>
                    <ejb-name>STCXAPublisherSLSessionBean</ejb-name>
                    <method-name>createAccountAndPublish</method-name>
                </method>
                <method>
                    <ejb-name>STCXAPublisherSLSessionBean</ejb-name>
                    <method-name>getBalance</method-name>
                </method>
                <trans-attribute>Required</trans-attribute>
            </container-transaction>
        </assembly-descriptor>
    </ejb-jar>
```

<ejb-name> Tag

The <ejb-name> defines the name of the MDB and is used to uniquely identify the MDB by the container. This name is displayed in the WebLogic Administrative Console to identify this MDB.

<ejb-class> Tag

The <ejb-class> tag defines the class that implements that MDB. The class that implements the Topic subscribing MDB is **com.stc.eways.ejb.messagebean.STCSubscriberMDBean**.

<destination-type> Tag

Since this MDB is subscribing to a Sun Microsystems Topic, the <destination-type> is specified as **javax.jms.Topic**.

<subscription-durability> Tag

In order to create a durable subscriber MDB, the <subscription-durability> is specified as **Durable**.

WebLogic-ejb-jar.xml

In addition to the **ejb-jar.xml** file, the MDB also needs to be included in the **WebLogic-ejb-jar.xml** file.

```
<weblogic-ejb-jar>
    <weblogic-enterprise-bean>
        <ejb-name>STCSubscriberMDBean</ejb-name>
        <message-driven-descriptor>
            <pool>
                <max-beans-in-free-pool>15</max-beans-in-free-pool>
                <initial-beans-in-free-pool>5</initial-beans-in-
free-pool>
            </pool>
            <destination-jndi-
name>SunMicrosystems.Topics.STCTopic1</destination-jndi-name>
            <initial-context-
factory>weblogic.jndi.WLInitialContextFactory</initial-context-
factory>
            <provider-url>t3://localhost:7001</provider-url>
            <connection-factory-jndi-
name>SunMicrosystems.TopicConnectionFactories.TopicConnectionFactory<
/connection-factory-jndi-name>
        </message-driven-descriptor>
        <jndi-name>SunMicrosystems.STCSubscriberMDBean</jndi-name>
    </weblogic-enterprise-bean>
    ...

</weblogic-ejb-jar>
```

<ejb-name> Tag

The value for <ejb-name> must match that defined in ejb-jar.xml.

<pool> Tag

The <pool> tag defines the maximum number of MDBs in the free pool and the initial pool size by using the <max-beans-in-free-pool> and <initial-beans-in-free-pool> tags respectively.

<destination-jndi-name> Tag

The <destination-jndi-name> tells the container the JNDI name of the Sun Microsystems Topic that this MDB is to subscribe.

<connection-factory-jndi-name> Tag

Also, the <connection-factory-jndi-name> specifies the **TopicConnectionFactory** to use. The Topic and **TopicConnectionFactory** must have already been created and registered with JNDI by the startup class. (See **"Sun Microsystems WebLogic Startup Class" on page 43** for details.) The container locates these JNDI objects in its own JNDI as specified by the <initial-context-factory> and <provider-url>.

9.2 MDB Subscribing to Sun Microsystems Queue

This MDB subscribes to only one Sun Microsystems JMS Queue and simply receives and displays the JMS Messages.

ejb-jar.xml

The following is the deployment descriptor for this MDB (ejb-jar.xml):

```
<ejb-jar>
    <enterprise-beans>
        ...
        <message-driven>
            <ejb-name>STCReceiverMDBean</ejb-name>
            <ejb-
class>com.stc.eways.ejb.messagebean.STCReceiverMDBean</ejb-class>
            <transaction-type>Bean</transaction-type>
            <message-driven-destination>
                <destination-type>javax.jms.Queue</destination-type>
                <subscription-durability>Durable</subscription-
durability>
            </message-driven-destination>
        </message-driven>
        ...
    <assembly-descriptor>
        ...
        <container-transaction>
            <method>
                <ejb-name>STCXAReceiverMDBean</ejb-name>
                <method-name>*</method-name>
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>

        ...
    </assembly-descriptor>

</ejb-jar>
```

<ejb-name> Tag

The <ejb-name> defines the name of the MDB and is used to uniquely identify the MDB by the container. This name is displayed in the WebLogic Administrative Console to identify this MDB.

<ejb-class> Tag

The <ejb-class> tag defines the class that implements that MDB. The class that implements the Queue subscribing MDB is **com.stc.eways.ejb.messagebean.STCReceiverMDBean**.

<destination-type> Tag

Since this MDB is subscribing to a Sun Microsystems Queue, you must specify the <destination-type> tag as javax.jms.Queue.

<subscription-durability> Tag

In order to create a durable subscriber MDB, the <subscription-durability> tag is specified as **Durable**.

weblogic-ejb-jar.xml

In addition to the **ejb-jar.xml** file, the MDB also needs to be included in the **weblogic-ejb-jar.xml** file:

```
<weblogic-ejb-jar>
```

```
    <weblogic-enterprise-bean>
        <ejb-name>STCReceiverMDBean</ejb-name>
        <message-driven-descriptor>
            <pool>
                <max-beans-in-free-pool>15</max-beans-in-free-pool>
                <initial-beans-in-free-pool>5</initial-beans-in-
free-pool>
            </pool>
            <destination-jndi-
name>SunMicrosystems.Queues.STCQueue1</destination-jndi-name>
            <initial-context-
factory>weblogic.jndi.WLInitialContextFactory</initial-context-
factory>
            <provider-url>t3://localhost:7001</provider-url>
            <connection-factory-jndi-
name>SunMicrosystems.QueueConnectionFactories.QueueConnectionFactory<
/connection-factory-jndi-name>
        </message-driven-descriptor>
        <jndi-name>SunMicrosystems.STCReceiverMDBean</jndi-name>
    </weblogic-enterprise-bean>
    ...

