SeeBeyond ICAN Suite

WebLogic eWay Intelligent Adapter User's Guide

Release 5.0

SEEBEYOND®

Contents

Chapter 1

Introducing the WebLogic eWay

This document describes the integration between BEA WebLogic™ application Server and SeeBeyond eGate using the WebLogic eWay Intelligent 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.

1.3 About This Document

This guide explains how to install, configure, and operate the SeeBeyond® Integrated Composite Application Network Suite™ (ICAN) WebLogic eWay Intelligent Adapter, referred to as the WebLogic eWay throughout this guide.

1.3.1. What's in 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 eWay":** Describes the system requirements and provides instructions for installing the WebLogic eWay.
- **Chapter 3 "Setting Properties of 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 eWay Intelligent Adapter for WebLogic communicate with the WebLogic Application Server.
- **Chapter 6 "Configuring WebLogic Server":** Provides directions for configuring WebLogic Server for asynchronous interaction with eGate.
- **Chapter 7 "Using the WebLogic OTD Wizard":** Describes how to build and use Object Type Definitions (OTDs) using the WebLogic OTD Wizard.
- **Chapter 8 "Implementing the WebLogic eWay":** Describes how to use the sample projects included in the installation CD-ROM package
- **Chapter 9 "SeeBeyond Sample Message Driven Beans"** Provides further information on the messaging objects designed to route messages from clients to other Enterprise Java Beans.

1.3.2. Scope

This document describes the process of installing, configuring, and running the WebLogic eWay.

1.3.3. Intended Audience

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

1.3.4. Document Conventions

The following writing conventions are observed throughout this document.

Table 1 Writing Conventions

Text	Convention	Example
Button, file, icon, parameter, variable, method, menu, and object names.	**Bold** text	▪ Click **OK** to save and close. ▪ From the **File** menu, select **Exit**. ▪ Select the **logicalhost.exe** file. ▪ Enter the **timeout** value. ▪ Use the **getClassName()** method. ▪ Configure the **Inbound** File eWay.
Command line arguments and code samples	`Fixed` font. Variables are shown in ***bold italic***.	`bootstrap -p` ***password***
Hypertext links	**Blue** text	**http://www.seebeyond.com**

1.3.5. Screenshots

Depending on what products you have installed, and how they are configured, the screenshots in this document may differ from what you see on your system.

1.4 Related Documents

The following SeeBeyond documents provide additional information about the ICAN product suite:

- *eGate Integrator User's Guide*
- *SeeBeyond ICAN Suite Installation Guide*

1.5 SeeBeyond Web Site

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

http://www.seebeyond.com

1.6 Feedback

If you have any feedback on any SeeBeyond documentation, please send an E-mail to:

docfeedback@seebeyond.com

Chapter 2

Installing the eWay

This chapter explains how to install the WebLogic eWay.

What's in This Chapter

2.1 Supported Operating Systems

The WebLogic eWay is available on the following operating systems:

- Windows XP, Windows 2000, and Windows Server 2003
- HP-UX 11.0 and HP-UX 11i (PA-RISC)
- IBM AIX 5.1L and 5.2
- Red Hat Linux 8 (Intel Version)
- Red Hat Enterprise Linux AS 2.1
- Sun Solaris 8 and 9
- Korean Windows XP, Windows 2000, and Windows Server 2003
- Korean HP-UX 11.0 and HP-UX 11i (PA-RISC)
- Korean IBM AIX 5.1L and 5.2
- Korean Sun Solaris 8 and 9

2.2 System Requirements

The system requirements for the WebLogic eWay are the same as for eGate Integrator. For information, refer to the *SeeBeyond ICAN Suite Installation Guide*. It is also helpful to review the **Readme.txt** for any additional requirements prior to installation. The **Readme.txt** is located on the installation CD-ROM.

2.3 Supported External Applications

The WebLogic eWay supports the following software for external systems:

- BEA WebLogic Server 6.1, 7.0, or 8.1
- WebLogic Server 6.1 SP 7 for Korean platforms

2.4 Installing the eWay Product Files

During the ICAN Suite installation process, the Enterprise Manager—a web-based application running on the Windows operating system—is used to select and upload eWay and Add-on files (.sar files) from the ICAN installation CD-ROM to the Repository.

Note: *Refer to the ICAN Installation Guide for additional installation instructions.*

2.4.1. Required Files

During the procedures for uploading files to the eGate Repository using the Enterprise Manager, select and upload the following files:

- **WebLogiceWay.sar** (to install the WebLogic eWay)
- **WebLogiceWayDocs.sar** (to install the eWay user guide, Javadoc, and Samples)
- **FileeWay.sar** (to install the File eWay, used in the sample Project)

2.4.2. Sample Projects

Sample projects are included to help demonstrate key features and technologies used in the WebLogic eWay.

To Download Sample Projects

1. In the Enterprise Manger, click the **DOCUMENTATION** tab.
2. Click **WebLogic eWay**.
3. In the right pane, click **Download Sample**, and select a location to save the .zip file.

Additional information on importing and using sample projects can be found in **Implementing the WebLogic eWay** on page 75.

2.5 After You Install

Once the eWay is installed and configured it must then be incorporated into a project before it performs its intended functions. See the *eGate Integrator User's Guide* for more information on incorporating the eWay into an eGate project.

Chapter 3

Setting Properties of WebLogic eWay

This chapter describes how to create and configure a WebLogic eWay.

What's in This Chapter:

3.1 Configuring the WebLogic eWay Properties

The WebLogic eWay includes a unique set of configuration parameters. After establishing an eWay and creating a WebLogic External System in the project's Environment, the property parameters are modified for your specific system.

WebLogic eWay properties are modified from two locations:

- From the **Connectivity Map**—which contains parameters specific to the WebLogic eWay.
- From the **Environment Explorer tree**—which contains parameters common to all eWays (of the same type) in the project.

Note: *You must set configuration parameters for WebLogic eWay in both locations.*

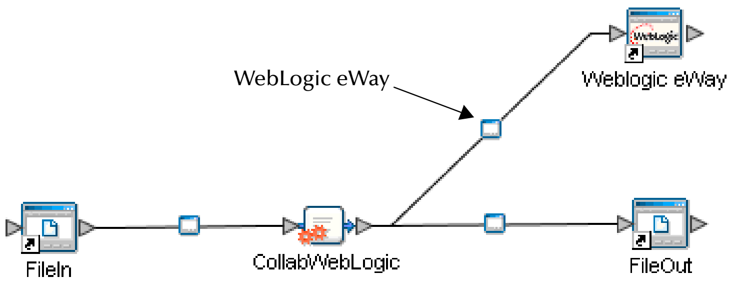
3.2 Configuring the eWay Connectivity Map Properties

When you connect an External Application to a Collaboration, Enterprise Designer automatically assigns the appropriate eWay to the link. Each eWay is supplied with a template containing default configuration properties that are accessible on the Connectivity Map.

To configure the eWay properties:

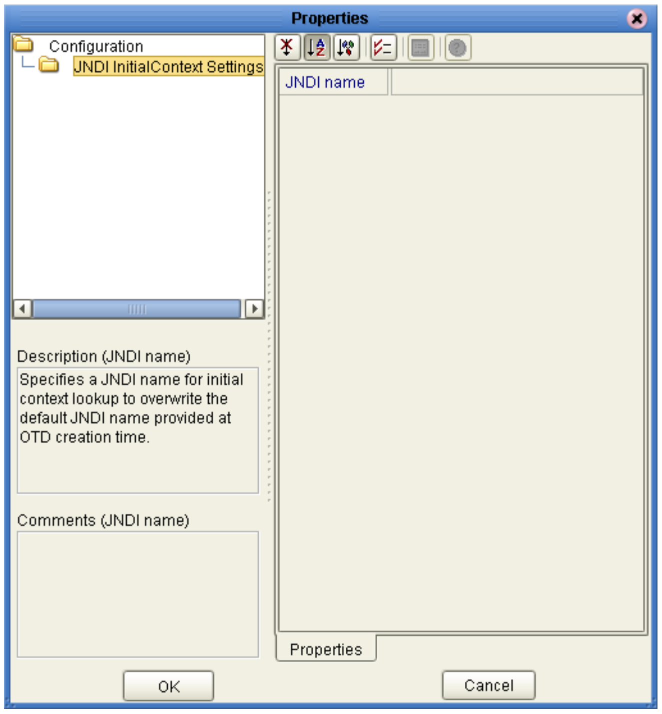
1 On the Enterprise Designer’s Connectivity Map (see Figure 1), double-click the inbound WebLogic eWay icon.

Figure 1 Connectivity Map with Components



2 The **Configuration** properties window opens, displaying the default properties for the eWay.

Figure 2 Configuration Editor: WebLogic eWay



Description (JNDI name)
Specifies a JNDI name for initial context lookup to overwrite the default JNDI name provided at OTD creation time.

Comments (JNDI name)

3. Click on the ellipsis (...) in the properties field (displayed during modification of the value) to open a separate configuration dialog box. 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 is now displayed in the parameter's property field.
4. 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.
5. The **Comments** pane provides an area for recording notes and information regarding the currently selected parameter. This is saved for future referral.
6. After modifying the configuration properties, click **OK** to close the Properties Sheet and save the changes.

3.2.1. eWay Property Settings

The WebLogic eWay Properties window contains one eWay specific property called JNDI name.

JNDI name

Description

Specifies a Java Naming and Directory Interface (JNDI) name for initial context lookup. This JNDI name overwrites the default JNDI name provided when the OTD was created.

Required Value

A JNDI name (String).

3.3 Configuring the Environment Properties

The eWay configuration properties contain parameters that define how the eWay connects to, and interacts with, other eGate components within the Environment.

3.3.1. WebLogic External System Properties

The WebLogic Environment contains outbound properties that are accessible via the Environment Explorer. 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. The **Comments** pane provides an area for recording notes and information regarding the currently selected parameter. This is saved for future referral.

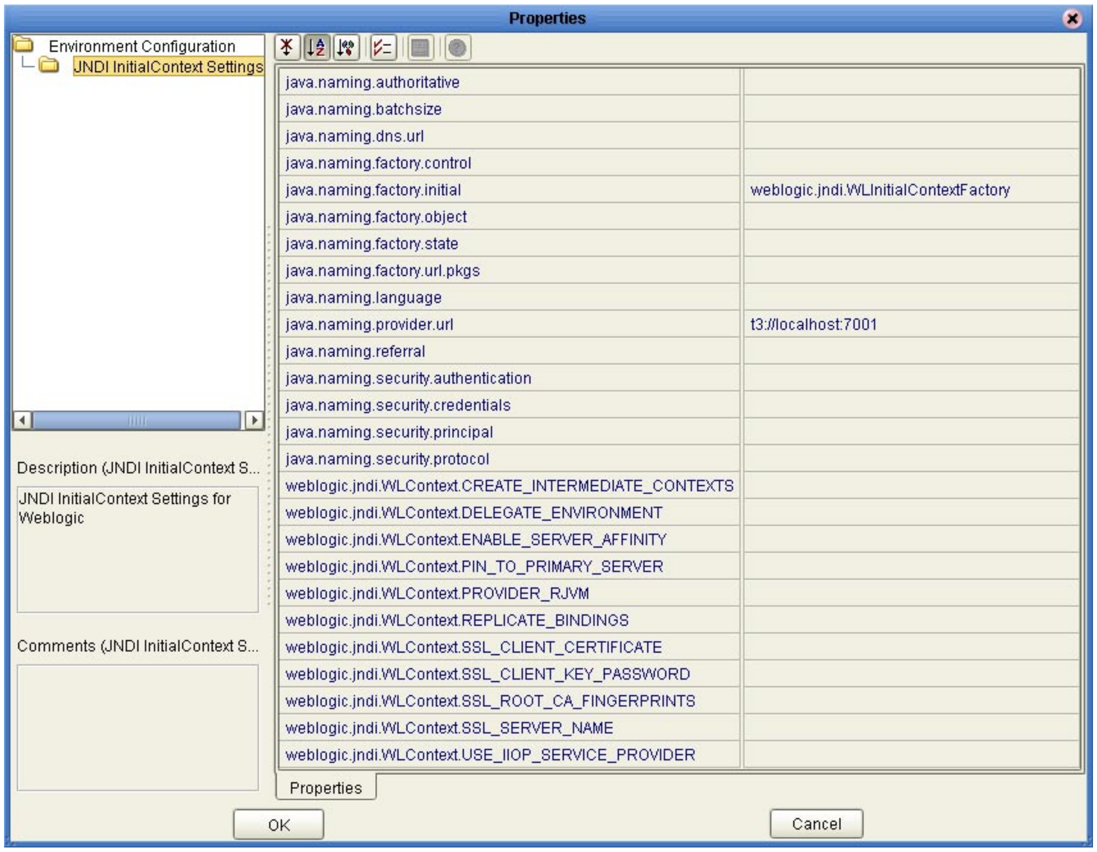
To Configure the Environment Properties:

1. In Enterprise Explorer, click the Environment Explorer tab.
2. Expand the Environment created for the WebLogic project and locate the WebLogic External System.