</weblogic-ejb-jar>
```

<ejb-name> Tag

The value for <ejb-name> tag must match that defined in **ejb-jar.xml**.

<pool> Tag

The <pool> tag defines the maximum number of MDBs in the free pool and the initial pool size by using the <max-beans-in-free-pool> and <initial-beans-in-free-pool> tags respectively.

<destination-jndi-name> Tag

The <destination-jndi-name> tag tells the container the JNDI name of the Sun Microsystems Queue that this MDB is to subscribe.

<connection-factory-jndi-name> Tag

The <connection-factory-jndi-name> specifies the **QueueConnectionFactory** to use. The Queue and **QueueConnectionFactory** must have already been created and registered with JNDI by the startup class. (See **"Sun Microsystems WebLogic Startup Class" on page 43** for details.) The container locates these JNDI objects in its own JNDI as specified by the <initial-context-factory> and <provider-url>.

9.3 Accessing Session Beans

Session Beans can be accessed from an eGate Collaboration by using the EJB OTD Builder to create an OTD for the Session Bean. This is done by:

1. Using **Create** on the home interface to create a remote instance.
2. Call methods on the remote instance.
3. Free resources by calling **remove()** when finished.

9.3.1. Sun Microsystems Sample Session Beans

There are two Stateless Session Beans available with the WebLogic eWay:

- A Session Bean that publishes to a Sun Microsystems JMS Topic
- A Session Bean that uses the **STCQueueRequestor** to send and receive a message to and from Sun Microsystems JMS.

In the following sections, there are references to two XML files: **ejb-jar.xml** and **weblogic-ejb-jar.xml.** These files are used as the Session Bean's deployment descriptor. The **ejb-jar.xml** deployment descriptor is specified by the EJB 2.0 specification. The **weblogic-ejb-jar.xml** is proprietary to WebLogic. Both need to define in order to deploy the MDBs.

9.3.2. SLS Bean Publishing to Sun Microsystems Topic

This Stateless Session Bean publishes to a Sun Microsystems JMS Topic. It exposes the remote method, **publish()**, which takes a String as an argument. The Session Bean gets the message and publishes the message to a Sun Microsystems JMS Topic.

ejb-jar.xml

The following is the deployment descriptor for this Session Bean (ejb-jar.xml):

```
<ejb-jar>
    <enterprise-beans>
        ...
        <session>
            <ejb-name>STCPublisherSLSessionBean</ejb-name>

<home>com.stc.eways.ejb.sessionbean.publisher.STCPublisherSLSessionHo
me</home>

<remote>com.stc.eways.ejb.sessionbean.publisher.STCPublisherSLSession
</remote>
            <ejb-
class>com.stc.eways.ejb.sessionbean.publisher.STCPublisherSLSessionBe
an</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Bean</transaction-type>
            <resource-ref>
                <res-ref-name>jms/TopicConnectionFactory</res-ref-
name>
                <res-type>javax.jms.TopicConnectionFactory</res-
type>
                <res-auth>Application</res-auth>
            </resource-ref>
            <resource-env-ref>
                <resource-env-ref-name>jms/Topic</resource-env-ref-
name>
                <resource-env-ref-type>javax.jms.Topic</resource-
env-ref-type>
            </resource-env-ref>
        </session>
        ...

</ejb-jar>
```

<ejb-name> Tag

The <ejb-name> tag defines the name of the Stateless Session Bean and is used to uniquely identify the Session Bean by the container. This name is displayed in the WebLogic Administrative Console to identify this Bean.

<ejb-class> Tag

The <ejb-class> tag defines the class that implements that Session Bean. The home interface for this bean is:

com.stc.eways.ejb.sessionbean.publisher.STCPublisherSLSessionHome

The remote interface for the bean is:

com.stc.eways.ejb.sessionbean.publisher.STCPublisherSLSession

The class which implements the home and remote interfaces as well as the bean itself is:

com.stc.eways.ejb.sessionbean.publisher.STCPublisherSLSessionBean

The Session Bean knows about the **TopicConnectionFactory** and Topic destinations via the resource reference tags. Notice that the value for the res-ref-name tag is **jms/TopicConnectionFactory** and the value for the resource-env-ref-name environment entry is **jms/Topic**. They are specified as **javax.jms.TopicConnectionFactory** and **javax.jms.Topic** for the resource type respectively. These resource references are another level of JNDI indirection. They don't specify the actual JNDI names of the JMS objects, but rather they are references to the JNDI name. So the EJB can reference **jms/TopicConnectionFactory** but does not really care what the actual JNDI name is. The actual JNDI names for these references are defined in the **weblogic-ejb-jar.xml** file.

weblogic-ejb-jar.xml

In addition to the **ejb-jar.xml** file, the Session Bean also needs to be included in the **weblogic-ejb-jar.xml** file:

```
<weblogic-ejb-jar>
    <weblogic-enterprise-bean>
        <ejb-name>STCPublisherSLSessionBean</ejb-name>
        <stateless-session-descriptor>
            <pool>
                <max-beans-in-free-pool>15</max-beans-in-free-pool>
                <initial-beans-in-free-pool>5</initial-beans-in-
free-pool>
            </pool>
        </stateless-session-descriptor>
        <reference-descriptor>
            <resource-description>
                <res-ref-name>jms/TopicConnectionFactory</res-ref-
name>
                <jndi-
name>SunMicrosystems.TopicConnectionFactories.TopicConnectionFactory<
/jndi-name>
            </resource-description>
            <resource-env-description>
                <res-env-ref-name>jms/Topic</res-env-ref-name>
                <jndi-name>SunMicrosystems.Topics.STCTopic2</jndi-
name>
            </resource-env-description>
        </reference-descriptor>
```

```
            <jndi-name>SunMicrosystems.STCPublisherSLSessionBean</jndi-
name>
        <weblogic-enterprise-bean>
        ...

    </weblogic-ejb-jar>
```

<ejb-name> Tag

The value for <ejb-name> tag must match that defined in **ejb-jar.xml**.

<pool> Tag

The <pool> tag defines the maximum number of Session Beans in the free pool and the initial pool size by using the <max-beans-in-free-pool> and <initial-beans-in-free-pool> tags respectively.

<jndi-name> Tag

The value for the <jndi-name> tag for the resource name **jms/TopicConnectionFactory** is:

SunMicrosystems.TopicConnectionFactories.TopicConnectionFactory

The value for the <jndi-name> tag for the **jms/Topic** entry is:

SunMicrosystems.Topics.STCTopic2

These values define the resource reference name to JNDI name mappings. The Topic and **TopicConnectionFactory** must have already been created and registered with JNDI by the startup class. (See **"Sun Microsystems WebLogic Startup Class" on page 43** for details.) The container locates these JNDI objects in its own JNDI as specified by the <initial-context-factory> and <provider-url>.

SLS Bean Request/Reply To Sun Microsystems Queue

This Stateless Session Bean sends a request to the Sun Microsystems JMS Queue and receives a reply on the request sent. It exposes the remote method, **request()**, which takes a String as an argument. The Session Bean receives the message and sends it to a Sun Microsystems JMS Queue. The Session Bean then gets a reply from eGate.

ejb-jar.xml

The following is the deployment descriptor for this MDB (ejb-jar.xml):

```
<ejb-jar>
    <enterprise-beans>
        ...
        <session>
            <ejb-name>STCQueueRequestorSLSessionBean</ejb-name>

<home>com.stc.eways.ejb.sessionbean.queuerequestor.STCQueueRequestorS
LSessionHome</home>

<remote>com.stc.eways.ejb.sessionbean.queuerequestor.STCQueueRequesto
rSLSession</remote>
            <ejb-
class>com.stc.eways.ejb.sessionbean.queuerequestor.STCQueueRequestorS
LSessionBean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Bean</transaction-type>
            <env-entry>
```

```
                    <env-entry-name>ReceiveTimeout</env-entry-name>
                    <env-entry-type>java.lang.Long</env-entry-type>
                    <env-entry-value>60000</env-entry-value>
                </env-entry>
                <resource-ref>
                    <res-ref-name>jms/QueueConnectionFactory</res-ref-
    name>
                    <res-type>javax.jms.QueueConnectionFactory</res-
    type>
                    <res-auth>Application</res-auth>
                </resource-ref>
                <resource-env-ref>
                    <resource-env-ref-name>jms/Queue</resource-env-ref-
    name>
                    <resource-env-ref-type>javax.jms.Queue</resource-
    env-ref-type>
                </resource-env-ref>
            </session>
            ...

    </ejb-jar>
```

<ejb-name> Tag

The <ejb-name> tag defines the name of the Stateless Session Bean and is used to uniquely identify the Session Bean by the container. This name is displayed in the WebLogic Administrative Console to identify this Bean.

<ejb-class> Tag

The <ejb-class> tag defines the class that implements the Session Bean. The home interface for this bean is:

com.stc.eways.ejb.sessionbean.publisher.STCPublisherSLSessionHome

The remote interface for the bean is:

com.stc.eways.ejb.sessionbean.publisher.STCPublisherSLSession

The class which implements the home and remote interfaces as well as the bean itself is:

com.stc.eways.ejb.sessionbean.publisher.STCPublisherSLSessionBean

The Session Bean knows about the **QueueConnectionFactory** and Queue destinations via the resource reference tags.

<res-ref-name> and <resource-env-ref-name> Tags

The value for the <res-ref-name> tag is **jms/QueueConnectionFactory** and the value for the <resource-env-ref-name> environment entry is **jms/Queue**. The EJB can reference **jms/QueueConnectionFactory** but is not concerned with what the actual JNDI name is. They are specified as: **javax.jms.QueueConnectionFactory** and **javax.jms.Queue** for the resource type respectively. These resource references are another level of JNDI indirection. They don't specify the actual JNDI names of the JMS objects, but rather they are references to the JNDI name. So the EJB can reference jms/QueueConnectionFactory but does not really care what the actual JNDI name is. The actual JNDI names for these references are defined in the **weblogic-ejb-jar.xml** file.

weblogic-ejb-jar.xml

In addition to the **ejb-jar.xml** file, the Session Bean also needs to be included in the **weblogic-ejb-jar.xml** file:

```
<weblogic-ejb-jar>
    <weblogic-enterprise-bean>
        <ejb-name>STCQueueRequestorSLSessionBean</ejb-name>
        <stateless-session-descriptor>
            <pool>
                <max-beans-in-free-pool>15</max-beans-in-free-pool>
                <initial-beans-in-free-pool>5</initial-beans-in-
free-pool>
            </pool>
        </stateless-session-descriptor>
        <reference-descriptor>
            <resource-description>
                <res-ref-name>jms/QueueConnectionFactory</res-ref-
name>
                <jndi-
name>SunMicrosystems.QueueConnectionFactories.QueueConnectionFactory<
/jndi-name>
            </resource-description>
            <resource-env-description>
                <res-env-ref-name>jms/Queue</res-env-ref-name>
                <jndi-name>SunMicrosystems.Queues.STCQueue2</jndi-
name>
            </resource-env-description>
        </reference-descriptor>
        <jndi-name>SunMicrosystems.STCQueueRequestorSLSessionBean</
jndi-name>
    </weblogic-enterprise-bean>
    ...

</weblogic-ejb-jar>
```

<ejb-name> Tag

The value for <ejb-name> must match that defined in **ejb-jar.xml**.

<pool> Tag

The <pool> tag defines the maximum number of Session Beans in the free pool and the initial pool size by using the <max-beans-in-free-pool> and <initial-beans-in-free-pool> tags respectively.

<jndi-name> Tag

The value for the <jndi-name> tag for the resource name **jms/QueueConnectionFactory** is:

SunMicrosystems.QueueConnectionFactories.QueueConnectionFactory

The value for the <jndi-name> tag for the **jms/Queue** entry is:

SunMicrosystems.Queues.STCQueue2

These values define the resource reference name to JNDI name mappings. The Queue and **QueueConnectionFactory** must have already been created and registered with JNDI by the startup class. (See **"Sun Microsystems WebLogic Startup Class" on page 43** for details.) The container locates these JNDI objects in its own JNDI as specified by the <initial-context-factory> and <provider-url>.

9.3.3. Lazy Loading

The following code is for the **publish()** method of the sample Topic Publisher Session Bean. **initialize()** is called in order to create the necessary JMS connections to publish to the JMS Topic. This process is known as "lazy loading." Lazy loading is used because JMS objects may not have been bound to the naming service during the deployment of the EJB. This is because the Sun Microsystems WebLogic startup class can not be deployed prior to the EJB. Therefore, it may not be guaranteed that calling **initalize()** in **ejbCreate()** creates the JMS Topic connection. WebLogic does not allow the user to specify the deployment of a startup class prior to the deployment of an EJB.

```
/**
     * Send a text message to Sun Microsystems JMS Topic.
     *
     * @param       message             The text message to send to a
JMS Topic.
     *
     * @throws      EJBException        Upon error.
     *
     * @author      SunMicrosystems
     */
    public void publish (String message) throws EJBException
    {
        // If not initialized already then do it (lazy loading)
        initialize();

        if (message == null)
           throw new EJBException ("Can not publish a null message.");

        try
        {
            TextMessage textMsg =
sbynJMSTopicObject.createTextMessage(message);
            sbynJMSTopicObject.publish(textMsg);
        }
        catch (Exception ex)
        {
           throw new EJBException ("Exception caught while publishing
message; exception : " + ex.toString());
        }
    }
```

The following code is for initialize(). Notice that the EJB's ENC is used for getting the **TopicConnectionFactory** and Topic destination. See the sample Java source code for details.