Note: *For more information on creating an Environment, see the eGate Integrator Tutorial.*

3 Right-click the WebLogic External System and select Properties from the list box. The Environment Configuration Properties window appears.

Figure 3 Environment Configuration outbound properties


Properties
Environment Configuration
JNDI InitialContext Settings
Description (JNDI InitialContext S...
Comments (JNDI InitialContext S...
Properties
OK
Cancel

JNDI InitialContext Settings for Weblogic

java.naming.authoritative	
java.naming.batchsize	
java.naming.dns.url	
java.naming.factory.control	
java.naming.factory.initial	weblogic.jndi.WLInitialContextFactory
java.naming.factory.object	
java.naming.factory.state	
java.naming.factory.url.pkgs	
java.naming.language	
java.naming.provider.url	t3://localhost:7001
java.naming.referral	
java.naming.security.authentication	
java.naming.security.credentials	
java.naming.security.principal	
java.naming.security.protocol	
weblogic.jndi.WLContext.CREATE_INTERMEDIATE_CONTEXTS	
weblogic.jndi.WLContext.DELEGATE_ENVIRONMENT	
weblogic.jndi.WLContext.ENABLE_SERVER_AFFINITY	
weblogic.jndi.WLContext.PIN_TO_PRIMARY_SERVER	
weblogic.jndi.WLContext.PROVIDER_RJVM	
weblogic.jndi.WLContext.REPLICATE_BINDINGS	
weblogic.jndi.WLContext.SSL_CLIENT_CERTIFICATE	
weblogic.jndi.WLContext.SSL_CLIENT_KEY_PASSWORD	
weblogic.jndi.WLContext.SSL_ROOT_CA_FINGERPRINTS	
weblogic.jndi.WLContext.SSL_SERVER_NAME	
weblogic.jndi.WLContext.USE_IIOP_SERVICE_PROVIDER	

4 Click on the ellipsis (**...**) in the properties field (displayed during modification of the value) to open a separate configuration dialog box. 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 is now displayed in the parameter's property field.

5 After modifying the configuration properties, click **OK** to close the Properties Sheet and save the changes.

3.3.2. Environment Property Settings

Configure your Environment properties to match the properties listed below.

Settings Include:

- **java.naming.authoritative** on page 19
- **java.naming.batchsize** on page 20

- **java.naming.dns.url** on page 20
- **java.naming.factory.control** on page 20
- **java.naming.factory.initial** on page 20
- **java.naming.factory.object** on page 20
- **java.naming.factory.state** on page 21
- **java.naming.factory.url.pkgs** on page 21
- **java.naming.language** on page 21
- **java.naming.provider.url** on page 21
- **java.naming.referral** on page 22
- **java.naming.security.authentication** on page 22
- **java.naming.security.credentials** on page 22
- **java.naming.security.principal** on page 23
- **java.naming.security.protocol** on page 23
- **weblogic.jndi.WLContext.CREATE_INTERMEDIATE_CONTEXTS** on page 23
- **weblogic.jndi.WLContext.DELEGATE_ENVIRONMENT** on page 23
- **weblogic.jndi.WLContext.PIN_TO_PRIMARY_SERVER** on page 24
- **weblogic.jndi.WLContext.PROVIDER_RJVM** on page 24
- **weblogic.jndi.WLContext.REPLICATE_BINDINGS** on page 24
- **weblogic.jndi.WLContext.SSL_CLIENT_CERTIFICATE** on page 24
- **weblogic.jndi.WLContext.SSL_CLIENT_KEY_PASSWORD** on page 25
- **weblogic.jndi.WLContext.SSL_ROOT_CA_FINGERPRINTS** on page 25
- **weblogic.jndi.WLContext.SSL_SERVER_NAME** on page 25
- **weblogic.jndi.WLContext.USE_IIOP_SERVICE_PROVIDER** on page 25

java.naming.authoritative

Description

Specifies the authoritativeness of the service requested. If **true** is specified, the most authoritative source is used (for example, bypass any caches, or bypass replicas in some systems). Otherwise, the source need not be (but can be) authoritative.

Required Value

Either the value **true** or **false**. **False** is the configured default.

java.naming.batchsize

Description

Specifies the preferred batch size to use when returning data using the WebLogic Server protocol. This suggestion, for the user to return the results of operations in batches of a specified size, optimizes its performance and resources. It does not affect number or size of the data returned.

Required Value

A preferred batch size. If not specified, it defaults to the service provider default.

java.naming.dns.url

Description

Specifies the DNS host and domain names (Context.DNS_URL).

Required Value

A valid DNS host. If not specified, it defaults to the service provider default.

java.naming.factory.control

Description

Specifies a colon-separated list of the class names for the response control factory classes to be used (LdapContext.CONTROL_FACTORIES). See ControlFactory.getControlInstance().

Required Value

Class names of the response control factory classes, separated by a colon.

java.naming.factory.initial

Description

Specifies the class name of initial context factory. Defines the implementation of JNDI to be used by the client (Context.INITIAL_CONTEXT_FACTORY). For most cases use the configured default.

Required Value

The class name of the initial context factory to be used. weblogic.jndi.WLInitialContextFactory is the configured default.

java.naming.factory.object

Description

Specifies a colon-separated list of the class names for the object factory classes to be used (Context.OBJECT_FACTORIES). See NamingManager.getObjectInstance() and DirectoryManager.getObjectInstance().

Required Value

Class names of object factory classes, separated by a colon.

java.naming.factory.state

Description

Specifies a colon-separated list of the class names for the state factory classes to be used (Context.STATE_FACTORIES). See NamingManager.getStateToBind() and DirectoryManager.getStateToBind().

Required Value

Class names of state factory classes, separated by a colon.

java.naming.factory.url.pkgs

Description

Specifies a colon-separated list of the package prefixes to use when loading in URL context factories (Context.URL_PKG_PREFIXES). See NamingManager.getURLContext().

Required Value

Valid package prefixes used for loading URL context factories, separated by a colon. com.sun.jndi.url is always added to end of list.

java.naming.language

Description

Specifies a colon-separated list of the preferred languages to use with this service. Languages are specified using tags defined in RFC 1766. (Context.LANGUAGE)

Required Value

Valid language tags as specified by RFC1776 protocol, separated by a colon (for example, en-US:fr:fr-CH:ja-JP-kanji). If not specified it defaults to the service provider default.

java.naming.provider.url

Description

Specifies the PROVIDER_URL (Context.PROVIDER_URL).

Required Value

The URL of the participating host (for example, t3://localhost:7001 or http:localhost:7003). If not specified, it defaults to the service provider default.

java.naming.referral

Description

Specifies whether referrals encountered by the service provider are to be followed automatically. (Context.REFERRAL) The value of the property is one of the following:

- **follow:** follow referrals automatically.
- **ignore:** ignore any encountered referrals.
- **throw:** throw a ReferralException when a referral is encountered.

Required Value

A naming referral property. Values are **follow**, **ignore**, or **throw**. If not specified, it defaults to the service provider default.

java.naming.security.authentication

Description

Specifies the security authentication scheme to use. (Context.SECURITY_AUTHENTICATION) The values are as follows:

- **simple**: provides user password authentication. Values must also be provided for java.naming.security.principal and java.naming.security.credentials parameters.
- **strong**: provides certificate authentication (a file name). May require the use of X.509 certificates for the java.naming.security.credentials property. Values must also be provided for java.naming.security.principal and java.naming.security.credentials parameters.
- **none**: no required authentication.
- **user-defined**: a user-defined key for authentication. Values must also be provided for java.naming.security.principal and java.naming.security.credentials parameters.

Required Value

A security authentication property. Values are **simple**, **strong**, **none**, or a **user-defined** key. If not specified it defaults to the service provider default.

java.naming.security.credentials

Description

Specifies the principal's (user's) credentials for the authentication scheme determined by the authentication value specified for java.naming.security.authentication. If the value is set as **simple** this would be a password. If the value is **strong** this would be certificate (a file). If the value is **user-defined** then it would be the user-specified key. If the authentication value is **none** no value is set for credentials.

Required Value

Either a password, a certificate (file), or a user-defined key depending on the value set for java.naming.security.authentication. If not specified, it defaults to "**guest**", the service provider default.

java.naming.security.principal

Description

Specifies the identity of the principal (user) for the authentication scheme when the java.naming.security.authentication value is set as **simple** or **strong**.

Required Value

Either a user **name** or a **certificate** depending on the value entered for java.naming.security.authentication. If not specified, it defaults to “**guest**”, the service provider default.

java.naming.security.protocol

Description

Specifies the security protocol to use (for example, “ssl”).

Required Value

A security protocol. If not specified, it defaults to the service provider default.

weblogic.jndi.WLContext.CREATE_INTERMEDIATE_CONTEXTS

Description

Specifies the way to handle non-existent intermediate contexts. If **true** then performing a bind, rebind, or createSubcontext with a name that specifies non- existent intermediate contexts creates those contexts.

Required Value

Either the value **true** or **false**. If not specified, it defaults to the service provider default.

weblogic.jndi.WLContext.DELEGATE_ENVIRONMENT

Description

Specifies the JNDI environment to use for connecting to a third-party naming service through the WebLogic Server. When specified, the WebLogic Server creates a three-tier connection to a third-party naming service. Properties contained in the Hashtable specified by this parameter are used to create an initial context for the third-party naming service. The original initial context then delegates its work to the third-party's initial context.

Required Value

A specified JNDI environment.

weblogic.jndi.WLContext.ENABLE_SERVER_AFFINITY

Description

This parameter applies to **WebLogic 8.1 only**. Specifies whether multiple context creations from the same VM will reuse existing connections or round-robin across all servers in the cluster (cluster-specific). By default, context creations will round-robin across all servers in a cluster. This property takes affect only when a cluster url is specified. If a specific server url is used, context creation would always connect to that server.

Required Value

Either the value **true** or **false**. **True** indicates that the VM will reuse existing connections. The configured default is false.

weblogic.jndi.WLContext.PIN_TO_PRIMARY_SERVER

Description

Specifies whether the context stub only connects to the primary naming server. Cluster-specific: If set as **true**, this parameter forces the context stub to connect to only the server currently running at the host specified by Context.PROVIDER_URL.

Required Value

Either the value **true** or **false**. The configured default is **false**.

weblogic.jndi.WLContext.PROVIDER_RJVM

Description

Specifies the RJVM to use as the naming server. This may be used as an alternative to Context.PROVIDER_URL. It specifies an RJVM representing the desired server rather than a URL.

Required Value

A specified RJVM.

weblogic.jndi.WLContext.REPLICATE_BINDINGS

Description

Cluster-specific: Specifies whether tree modifications are replicated. This only applies when connecting to WebLogic Servers that are running in a cluster. If set to **false**, modifications to the tree caused by bind, unbind, createSubcontext, and destroySubcontext, are not replicated. A **false** value should only be used with extreme caution. The default setting for the parameter is **true**, which allows any modification to the naming tree to be replicated across the cluster. This ensures that any server can act as a naming server for the entire cluster.

Required Value

Either the value **true** or **false**. The default value is **true**.

weblogic.jndi.WLContext.SSL_CLIENT_CERTIFICATE

Description

Specifies an RSA private key and a chain of certificates for client authentication. This can be set to **SERVER**, a special string that refers to the server’s private key and certificate chain. Generally, it is set to an array of InputStreams, the first element being a DER-encoded RSA private key, followed by DER_encoded X.509 certificates. Other than the first, all certificates must be an issuer certificate of the preceding certificate.

Required Value

An RSA private key and a chain of certificates.

weblogic.jndi.WLContext.SSL_CLIENT_KEY_PASSWORD

Description

Specifies the password for an encrypted PKCS5/PKCS8 RSA private key.

Required Value

A valid password.

weblogic.jndi.WLContext.SSL_ROOT_CA_FINGERPRINTS

Description

Specifies valid certificate authorities using a set of fingerprints (MD5) of the authorities' certificates encoded either as an array of byte arrays, or a comma-separated string of hex values. When specified, the SSL connection can only be established to a server that presents a certificate chain in which the fingerprint of the root matches one of the fingerprints specified by the parameter value.

Required Value

A set of fingerprints (MD5) of the authorities' certificates encoded either as an array of byte arrays, or a comma-separated string of hex values.

weblogic.jndi.WLContext.SSL_SERVER_NAME

Description

Specifies an expected name of an SSL server as a String. The value must match the common name field in the certificate provided by the server (typically the WebLogic Server’s DNS name).

Required Value

A specific SSL server name.

weblogic.jndi.WLContext.USE_IIOP_SERVICE_PROVIDER

Description

Applies to **WebLogic 6.1 and 7.0 only**. Specified when the caller intends to use the WebLogic IIOP service provider to establish an IIOP connection to the naming server.

Required Value

USE_IIOP_SERVICE_PROVIDER to specify use.

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:7003/
```

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:7003/");
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.inTopicToSeeBeyondTopic");
...
```