```
    protected void initialize () throws EJBException
    {
        if (!bInitialized)
        {
            Exception savedException = null;

            try
            {
                // Get the InitialContext
                jndiInitialContext = new InitialContext();

                // Get the TopicConnectionFactory using JNDI ENC
```

```
                TopicConnectionFactory tcf =
(TopicConnectionFactory)jndiInitialContext.lookup("java:comp/env/" +
ENV_TOPIC_CONNECTION_FACTORY);

                // Get the Topic using JNDI ENC
                Topic topic =
(Topic)jndiInitialContext.lookup("java:comp/env/" +
ENV_TOPIC_DESTINATION);

                // Create our JMSTopic object
           sbynJMSTopicObject = new JMSTopicObject (tcf, topic);

                bInitialized = true;
            }
            catch (Exception ex1)
            {
                throw new EJBException(ex1);
            }
        }
    }
```

Accessing Entity Beans

Entity Beans can be accessed from an eGate Collaboration by using the EJB OTD Builder to create an OTD for the Session Bean. This is done by:

1. Using Creators or Finders on the home interface to create remote instances.
2. Using **hasNext()** and **next()** to access the instance.
3. Call methods on the remote instance.

By calling "remove", the Entity Bean instance is removed from the permanent storage, for example deleting an account from a database (or databases).

9.4 Sun Microsystems Sample XA Message Driven Beans

A MDB can subscribe to a Sun Microsystems JMS Topic or Queue in an XA transaction. If the transaction needs to roll back, the message received by the MDB is rolled back and re-delivered to the MDB.

MDB Subscribing to Sun Microsystems JMS Queue Transactionally

The MDB subscribes to a (single) Sun Microsystems JMS Queue. This MDB uses Container Managed Transaction. Because the WebLogic container optimizes to one-phase commit (or rollback) if only one XA resource is used, the MDB must also be configured to use another XA Resource in order to observe a two-phase commit (or rollback). Therefore, in addition to the Sun Microsystems JMS XAResource, the MDB is also deployed to use the demo XA database resource pool. The "examples" WebLogic Server instance already has a XA database resource pool configured. The pool's JNDI name is **examples-dataSource-demoXAPool**. The MDB references this pool (see **"examples-dataSource-demoXAPool" on page 111** for more information). The MDB expects the JMS TextMessage to contain, in its body content, a text string that looks like the following:

```
accountId|balance
```

where **accountId** is a String ID for the account to create in the database and **balance** is the initial balance of the account to be created.

The MDB parses these values separated by the "|" (pipe) character. If a XA commit occurs successfully, both the *JMS Message receive* and the *insert into the database* get committed. To simulate an XA rollback, create a JMS Message with an accountId of **rollback**. The MDB throws an EJBException (or any EJB SystemException), if it sees rollback as the accountId, after preparing to insert into the database table. Throwing EJBException causes the XA rollback to happen on both the database and the Sun Microsystems JMS Queue. Upon rollback, the JMS Message is again delivered to the MDB. The MDB can't keep any state; therefore, in order to determine whether the rollback message has been sent again, it checks the **JMSRedelivered** flag on the JMS Message it received. If the **JMSRedelivered** flag is set to true, the MDB does not open a connection to the database or throw any exceptions. By not throwing an exception on a rollback message that is being resent, a one-phase commit on the JMS Queue occurs. The MDB must check the JMSRedelivered flag in order to prevent indefinite rollbacks.

ejb-jar.xml

The following is the deployment descriptor for this MDB (ejb-jar.xml):

```
<ejb-jar>
    <enterprise-beans>
        <message-driven>
            <ejb-name>STCXAReceiverMDBean</ejb-name>
            <ejb-
class>com.stc.eways.ejb.messagebean.STCXAReceiverMDBean</ejb-class>
            <transaction-type>Container</transaction-type>
            <message-driven-destination>
                <destination-type>javax.jms.Queue</destination-type>
                <subscription-durability>Durable</subscription-
durability>
            </message-driven-destination>
            <resource-ref>
                <res-ref-name>jdbc/demoXAPool</res-ref-name>
                <res-type>javax.sql.DataSource</res-type>
                <res-auth>Container</res-auth>
            </resource-ref>
        </message-driven>
            ...
    </enterprise-beans>
    <assembly-descriptor>
        <container-transaction>
            <method>
                <ejb-name>STCXAReceiverMDBean</ejb-name>
                <method-name>*</method-name>
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>

        ...
    </assembly-descriptor>

</ejb-jar>
```

Notice that MDB references another resource by the reference name **jdbc/demoXAPool**. This resource is of type **javax.sql.DataSource**. The actual JNDI name of this resource is defined in the weblogic-ejb-jar.xml deployment descriptor. Notice, also,

that the CMT (Container Managed Transaction) is specified in the <transaction-type> for the MDB. It is also required that the <container-transaction> be specified for the MDB in the <assembly-descriptor> tag. In <container-transaction>, it's specified that all methods (including the **onMessage()** method) are required to participate in an XA transaction. This is done by setting <trans-attribute> to "Required" and the <method> tag with <ejb-name> set to the name of the MDB and <method-name> set to * (used as a wildcard to signify all methods).

weblogic-ejb-jar.xml

In addition to the **ejb-jar.xml** file, the MDB also needs to be included in the **weblogic-ejb-jar.xml** file:

```
<weblogic-ejb-jar>
    <weblogic-enterprise-bean>
        <ejb-name>STCXAReceiverMDBean</ejb-name>
        <message-driven-descriptor>
            <pool>
                <max-beans-in-free-pool>15</max-beans-in-free-pool>
                <initial-beans-in-free-pool>5</initial-beans-in-
free-pool>
            </pool>
            <destination-jndi-
name>SunMicrosystems.Queues.STCQueue3</destination-jndi-name>
            <initial-context-
factory>weblogic.jndi.WLInitialContextFactory</initial-context-
factory>
            <provider-url>t3://localhost:7001</provider-url>
            <connection-factory-jndi-
name>SunMicrosystems.QueueConnectionFactories.XAQueueConnectionFactor
y</connection-factory-jndi-name>
        </message-driven-descriptor>
        <reference-descriptor>
            <resource-description>
                <res-ref-name>jdbc/demoXAPool</res-ref-name>
                <jndi-name>examples-dataSource-demoXAPool</jndi-
name>
            </resource-description>
        </reference-descriptor>
        <jndi-name>SunMicrosystems.STCXAReceiverMDBean</jndi-name>
    </weblogic-enterprise-bean>
    ...