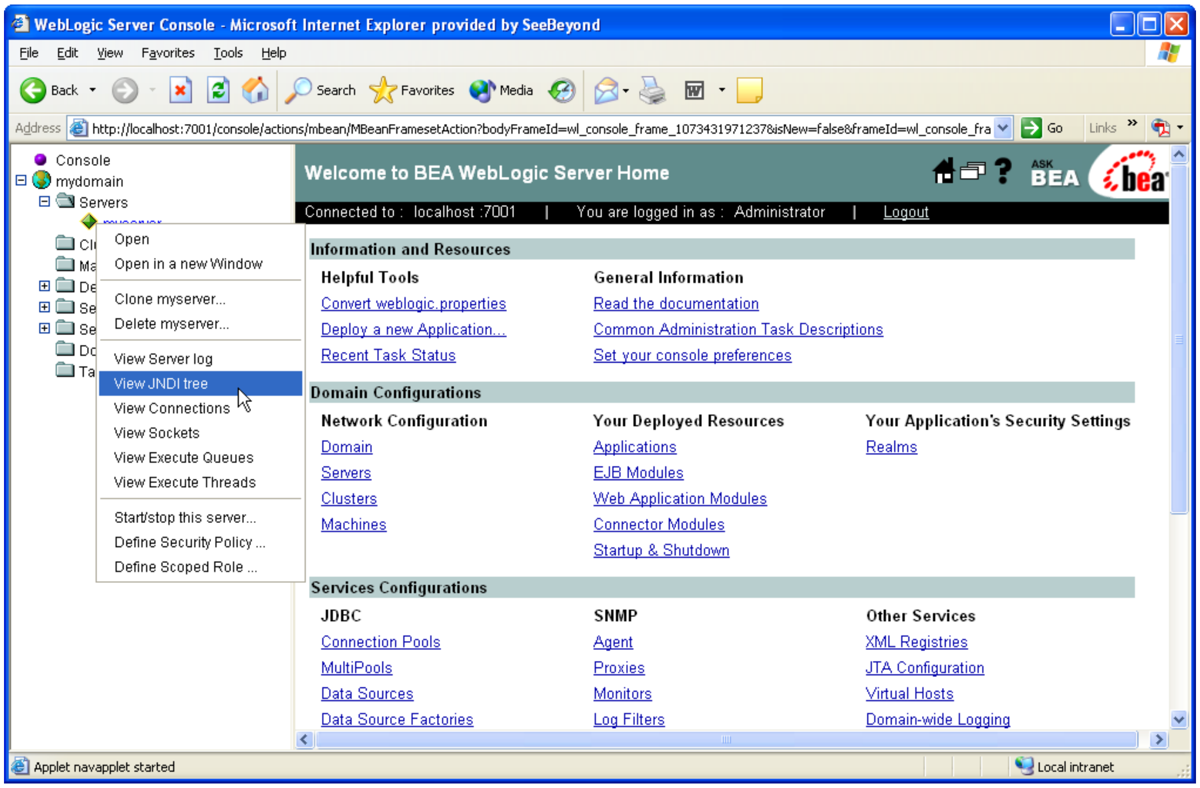
Here's an example of how to bind a SeeBeyond Queue object:

```
Queue queue = null;
try {
   queue = new STCQueue("inQueueToSeeBeyondQueue");
   initContext.bind ("sbyn.ToSeeBeyondQueue", queue);
}
catch (NameAlreadyBoundException ex)
{
   try
   {
    if (queue != null)
     initContext.rebind ("sbyn.ToSeeBeyondQueue", 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 4), 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.

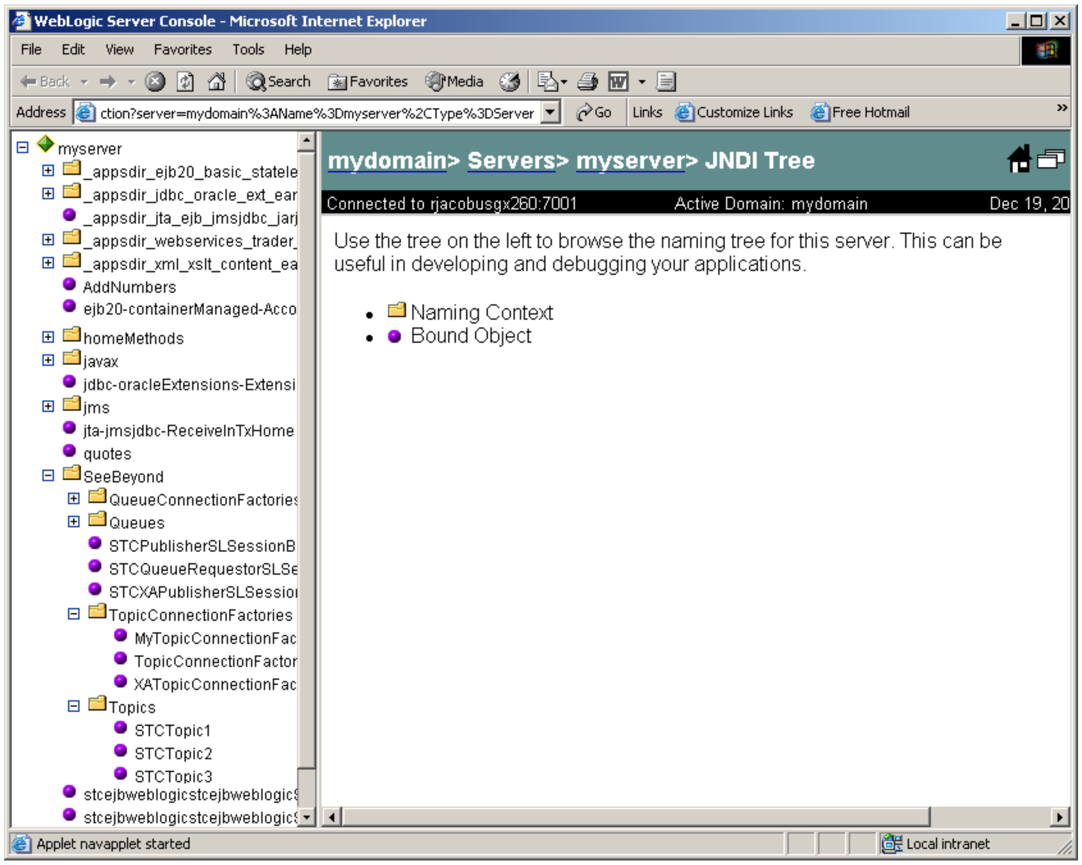
Figure 4 Administrative Console - View JNDI Tree



In the following example, (see **Figure 5 on page 29**) the JNDI tree Web page shows that the **SeeBeyond** subcontext was expanded in order to view the SeeBeyond JMS objects that were bound to the WebLogic JNDI. These objects are bound when the **STCWLStartup** class is loaded and run by the WebLogic Server. (See **SeeBeyond WebLogic Startup Class** on page 47 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 5 Administrative Console - The JNDI Tree Web Page


WebLogic Server Console - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address ction?server=mydomain%3AName%3Dmyserver%2CType%3DServer
Links Customize Links Free Hotmail
myserver
_appsdir_ejb20_basic_statele
_appsdir_jdbc_oracle_ext_ear
_appsdir_jta_ejb_jmsjdbc_jarj
_appsdir_webservices_trader_
_appsdir_xml_xslt_content_ea
AddNumbers
ejb20-containerManaged-Acco
homeMethods
javax
jdbc-oracleExtensions-Extensi
jms
jta-jmsjdbc-ReceiveInTxHome
quotes
SeeBeyond
QueueConnectionFactories
Queues
STCPublisherSLSessionB
STCQueueRequestorSLSe
STCXAPublisherSLSessio
TopicConnectionFactories
MyTopicConnectionFac
TopicConnectionFactor
XATopicConnectionFac
Topics
STCTopic1
STCTopic2
STCTopic3
stcejbweblogicstcejbweblogic
stcejbweblogicstcejbweblogic
mydomain> Servers> myserver> JNDI Tree
Connected to rjacobusgx260:7001 Active Domain: mydomain Dec 19, 20
Naming Context
Bound Object
Applet navapplet started
Local intranet

Use the tree on the left to browse the naming tree for this server. This can be useful in developing and debugging your applications.

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 6 and Figure 7). 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 **SeeBeyond JMS** on page 38).

Figure 6 Topic - The Publish-Subscribe Model.

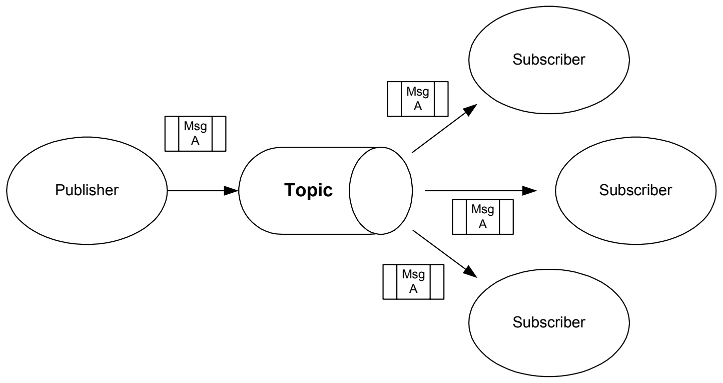
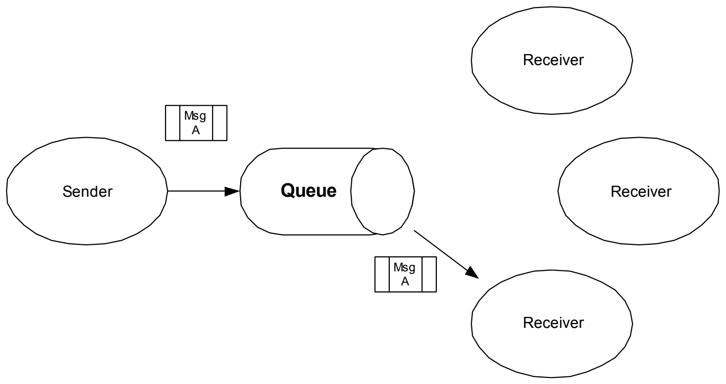


Figure 6 shows multiple subscribers receiving the same messages when the publisher publishes the message to a Topic. This is the pubsub (publish-subscribe) model.

Figure 7 Queue - The Point-to-Point Model



The Point-to-Point model (Figure 7), 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 98.)

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 31.)

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, 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 SeeBeyond 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 SeeBeyond JMS in XA mode. When running in XA mode, the EJBs subscribing or publishing to SeeBeyond 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 SeeBeyond JMS XA Resource is enlisted to a transaction so that the WebLogic Transaction Manager is aware of the SeeBeyond 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 **SeeBeyond Sample XA Session Beans** on page 106.)

Chapter 5

WebLogic eWay Component Communication

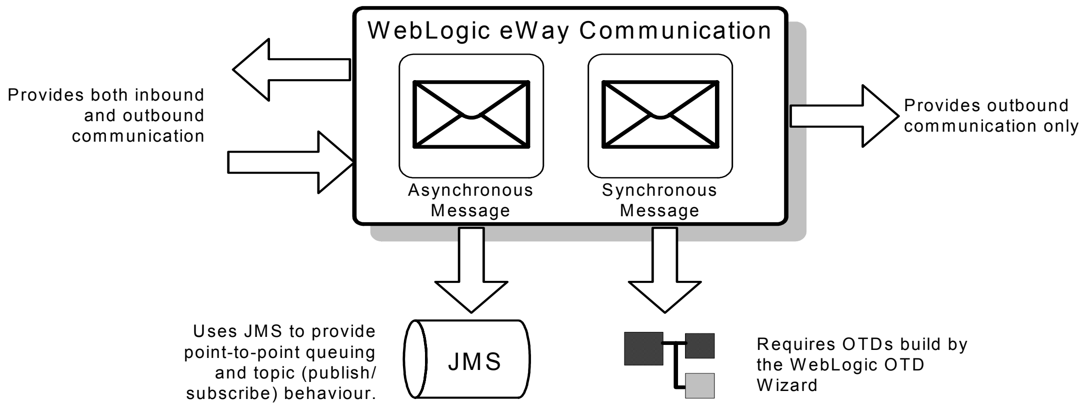
This chapter provides an overview of how components of the eWay Intelligent Adapter for WebLogic communicate with the WebLogic Application Server.

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 8 WebLogic Synchronous and Asynchronous Communication

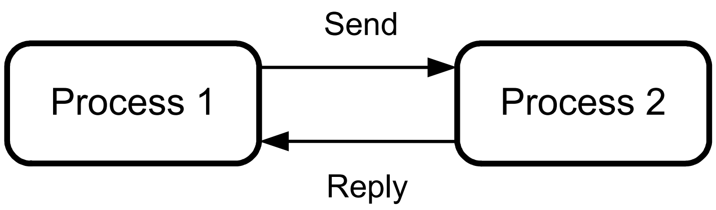


WebLogic eWay Communication
Provides both inbound and outbound communication
Asynchronous Message
Synchronous Message
Provides outbound communication only
Uses JMS to provide point-to-point queuing and topic (publish/ subscribe) behaviour.
JMS
Requires OTDs build by the WebLogic OTD Wizard

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 9.

Figure 9 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 36**.

Associated Sample Projects:

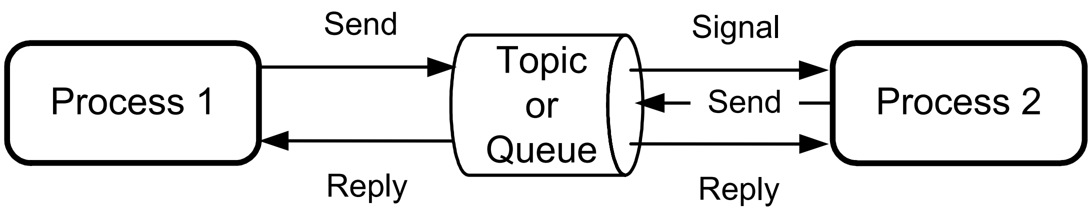
Two sample projects—**WebLogic_BPEL.zip** and **WebLogic_JCE.zip**—are included with the WebLogic eWay to demonstrate synchronous message interactions.

- **WebLogic_BPEL** – demonstrates how to deploy an eGate component as an Activity in an eInsight Business Process. For more information, see **"Implementing the WebLogic eWay" on page 75**
- **WebLogic_JCE** – demonstrates how to deploy and eGate component using java collaborations. For more information, see **"Using the Sample Projects in eGate" on page 84**.

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 10 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 37**.
- **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 37**.

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.

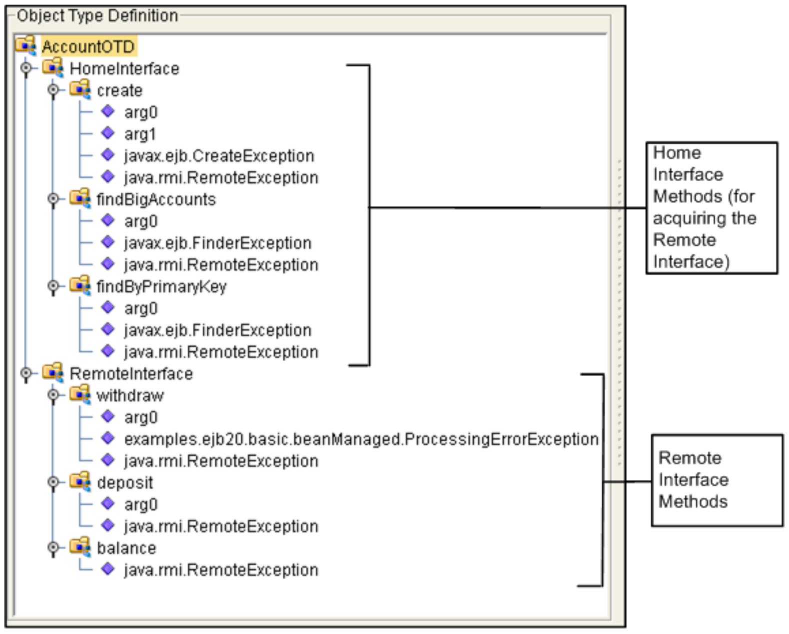
The OTD is divided into two portions:

- **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 11, 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 11 EJB OTD nodes represent both Home and Remote Interface methods



5.3 Asynchronous Communication in eGate

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

The eWay incorporates the SeeBeyond 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 SeeBeyond JMS**
- **Session Beans publishing/sending to SeeBeyond JMS**

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.

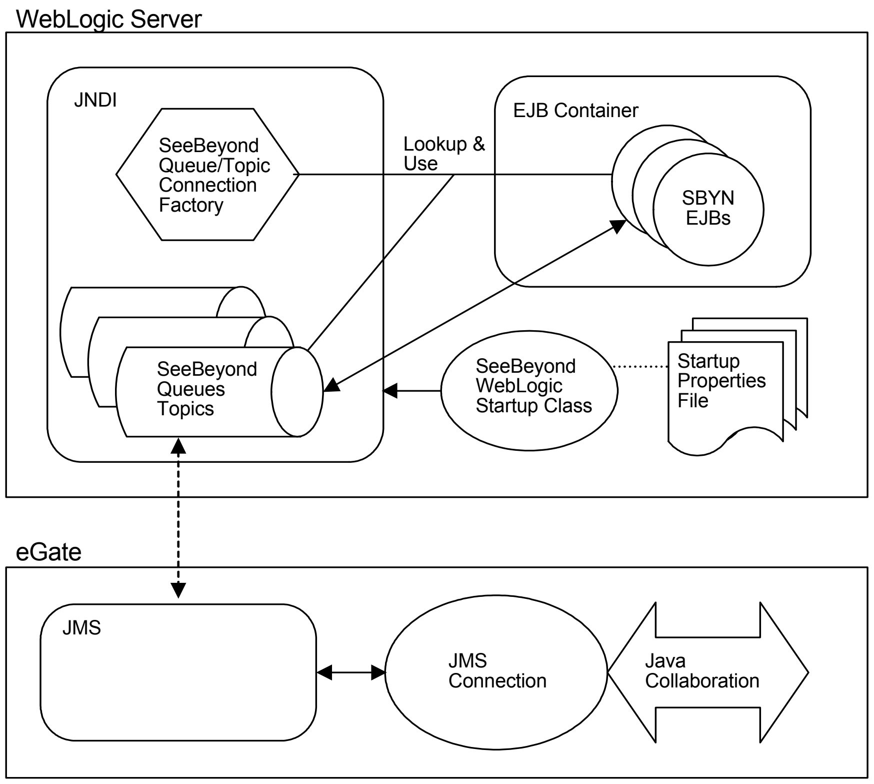
  The naming service allows you to "bind" the following SeeBeyond JMS objects:

  - **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 SeeBeyond JMS provider driving MDBs running in WebLogic. Session Beans are used for publishing and sending Topic/Queue messages through the SeeBeyond JMS provider as well.

  The following architectural diagram (Figure 12) illustrates the components involved:

Figure 12 WebLogic Server and WebLogic eWay Components



5.3.2. SeeBeyond JMS

As part of the WebLogic eWay installation, SeeBeyond 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 SeeBeyond 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 SeeBeyond **TopicConnectionFactory** to create the necessary JMS TopicConnection(s) and TopicSession(s) and uses the SeeBeyond **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 SeeBeyond **XAQueueConnectionFactory** is used to create the JMS

XAQueueConnection(s) and XAQueueSession(s). The **weblogic-ejb-jar.xml** deployment descriptor allows the configuration of SeeBeyond JMS as a foreign JMS to which the MDBs subscribe. The diagram in Figure 13 shows the components involved in eGate to WebLogic mode. The arrows represent message flow.

Figure 13 Message Flow from eGate to WebLogic

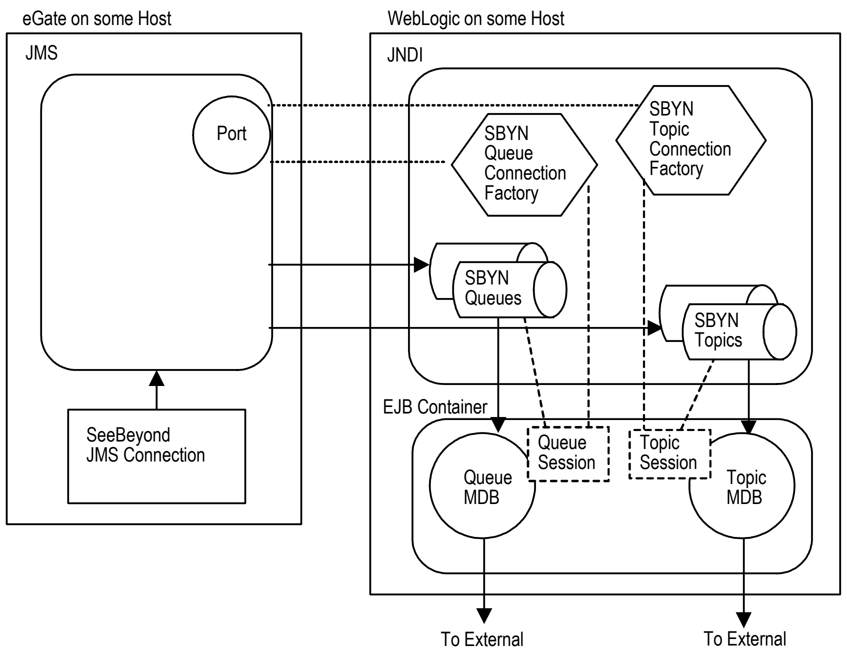
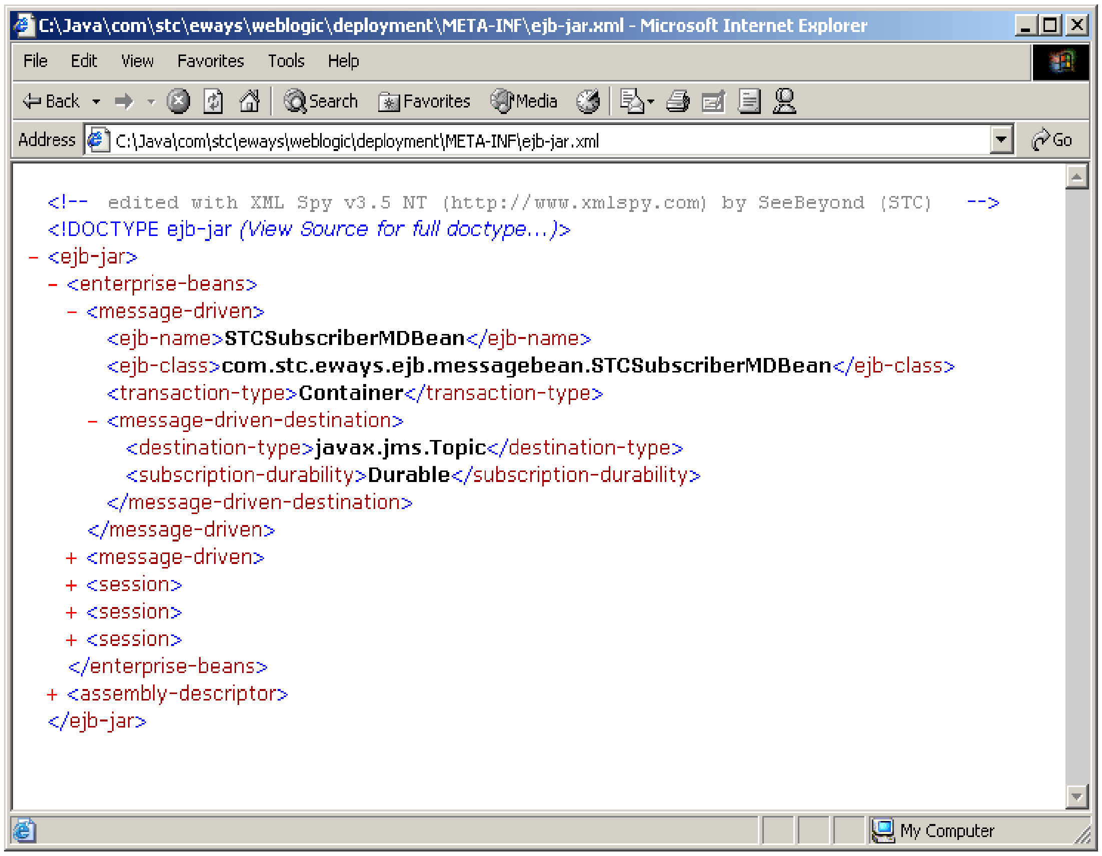


Figure 14 displays an example of the **ejb-jar.xml** for the Topic MDB which receives messages from a SeeBeyond JMS Topic.

Figure 14 ejb-jar.xml - Topic MDB

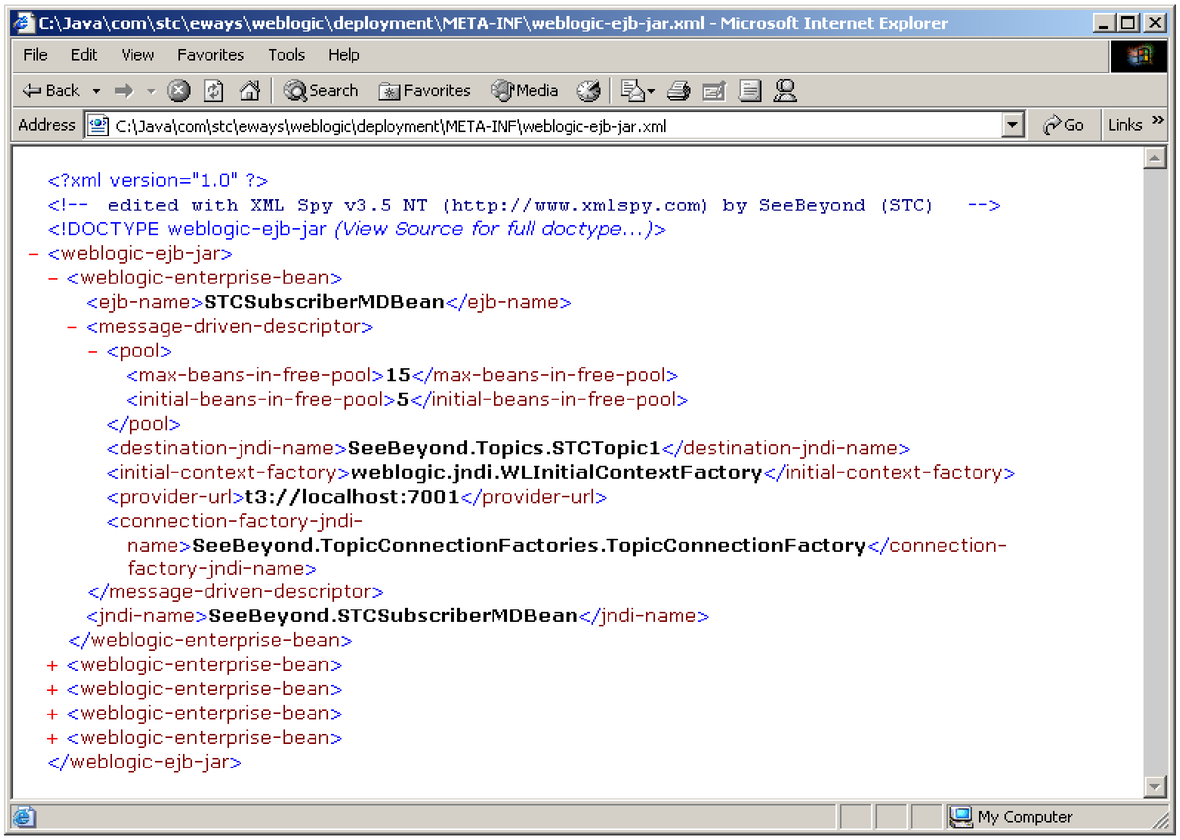

C:\Java\com\stc\eways\weblogic\deployment\META-INF\ejb-jar.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\Java\com\stc\eways\weblogic\deployment\META-INF\ejb-jar.xml
Go
My Computer