</weblogic-ejb-jar>
```

The value for <ejb-name> must match the value defined in **ejb-jar.xml**.

<pool> Tag

The <pool> tag defines the maximum number of MDBs in the free pool and the initial pool size by using the <max-beans-in-free-pool> and <initial-beans-in-free-pool> tags respectively.

<destination-jndi-name> Tag

The <destination-jndi-name> tag tells the container the JNDI name of the Sun Microsystems Queue to which this MDB is to subscribe.

<connection-factory-jndi-name> Tag

The <connection-factory-jndi-name> tag specifies the **XAQueueConnectionFactory** to use. The Queue and **XAQueueConnectionFactory** must already be created and

registered with JNDI by the startup class. (See **"Sun Microsystems WebLogic Startup Class" on page 43** for details.) The container locates these JNDI objects in its own JNDI as specified by the <initial-context-factory> and <provider-url>. Notice also that the actual JNDI name for the **jdbc/demoXAPool** resource is **examples-dataSource-demoXAPool**. This is the JNDI name of the datasource XA pool that is already created and configured for the "examples" WebLogic Server when WebLogic is installed.

9.4.1. Sun Microsystems Sample XA Session Beans

A Session Bean (Stateless or Stateful) can publish a message to a Sun Microsystems JMS Topic or send a message to a Sun Microsystems JMS Queue in an XA transaction. The Session Bean accesses the Sun Microsystems **JMS XAConnectionFactory** and Destination via the Bean's Environment Naming Context (ENC). The **XAConnectionFactory** and Destination are denoted using the following tags of the Bean's deployment descriptor:

- <resource-ref>
- <resource-env-ref>
- <resource-ref-name>
- <resource-env-ref-name>

The Session Bean must enlist the Sun Microsystems JMS XA Resource to WebLogic TransactionManager. The enlistment must be done to the current XA transaction created by the WebLogic container.

How To Enlist Sun Microsystems JMS XAResource

WebLogic provides a helper class, **weblogic.transaction.TxHelper**, which the EJB developer can use to get a hold of the current transaction and to enlist the Sun Microsystems JMS XA Resource to the current transaction. The enlistment process can be done in the Bean's ejbCreate method(s). The Session Bean relies on the Sun Microsystems Startup Class (see Sun Microsystems WebLogic Startup Class) to create and bind the JMS **XAConnectionFactory** and Destination prior to WebLogic deploying the EJBs. Because WebLogic does not allow startup classes to be deployed prior to EJBs, the sample EJBs do "lazy loading" of the JMS objects. EJBs should only lookup the JMS objects once they are created or during initialization. This can be done only when the EJB is ready to publish or send a message(s) to a destination.

In the usual manner, use the **XAConnectionFactory** and Destination to create the XAConnection and XASession. The Bean can get a hold of the **XAConnectionFactory** and Destination via the Bean's ENC. Once the XASession has been created, get a reference to the XAResource by calling **XASession.getXAResource()**; then enlist the XAResource to the current transaction. Before you enlist, call the WebLogic static method, **TxHelper.getTransaction**, to get a reference to the current transaction allocated by the container. **TxHelper.getTransaction** returns a **javax.transaction.Transaction**. You can then call **javax.transaction.Transaction.enlistResource** passing in the XAResource retrieved for the XASession that you had created.

SLS Bean Publishing to Sun Microsystems JMS Topic Transactionally

This Stateless Session Bean publishes to a Sun Microsystems JMS Topic transactionally. The sample Session Bean uses CMT (Container Managed Transaction). As with the transactional MDB, the Session Bean also utilizes two XA Resources in order to exhibit a two-phase commit or rollback behavior. The sample Session Bean uses both the Sun Microsystems JMS XAResource and the demo XA database resource pool.(See **"examples-dataSource-demoXAPool" on page 111** for details.) This Session Bean exposes two remote methods:

- createAccountAndPublish()
- getBalance()

The **createAccountAndPublish()** method takes two parameters: **accountId** of type java.lang.String and **balance** of type double. This method inserts a new record into a table of the demo database and publishes a JMS Message to a Sun Microsystems JMS Topic upon successfully inserting the record into the table. Both the insert and the publish are treated as a single XA transaction.

The **getBalance()** method accesses the database and retrieves the balance for the record specified by the account ID, passed to the method as argument. This method can be used to verify that a particular record has been successfully inserted into the database by the **createAccountAndPublish()** method. In fact, the remote client tester for this Session Bean does invoke **createAccountAndPublish()** and then invokes the **getBalance()** method immediately after the **createAccountAndPublish()** method invocation returns. Upon successful commit of the XA transaction, both the insert to the database table and the publish to the Sun Microsystems JMS Topic are committed. The **getBalance()** method returns the correct balance and eGate receives the published message.

To simulate an XA rollback, the remote client can pass in an accountId of **rollback** in the **createAccountAndPublish()** remote method call. The Session Bean prepares to insert the record to the database and prepares to publish to the Sun Microsystems JMS Topic. Finally, it checks whether the accountId is "rollback." If it is, the Session Bean throws an EJBException (or any EJB SystemException) so that the container calls rollback on both XA resources. When the client calls **getBalance()**, passing in an accountId of rollback, the client will see that this record is not inserted. Moreover, eGate does not receive the rollback message.

ejb-jar.xml Tag

The following is the deployment descriptor for this Session Bean (ejb-jar.xml):

```
<ejb-jar>
    <enterprise-beans>
        ...
        <session>
            <ejb-name>STCXAPublisherSLSessionBean</ejb-name>

<home>com.stc.eways.ejb.sessionbean.xapublisher.STCXAPublisherSLSessi
onHome</home>

<remote>com.stc.eways.ejb.sessionbean.xapublisher.STCXAPublisherSLSes
sion</remote>
```