```
  <!-- edited with XML Spy v3.5 NT (http://www.xmlspy.com) by SeeBeyond (STC)  -->
  <!DOCTYPE ejb-jar (View Source for full doctype...)>
- <ejb-jar>
  - <enterprise-beans>
    - <message-driven>
        <ejb-name>STCSubscriberMDBean</ejb-name>
        <ejb-class>com.stc.eways.ejb.messagebean.STCSubscriberMDBean</ejb-class>
        <transaction-type>Container</transaction-type>
      - <message-driven-destination>
          <destination-type>javax.jms.Topic</destination-type>
          <subscription-durability>Durable</subscription-durability>
        </message-driven-destination>
      </message-driven>
    + <message-driven>
    + <session>
    + <session>
    + <session>
    </enterprise-beans>
  + <assembly-descriptor>
  </ejb-jar>
```

Updating the WebLogic JMS

An updated WebLogic JMS is required to ensure communication between eGate and WebLogic. Figure 15 displays an example of the **weblogic-ejb-jar.xml** for the Topic MDB which receives messages from a SeeBeyond JMS Topic.

Figure 15 weblogic-ejb-jar.xml - Topic MDB


C:\Java\com\stc\eways\weblogic\deployment\META-INF\weblogic-ejb-jar.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\Java\com\stc\eways\weblogic\deployment\META-INF\weblogic-ejb-jar.xml
Go Links
My Computer

```
<?xml version="1.0" ?>
<!-- edited with XML Spy v3.5 NT (http://www.xmlspy.com) by SeeBeyond (STC)   -->
<!DOCTYPE weblogic-ejb-jar (View Source for full doctype...)>
- <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>SeeBeyond.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>SeeBeyond.TopicConnectionFactories.TopicConnectionFactory</connection-
          factory-jndi-name>
      </message-driven-descriptor>
      <jndi-name>SeeBeyond.STCSubscriberMDBean</jndi-name>
    </weblogic-enterprise-bean>
  + <weblogic-enterprise-bean>
  + <weblogic-enterprise-bean>
  + <weblogic-enterprise-bean>
  + <weblogic-enterprise-bean>
  </weblogic-ejb-jar>
```

In the above figure, the <destination-jndi-name> tag of the Topic is **SeeBeyond.Topics.STCTopic1**; this is a SeeBeyond JMS Topic. Using the WebLogic naming service, the two entries *initial-context-factory* and *provider-url* are **weblogic.jndi.WLInitialContextFactory** and **t3://localhost:7003** respectively. Since the container needs to use the SeeBeyond **JMS TopicConnectionFactory**, we specify the SeeBeyond **TopicConnectionFactory** with the <connection-factory-jndi-name> tag as **SeeBeyond.TopicConnectionFactories.TopicConnectionFactory**. The JNDI bound objects **SeeBeyond.Topics.STCTopic1** and **SeeBeyond.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 does NOT allow creation of any foreign JMS objects. This must be done outside of the Administrative Console.

The task of creating the SeeBeyond JMS objects is done by the SeeBeyond WebLogic startup class called **STCWLStartup**. (See the section **SeeBeyond WebLogic Startup Class** on page 47 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 SeeBeyond JMS is being used as a foreign JMS into WebLogic.

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

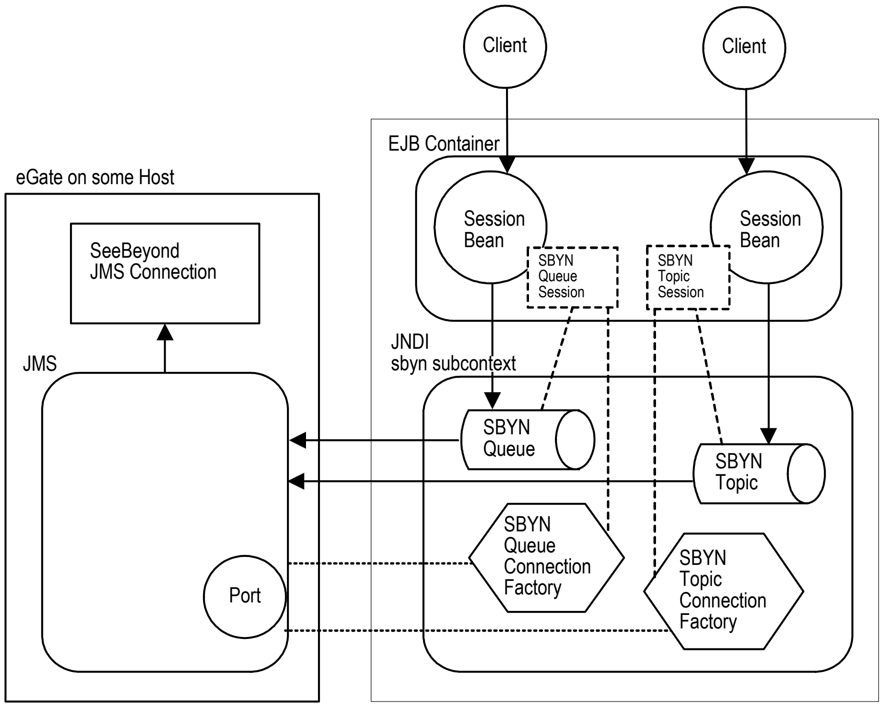
Message Flow from WebLogic to eGate Using JMS Objects

For message flow from WebLogic to eGate, Session Beans can publish/send JMS messages to SeeBeyond 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 SeeBeyond JMS objects and JNDI bindings is done through the SeeBeyond WebLogic startup class, **STCWLStartup**. (See **SeeBeyond WebLogic Startup Class** on page 47 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 16 displays a diagram of the components involved for the WebLogic to eGate mode. The arrows represent the message flow.

Figure 16 Message Flow from WebLogic to eGate

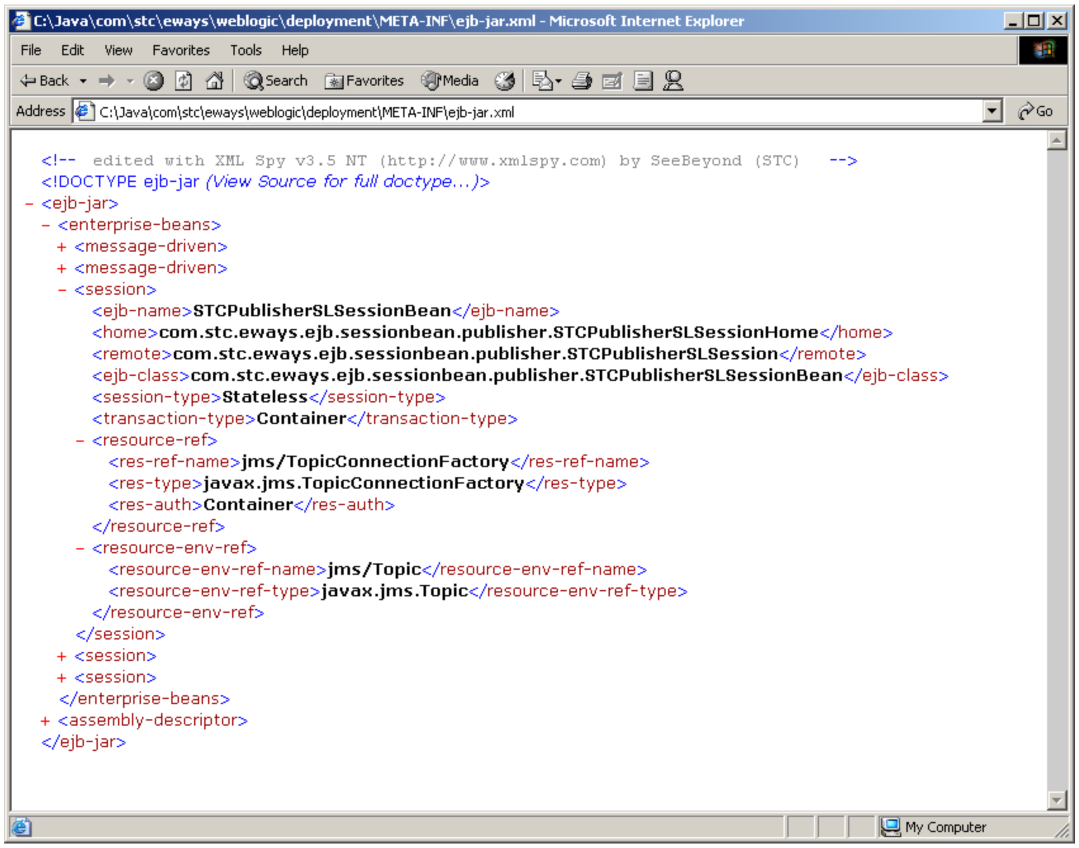


Every bean automatically has access to a special naming system called the **Environment Naming Context** (ENC). The ENC is managed by the container and accessed by beans using JNDI. The JNDI ENC allows a bean to access resources like JDBC connections, other enterprise beans, and properties specific to 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 17 displays the Session Bean **ejb-jar.xml** deployment descriptor.

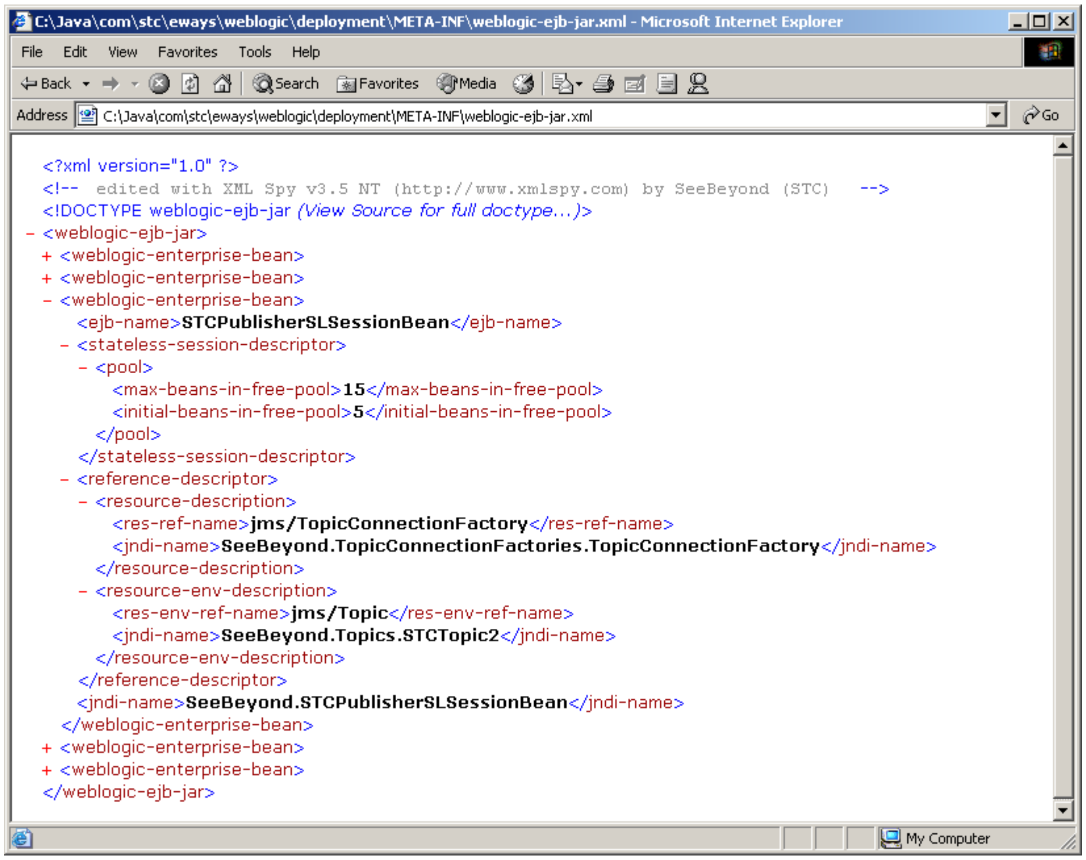
Figure 17 Session Bean ejb-jar.xml deployment descriptor


C:\Java\com\stc\eways\weblogic\deployment\META-INF\ejb-jar.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\Java\com\stc\eways\weblogic\deployment\META-INF\ejb-jar.xml
Go
My Computer

```
  <!-- edited with XML Spy v3.5 NT (http://www.xmlspy.com) by SeeBeyond (STC) -->
  <!DOCTYPE ejb-jar (View Source for full doctype...)>
- <ejb-jar>
  - <enterprise-beans>
    + <message-driven>
    + <message-driven>
    - <session>
        <ejb-name>STCPublisherSLSessionBean</ejb-name>
        <home>com.stc.eways.ejb.sessionbean.publisher.STCPublisherSLSessionHome</home>
        <remote>com.stc.eways.ejb.sessionbean.publisher.STCPublisherSLSession</remote>
        <ejb-class>com.stc.eways.ejb.sessionbean.publisher.STCPublisherSLSessionBean</ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Container</transaction-type>
      - <resource-ref>
          <res-ref-name>jms/TopicConnectionFactory</res-ref-name>
          <res-type>javax.jms.TopicConnectionFactory</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>
    + <session>
    + <session>
    </enterprise-beans>
  + <assembly-descriptor>
  </ejb-jar>
```

Figure 18 displays the Session Bean **weblogic-ejb-jar.xml** deployment descriptor.

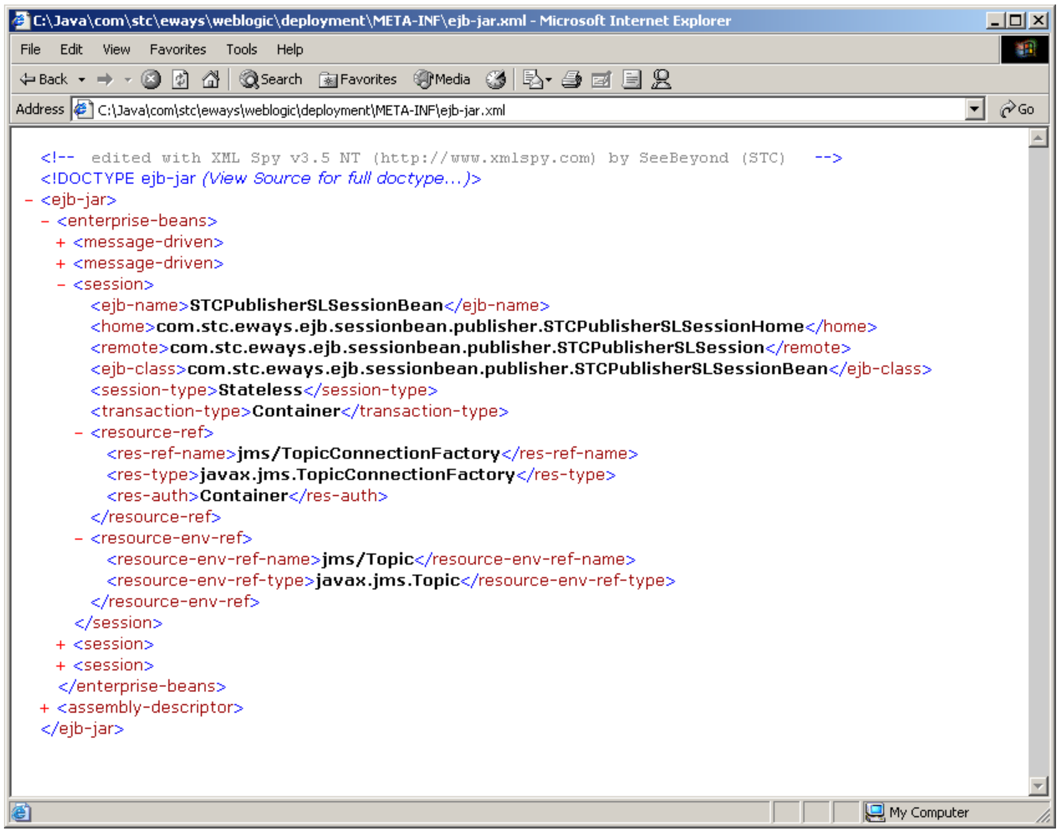
Figure 18 Session Bean weblogic-ejb-jar.xml deployment descriptor


C:\Java\com\stc\eways\weblogic\deployment\META-INF\weblogic-ejb-jar.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\Java\com\stc\eways\weblogic\deployment\META-INF\weblogic-ejb-jar.xml
Go
My Computer