```
                <ejb-
class>com.stc.eways.ejb.sessionbean.xapublisher.STCXAPublisherSLSessi
onBean</ejb-class>
                <session-type>Stateless</session-type>
                <transaction-type>Container</transaction-type>
                <resource-ref>
                    <res-ref-name>jms/XATopicConnectionFactory</res-ref-
name>
                    <res-type>javax.jms.XATopicConnectionFactory</res-
type>
                    <res-auth>Container</res-auth>
                </resource-ref>
                <resource-ref>
                    <res-ref-name>jdbc/demoXAPool</res-ref-name>
                    <res-type>javax.sql.DataSource</res-type>
                    <res-auth>Container</res-auth>
                </resource-ref>
                <resource-env-ref>
                    <resource-env-ref-name>jms/Topic</resource-env-ref-
name>
                    <resource-env-ref-type>javax.jms.Topic</resource-
env-ref-type>
                </resource-env-ref>
            </session>
            ...
        </enterprise-beans>
        <assembly-descriptor>
            <container-transaction>
                <method>
                    <ejb-name>STCXAPublisherSLSessionBean</ejb-name>
                    <method-name>createAccountAndPublish</method-name>
                </method>
                <method>
                    <ejb-name>STCXAPublisherSLSessionBean</ejb-name>
                    <method-name>getBalance</method-name>
                </method>
                <trans-attribute>Required</trans-attribute>
            </container-transaction>
            ...
        </assembly-descriptor>

    </ejb-jar>
```

<ejb-name> Tag

The <ejb-name> defines the name of the Stateless Session Bean and is used to uniquely identify the Session Bean by the container. This name is displayed in the WebLogic Administrative Console to identify this Bean.

<ejb-class> Tag

The <ejb-class> tag defines the class that implements that Session Bean. The home interface for this bean is:

com.stc.eways.ejb.sessionbean.publisher.STCXAPublisherSLSessionHome

The remote interface for the bean is:

com.stc.eways.ejb.sessionbean.publisher.STCXAPublisherSLSession

The class which implements the home and remote interfaces as well as the bean itself is:

com.stc.eways.ejb.sessionbean.publisher.STCXAPublisherSLSessionBean

The Session Bean is aware of the **XATopicConnectionFactory** and Topic destinations via the resource reference tags.

<res-ref-name> and <resource-env-ref-name> Tag

The value for the <res-ref-name> tag is **jms/XATopicConnectionFactory** and the value for the <resource-env-ref-name> environment entry is **jms/Topic**. They are specified as **javax.jms.XATopicConnectionFactory** and **javax.jms.Topic** for the resource type respectively. These resource references are another level of JNDI indirection. They don't specify the actual JNDI names of the JMS objects, but rather they are references to the JNDI name. The EJB can reference j**ms/XATopicConnectionFactory** but does not really care what the actual JNDI name is. The actual JNDI names for these references are defined in the **weblogic-ejb-jar.xml** file.

Notice that the SLS Bean references another resource by the reference name **jdbc/demoXAPool**. This resource is of type **javax.sql.DataSource**. The actual JNDI name of this resource is defined in the **weblogic-ejb-jar.xml** deployment descriptor.

The CMT is specified in the <transaction-type> for the SLS Bean. It is also required that the <container-transaction> be specified for the SLS Bean in the <assembly-descriptor> tag. In <container-transaction>, it's specified that the methods **createAccountAndPublish** and **getBalance** are required to participate in an XA transaction. Although **getBalance** is marked as required, the container optimizes for a one-phase commit or rollback because it only accesses one XA Resource (the database XA Resource).

weblogic-ejb-jar.xml

In addition to the **ejb-jar.xml** file, the Session Bean must also be included in the **weblogic-ejb-jar.xml file**:

```
<weblogic-ejb-jar>
         <ejb-name>STCXAPublisherSLSessionBean</ejb-name>
         <stateless-session-descriptor>
              <pool>
                   <max-beans-in-free-pool>15</max-beans-in-free-pool>
                   <initial-beans-in-free-pool>5</initial-beans-in-
free-pool>
              </pool>
         </stateless-session-descriptor>
         <reference-descriptor>
              <resource-description>
                   <res-ref-name>jms/XATopicConnectionFactory</res-ref-
name>
                   <jndi-
name>SunMicrosystems.TopicConnectionFactories.XATopicConnectionFactor
y</jndi-name>
              </resource-description>
              <resource-description>
                   <res-ref-name>jdbc/demoXAPool</res-ref-name>
                   <jndi-name>examples-dataSource-demoXAPool</jndi-
name>
              </resource-description>
              <resource-env-description>
                   <res-env-ref-name>jms/Topic</res-env-ref-name>
                   <jndi-name>SunMicrosystems.Topics.STCTopic3</jndi-
name>
              </resource-env-description>
         </reference-descriptor>
```

```
        <jndi-name>SunMicrosystems.STCXAPublisherSLSessionBean</
jndi-name>
     ...

</weblogic-ejb-jar>
```

The value for <ejb-name> must match that defined in **ejb-jar.xml**.

The <pool> tag defines the maximum number of Session Beans in the free pool and the initial pool size by using the <max-beans-in-free-pool> and <initial-beans-in-free-pool> tags respectively. The value for the jndi-name tag for the resource name **jms/XATopicConnectionFactory** is:

SunMicrosystems.TopicConnectionFactories.XATopicConnectionFactory

The value for the jndi-name tag for the jms/Topic entry is:

SunMicrosystems.Topics.STCTopic3

These values define the resource reference name to JNDI name mappings. The Topic and **XATopicConnectionFactory** must already be created and registered with JNDI by the startup class. (See **"Sun Microsystems WebLogic Startup Class" on page 43** for details.) The container locates these JNDI objects in its own JNDI as specified by the <initial-context-factory> and <provider-url>. Notice that the actual JNDI name for the jdbc/demoXAPool resource is **examples-dataSource-demoXAPool**. This is the JNDI name of the datasource XA pool that is already created and configured for the examples WebLogic Server when WebLogic is installed.

Verifying XA At Work

XA works transparently when the EJBs are running. To observe XA working, look at the Sun Microsystems JMS server log. When XA works, the user sees the XA APIs being called. To see the XA APIs being logged, write the trace messages to a file.

The JMS server log should appear something like this :

```
17:49:53.299 JMS  I 2676 (Session.cpp:716): XA prepare for Session sessionid=63737404, transaction
txnid=63737405
17:49:53.299 JMS  I 2676 (SessionManager.cpp:694): XAPrepare() :
xid:48801:0005fa80c71858e3d95b:636f6d2e7365656265796f6e642e6a6d732e636c69656e742e53544358415265736
f75726365
…
17:49:53.460 JMS  I 2676 (Session.cpp:775): Session::XACommit() session sessionid=63737404,
transaction txnid=63737438
17:49:53.460 JMS  I 2676 (SessionManager.cpp:710): XACommit() :
xid:48801:0005fa80c71858e3d95b:636f6d2e7365656265796f6e642e6a6d732e636c69656e742e53544358415265736
f75726365
```

In addition, WebLogic JTA and JMS XA tracing can be turned on by doing the following:

For **WebLogic 7.0** and **8.1**, modify startExamplesServer.cmd at:

<BEA-HOME>\user_projects\<domain name> to set the JTA / JMS debug flag as follows:

```
JAVA_VM=-Dweblogic.Debug=weblogic.JTAXA -Dweblogic.Debug.DebugJMSXA=true
```

or

```
JAVA_OPTIONS=-Dweblogic.Debug=weblogic.JTAXA -Dweblogic.Debug.DebugJMSXA=true
```

Once these properties are added, restart the server. JTA and JMS XA tracing is written to the server log which is typically located in a subdirectory with the same name as the

server, under the current domain in use. For example, given a server named “serv” the location would be:

```
BEA\WebLogic7\user_projects\mydomain\serv\serv.log
```

```
####<Apr 4, 20xx 5:49:52 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <5:fa80c71858e3d95b:
XA.start(rm=com.SunMicrosystems.jms.client.STCXAResource,
xar=com.SunMicrosystems.jms.client.STCXAResource@82e1a, flags=TMNOFLAGS)>
####<Apr 4, 20xx 5:49:52 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <> <>
<000000> <ResourceDescriptor[com.SunMicrosystems.jms.client.STCXAResource]: startResourceUse,
Number of active requests:1, last alive time:0 ms ago.>
####<Apr 4, 20xx 5:49:52 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <5:fa80c71858e3d95b: XA.start DONE
(rm=com.SunMicrosystems.jms.client.STCXAResource,
xar=com.SunMicrosystems.jms.client.STCXAResource@82e1a>
####<Apr 4, 20xx 5:49:52 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <ResourceDescriptor[com.SunMicrosystems.jms.client.STCXAResource]:
endResourceUse, Number of active requests:0>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <ResourceDescriptor[demoXAPool]: getOrCreate gets rd: name =
demoXAPool
xar = demoXAPool
registered = true
enlistStatically = false
healthy = true
lastAliveTimeMillis = -1
numActiveRequests = 0
scUrls = examplesServer+10.1.50.134:7003+examples+
>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <5:fa80c71858e3d95b: XA.start(rm=demoXAPool, xar=demoXAPool,
flags=TMNOFLAGS)>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <> <>
<000000> <ResourceDescriptor[demoXAPool]: startResourceUse, Number of active requests:1, last alive
time:0 ms ago.>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <5:fa80c71858e3d95b: XA.start DONE (rm=demoXAPool, xar=demoXAPool>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <ResourceDescriptor[demoXAPool]: endResourceUse, Number of active
requests:0>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <> <>
<000000> <5:fa80c71858e3d95b: XA.end(rm=com.SunMicrosystems.jms.client.STCXAResource,
xar=com.SunMicrosystems.jms.client.STCXAResource@82e1a, flags=TMSUCCESS)>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <> <>
<000000> <ResourceDescriptor[com.SunMicrosystems.jms.client.STCXAResource]: startResourceUse,
Number of active requests:1, last alive time:0 ms ago.>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <5:fa80c71858e3d95b: XA.end DONE
(rm=com.SunMicrosystems.jms.client.STCXAResource,
xar=com.SunMicrosystems.jms.client.STCXAResource@82e1a>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <ResourceDescriptor[com.SunMicrosystems.jms.client.STCXAResource]:
endResourceUse, Number of active requests:0>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <> <>
<000000> <5:fa80c71858e3d95b: XA.end(rm=demoXAPool, xar=demoXAPool, flags=TMSUCCESS)>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <> <>
<000000> <ResourceDescriptor[demoXAPool]: startResourceUse, Number of active requests:1, last alive
time:0 ms ago.>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <5:fa80c71858e3d95b: XA.end DONE (rm=demoXAPool, xar=demoXAPool>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <ResourceDescriptor[demoXAPool]: endResourceUse, Number of active
requests:0>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <> <>
<000000> <5:fa80c71858e3d95b: XA.prepare(rm=com.SunMicrosystems.jms.client.STCXAResource,
xar=com.SunMicrosystems.jms.client.STCXAResource@82e1a>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <> <>
<000000> <ResourceDescriptor[com.SunMicrosystems.jms.client.STCXAResource]: startResourceUse,
Number of active requests:1, last alive time:0 ms ago.>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <5:fa80c71858e3d95b: XA.prepare DONE:ok>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <ResourceDescriptor[com.SunMicrosystems.jms.client.STCXAResource]:
endResourceUse, Number of active requests:0>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <> <>
<000000> <5:fa80c71858e3d95b: XA.prepare(rm=demoXAPool, xar=demoXAPool>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <> <>
<000000> <ResourceDescriptor[demoXAPool]: startResourceUse, Number of active requests:1, last alive
time:0 ms ago.>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <5:fa80c71858e3d95b: XA.prepare DONE:ok>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <ResourceDescriptor[demoXAPool]: endResourceUse, Number of active
requests:0>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <> <>
<000000>
<XAResource[com.SunMicrosystems.jms.client.STCXAResource].commit(xid=5:fa80c71858e3d95b,onePhase=f
alse)>
```

```
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <> <>
<000000> <ResourceDescriptor[com.SunMicrosystems.jms.client.STCXAResource]: startResourceUse,
Number of active requests:1, last alive time:0 ms ago.>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <5:fa80c71858e3d95b: XA.commit DONE
(rm=com.SunMicrosystems.jms.client.STCXAResource,
xar=com.SunMicrosystems.jms.client.STCXAResource@82e1a>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <ResourceDescriptor[com.SunMicrosystems.jms.client.STCXAResource]:
endResourceUse, Number of active requests:0>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <> <>
<000000> <XAResource[demoXAPool].commit(xid=5:fa80c71858e3d95b,onePhase=false)>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <> <>
<000000> <ResourceDescriptor[demoXAPool]: startResourceUse, Number of active requests:1, last alive
time:0 ms ago.>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <5:fa80c71858e3d95b: XA.commit DONE (rm=demoXAPool, xar=demoXAPool>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <ResourceDescriptor[demoXAPool]: endResourceUse, Number of active
requests:0>
```

Additional Logging and Monitoring of JTA and JMS XA

Additional logging and monitoring of JTA and JMS XA can be configured for WebLogic Server 7.0 and 8.1 through the Administrator Console. From the navigation pane on the left, expand the Servers node and select the appropriate server. Configure monitoring and logging in the following locations:

- Select the Monitoring tab and click on the JMS and JTA subtabs.
- Select the Logging tab and click on the JTA and Debugging subtabs.

examples-dataSource-demoXAPool

As part of its examples server, WebLogic pre-installs a pre-configured datasource named examples-dataSource-demoXAPool (see Figure 58) and associates it with the pre-installed connection pool named demoXAPool (see Figure 59). This datasource is intended for use with the sample WebLogic EJBs that are deployed with the examples server, but it is also used by the EJBs supplied with the WebLogic eWay. Use the figures below to verify that the WebLogic examples server is properly set up to work with the sample eGate projects/EJBs discussed in this document.

Figure 58 WebLogic (8.1) Administrative Console - demoXAPool

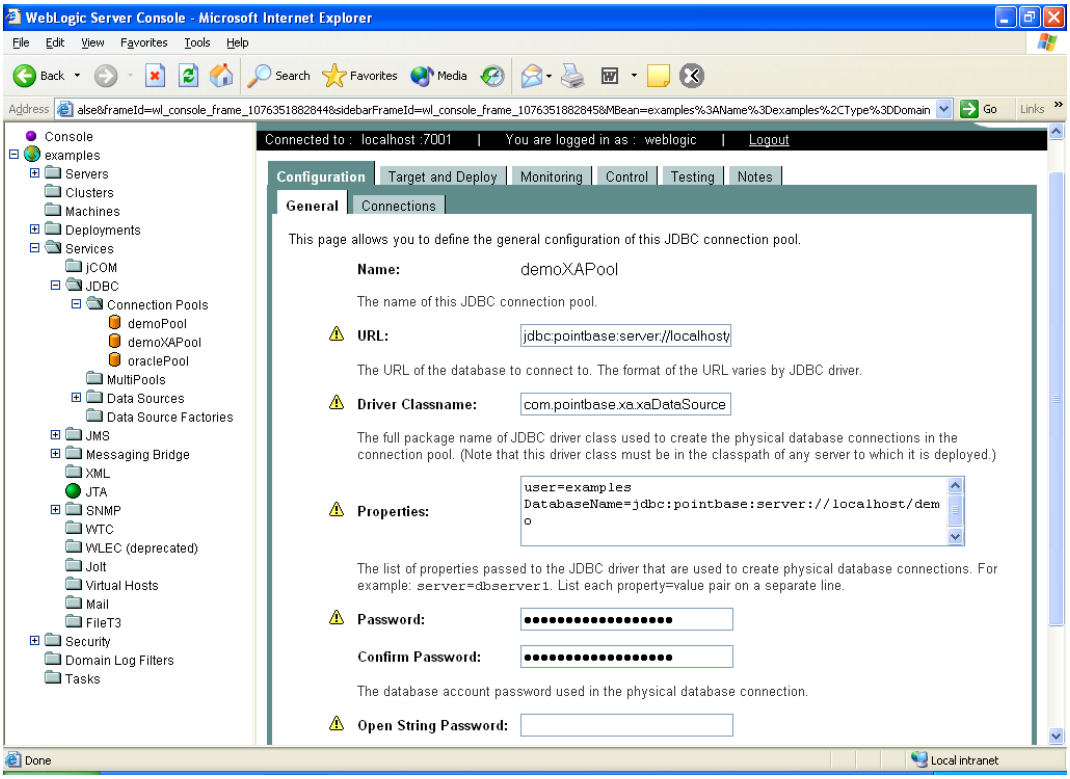
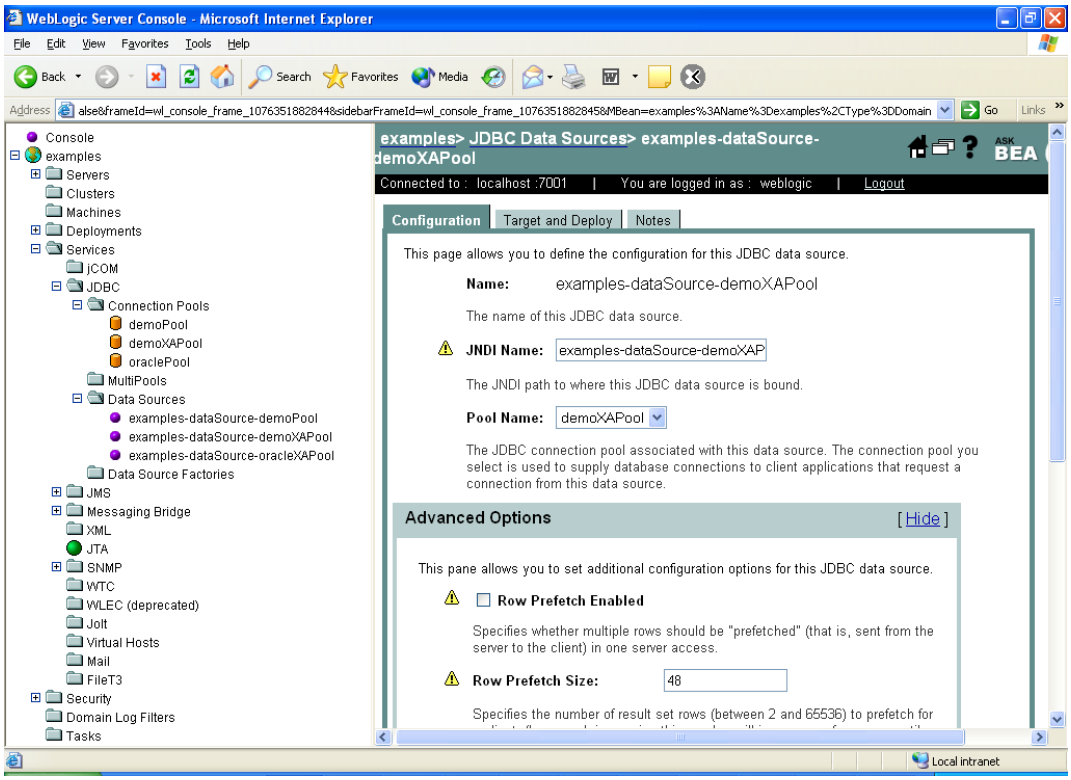


Figure 59 WebLogic (8.1) Administrative Console - demoXAPool



Index

L

M

N

O

P

Q

R

S

T

U

W

X