```
  <?xml version="1.0" ?>
  <!--  edited with XML Spy v3.5 NT (http://www.xmlspy.com) by SeeBeyond (STC)   -->
  <!DOCTYPE weblogic-ejb-jar (View Source for full doctype...)>
- <weblogic-ejb-jar>
  + <weblogic-enterprise-bean>
  + <weblogic-enterprise-bean>
  - <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>SeeBeyond.TopicConnectionFactories.TopicConnectionFactory</jndi-name>
        </resource-description>
      - <resource-env-description>
          <res-env-ref-name>jms/Topic</res-env-ref-name>
          <jndi-name>SeeBeyond.Topics.STCTopic2</jndi-name>
        </resource-env-description>
      </reference-descriptor>
      <jndi-name>SeeBeyond.STCPublisherSLSessionBean</jndi-name>
    </weblogic-enterprise-bean>
  + <weblogic-enterprise-bean>
  + <weblogic-enterprise-bean>
  </weblogic-ejb-jar>
```

Figure 19 displays an example of the **ejb-jar.xml** deployment descriptor for the Session Bean publishing to a SeeBeyond JMS Topic:

Figure 19 ejbjar.xml deployment descriptor - Session Bean to SeeBeyond JMS Topic

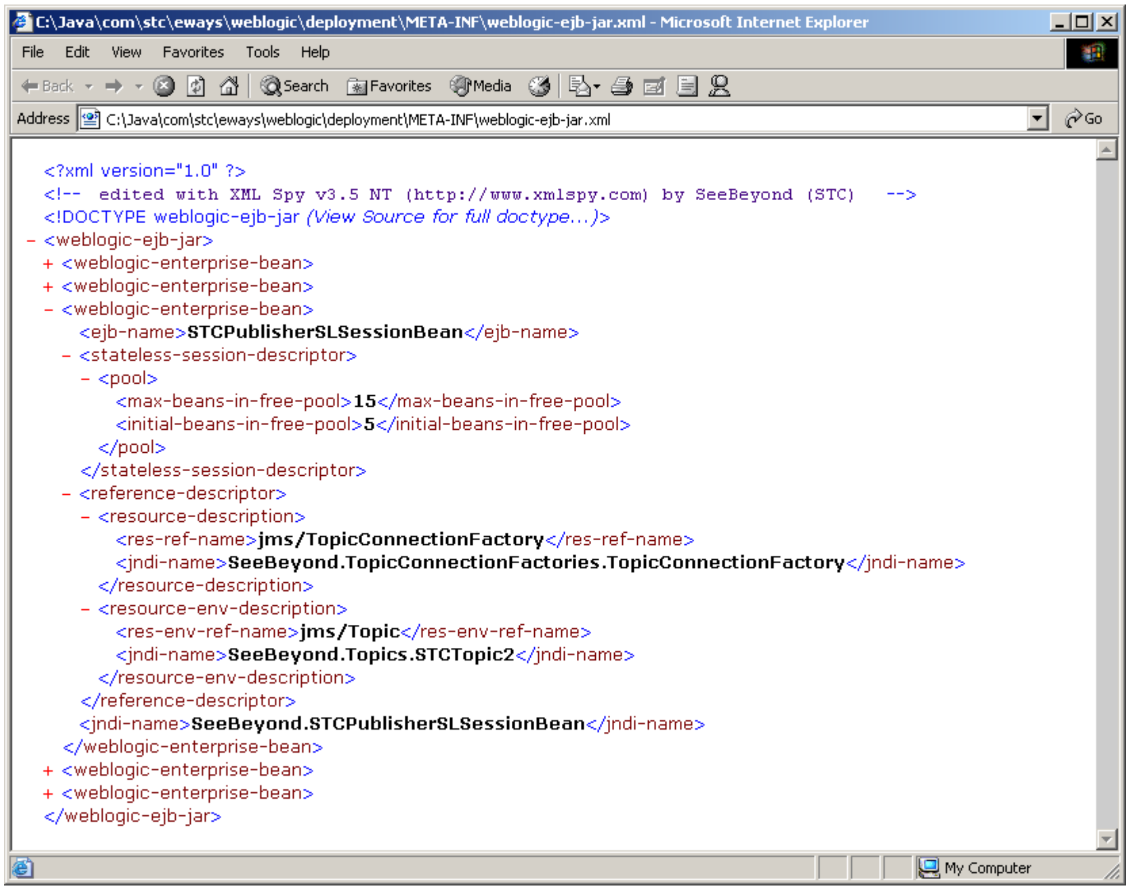

C:\Java\com\stc\eways\weblogic\deployment\META-INF\ejb-jar.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\Java\com\stc\eways\weblogic\deployment\META-INF\ejb-jar.xml
Go
My Computer

```
  <!-- edited with XML Spy v3.5 NT (http://www.xmlspy.com) by SeeBeyond (STC)  -->
  <!DOCTYPE ejb-jar (View Source for full doctype...)>
- <ejb-jar>
  - <enterprise-beans>
    + <message-driven>
    + <message-driven>
    - <session>
        <ejb-name>STCPublisherSLSessionBean</ejb-name>
        <home>com.stc.eways.ejb.sessionbean.publisher.STCPublisherSLSessionHome</home>
        <remote>com.stc.eways.ejb.sessionbean.publisher.STCPublisherSLSession</remote>
        <ejb-class>com.stc.eways.ejb.sessionbean.publisher.STCPublisherSLSessionBean</ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Container</transaction-type>
      - <resource-ref>
          <res-ref-name>jms/TopicConnectionFactory</res-ref-name>
          <res-type>javax.jms.TopicConnectionFactory</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>
    + <session>
    + <session>
    </enterprise-beans>
  + <assembly-descriptor>
  </ejb-jar>
```

The value for the <res-ref-name> tag is **jms/TopicConnectionFactory** and the value for the <resource-env-ref-name> environment entry tag is **jsm/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 20.

Figure 20 weblogic-ejb-jar.xml defines the actual JNDI name


C:\Java\com\stc\eways\weblogic\deployment\META-INF\weblogic-ejb-jar.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\Java\com\stc\eways\weblogic\deployment\META-INF\weblogic-ejb-jar.xml
Go
My Computer

```
  <?xml version="1.0" ?>
  <!-- edited with XML Spy v3.5 NT (http://www.xmlspy.com) by SeeBeyond (STC)   -->
  <!DOCTYPE weblogic-ejb-jar (View Source for full doctype...)>
- <weblogic-ejb-jar>
  + <weblogic-enterprise-bean>
  + <weblogic-enterprise-bean>
  - <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>SeeBeyond.TopicConnectionFactories.TopicConnectionFactory</jndi-name>
        </resource-description>
      - <resource-env-description>
          <res-env-ref-name>jms/Topic</res-env-ref-name>
          <jndi-name>SeeBeyond.Topics.STCTopic2</jndi-name>
        </resource-env-description>
      </reference-descriptor>
      <jndi-name>SeeBeyond.STCPublisherSLSessionBean</jndi-name>
    </weblogic-enterprise-bean>
  + <weblogic-enterprise-bean>
  + <weblogic-enterprise-bean>
  </weblogic-ejb-jar>
```

The value for the jndi-name tag for the resource name **jms/TopicConnectionFactory** is **SeeBeyond.TopicConnectionFactories.TopicConnectionFactory** and the value for the **jndi-name** tag for the **jms/Topic** entry is **SeeBeyond.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 SeeBeyond WebLogic Startup Class

To bind the SeeBeyond JMS objects into the WebLogic naming service, a SeeBeyond 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 SeeBeyond MyTopicConnectionFactory**
- **A SeeBeyond 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 **STCWLStartup.class**. It implements the **weblogic.common.T3StartupDef interface**. The **STCWLStartup.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 SeeBeyond 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, **STCWLStartup.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 SeeBeyond 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 **STCWLStartup.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.

STCWLStartup.properties File

SeeBeyond JNDI Sub-context

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

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

Subcontext.SeeBeyond=SeeBeyond
```

The user should not have to change this.

SeeBeyond JMS TopicConnectionFactory Sub-context

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

```
#--------------------------------------------------------------------
#
# JNDI subcontext for SeeBeyond JMS Topic connection factories.
# This section configures the JNDI subcontext to which all the
SeeBeyond
# 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.

SeeBeyond JMS QueueConnectionFactory Sub-context

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

```
#-------------------------------------------------------------------
#
# JNDI subcontext for SeeBeyond JMS Queue connection factories.
# This section configures the JNDI subcontext to which all the 
SeeBeyond
# 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.

SeeBeyond JMS Topic Sub-context

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

```
#-------------------------------------------------------------------
#
# JNDI subcontext for SeeBeyond JMS Topics.
# This section configures the JNDI subcontext to which all the 
SeeBeyond
# 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.

SeeBeyond JMS Queue Sub-context

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

```
#-------------------------------------------------------------------
#
# JNDI subcontext for SeeBeyond JMS Queues.
# This section configures the JNDI subcontext to which all the 
SeeBeyond
# 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.

SeeBeyond 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: SeeBeyondJMS&MyJMS&JMSOnHostA
#---------------------------------------------------------------------

JMSServerNames=SeeBeyondJMS&MyJMS
```

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.

SeeBeyond 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 are used to configure JMS Connection Factories:
#  TopicConnectionFactory, QueueConnectionFactory
#  XATopicConnectionFactory, XAQueueConnectionFactory
#
#---------------------------------------------------------------------

! SeeBeyondJMS Server configuration
! Notice that "SeeBeyondJMS" is in the JMS Server Names list.
SeeBeyondJMS.Host=localhost
SeeBeyondJMS.Port=18007
SeeBeyondJMS.TopicConnectionFactory=TopicConnectionFactory
SeeBeyondJMS.QueueConnectionFactory=QueueConnectionFactory
SeeBeyondJMS.XATopicConnectionFactory=XATopicConnectionFactory
SeeBeyondJMS.XAQueueConnectionFactory=XAQueueConnectionFactory


! MyJMS Server configuration
! Notice that "MyJMS" is in the JMS Server Names list.
MyJMS.Host=localhost
MyJMS.Port=9876
```

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 **SeeBeyond.TopicConnectionFactories.TopicConnectionFactory** as the JNDI name for the **TopicConnectionFactory**. Notice that the SeeBeyond sub-context and the **TopicConnectionFactories** sub-context are pre-pended.

SeeBeyond JMS Topic Destinations

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

```
#---------------------------------------------------------------------
#
# SeeBeyond JMS Topics
# This section configures the SeeBeyond 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 SeeBeyond 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 38 and **Message Flow from WebLogic to eGate Using JMS Objects** on page 42 for more information on the EJB deployment descriptors.

SeeBeyond JMS Queue Destinations

```
The next section allows the user to specify the Queues to create and
bind to JNDI:
#----------------------------------------------------------------------
#
# SeeBeyond JMS Queues
# This section configures the SeeBeyond 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
```

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 SeeBeyond 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 38 and **Message Flow from WebLogic to eGate Using JMS Objects** on page 42 for more information on the EJB deployment descriptors.

Chapter 6

Configuring WebLogic Server

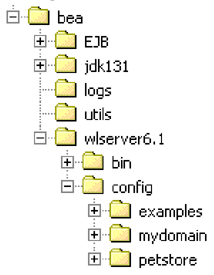
The following chapter provides directions for configuring WebLogic Server for asynchronous interaction with eGate. Setup directions are provided for both WebLogic version 6.1 and 7.0.

What's in This Chapter:

6.1 Configuration for WebLogic 6.1

WebLogic Server 6.1 installation creates a home or root directory named "**bea**" by default (this name may be changed during installation). Under the Home directory first open the **wlserver6.1** directory, then open the **config** directory. Sample servers created on WebLogic Server are located in the config directory (see Figure 21).

Figure 21 WebLogic Server 6.1 File Structure



1 Verify that the system classpath contains ejb.jar and weblogic.jar (with ejb.jar proceeding weblogic.jar).

2 Copy the following files to WebLogic's <BEA-HOME>\wlserver6.1\lib directory.

- com.stc.jms.stcjms.jar
- stcejbweblogic.jar
- stcwlstartup.jar
- STCWLStartup.properties

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

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

stcejbweb

+logic.jar, stcwlstartup.jar, and STCWLStartup.properties can be found at:

```
edesigner\usrdir\modules\ext\weblogic
```

3 Copy the JSSE.jar file from: `repository\jre\1.4.2\lib` and place into `bea\wlserver61\lib`.

4 Modify **startExamplesServer.cmd** and **setExamplesServer.cmd** located at <WL-HOME/config/examples. Append com.stc.jms.stcjms.jar and stcwlstartup.jar to the classpath as follows:

For startExamplesServer.cmd

```
CLASSPATH=.;.\lib\weblogic_sp.jar;.\lib\weblogic.jar;.\samples\eva
l\cloudscape\lib\cloudscape.jar;.\config\examples\serverclasses;.\
lib\com.stc.jms.stcjms.jar;.\lib\stcwlstartup.jar
```

For setExampleEnv.cmd

```
setCLASSPATH=%CLASSPATH%;%WL_HOME%\lib\com.stc.jms.stcjms.jar;%WL_
HOME%\lib\stcwlstartup.jar
```

com.stc.jms.stcjms.jar is located in the ..\eGate\server\regestry\repository\default\classes directory.

5 The sample EJBs have been configured to reference the T3 naming service that is running on the localhost at port 7003. By default, each WebLogic Server instance is installed to listen on port 7001. If your server instance is running, listening on port 7003, 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 stcejbweblogic.jar and edit **META-INF\weblogic-ejb-jar.xml**.

   B For each Bean that is run, find the Provider_URL tag of the deployment descriptor and change the port number from 7003 to 7001.

   C Re-jar (zip) the stcejbweblogic.jar.

6 Start an instance of the application server (in this case, Examples Server).

7 When the server has finished booting, start the Default Console. Go to Deployments, Startup & Shutdown, and click on **Configure a New Startup Class** (see **WebLogic Server Console - Create a New StartupClass** on page 55.) Enter the following Values:

Name: SeeBeyond_Startup

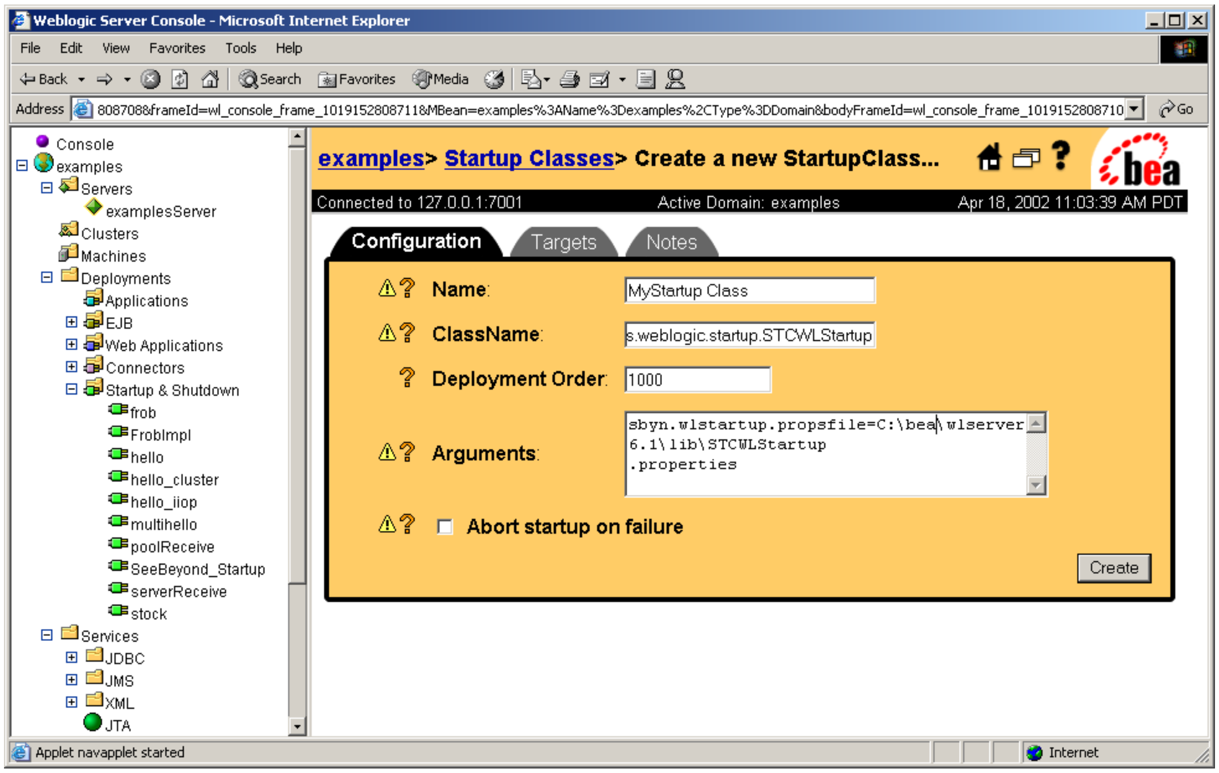
CLASSNAME: com.stc.eways.weblogic.startup.STCWLStartup

Deployment Order: 1000 (default)

Arguments: sbyn.wlstartup.propsfile=<*WL Home*>\wlserver6.1\lib\STCWLStartup
.properties (where <*WL Home*> is the home directory of WebLogic Server.)

Click **Create**.

Figure 22 WebLogic Server Console - Create a New StartupClass


Weblogic Server Console - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address 8087088&frameId=wl_console_frame_1019152808711&MBean=examples%3AName%3Dexamples%2CType%3DDomain&bodyFrameId=wl_console_frame_1019152808710
Console
examples
Servers
examplesServer
Clusters
Machines
Deployments
Applications
EJB
Web Applications
Connectors
Startup & Shutdown
frob
FrobImpl
hello
hello_cluster
hello_iiop
multihello
poolReceive
SeeBeyond_Startup
serverReceive
stock
Services
JDBC
JMS
XML
JTA
examples> Startup Classes> Create a new StartupClass...
Connected to 127.0.0.1:7001
Active Domain: examples
Apr 18, 2002 11:03:39 AM PDT
Configuration
Targets
Notes
Create
Applet navapplet started
Internet

Name: MyStartup Class

ClassName: s.weblogic.startup.STCWLStartup

Deployment Order: 1000

Arguments: sbyn.wlstartup.propsfile=C:\bea\wlserver6.1\lib\STCWLStartup.properties

Abort startup on failure

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

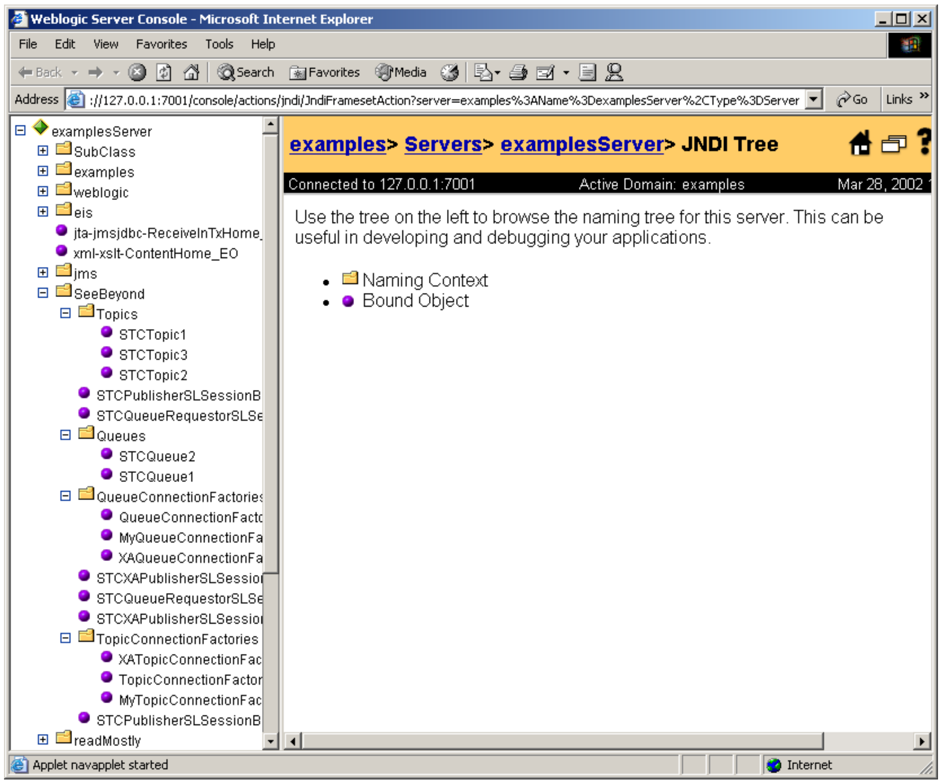
9 Stop and restart the server. If the startup class is successfully invoked, you should see:

```
STCWLStartup - SeeBeyond startup class invoked - STCWLStartup
STCWLStartup - Successfully invoked SeeBeyond startup
```

10 Start the Default Console.

11 In the Console, go to Servers, examplesServer (or the new server instance). Right-click exampleServer and select View JNDI Tree to open the JNDI Tree window. Expand the SeeBeyond node to verify that all SeeBeyond JMS objects are now available (see Figure 23).

Figure 23 View the JNDI Tree



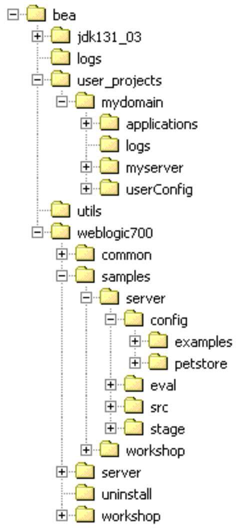
Use the tree on the left to browse the naming tree for this server. This can be useful in developing and debugging your applications.

12 On the Console, click on Examples, Deployments, EJB. Click on **Install a new EJB**. Browse to and select <WL-HOME>\wlserver6.1\lib\stcejbweblogic.jar. Click **Upload** to install it on the WebLogic Administration Server.

6.2 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 24).

Figure 24 WebLogic Server File Structure



1 Verify that the system classpath contains ejb.jar, weblogic.jar (with ejb.jar proceeding weblogic.jar in order), and stcejbweblogic.jar.

2 Copy the following files to the <BEA-HOME>\weblogic700\server\lib directory.

- com.stc.jms.stcjms.jar
- stcejbweblogic.jar
- stcwlstartup.jar
- STCWLStartup.properties

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

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

stcejbweblogic.jar, stcwlstartup.jar, and STCWLStartup.properties can be found at:

```
edesigner\usrdir\modules\ext\weblogic
```

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 stcwlstartup.jar to the classpath for each. For example:

For startExamplesServer.cmd

```
CLASSPATH=C:\bea\jdk131_03\lib\tools.jar;%POINTBASE_HOME%\lib\pbse
rver42ECF183.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\stcwlstartup.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\stcwlstartup.jar
```

5 The sample EJBs have been configured to reference the T3 naming service that is running on the localhost at port 7003. By default, each WebLogic Server instance is installed to listen on port 7001. If your server instance is running, listening on port 7003, 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 stcejbweblogic.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 **7003** to **7001**, and if necessary, change l**ocalhost** to the name of your specific computer.

   C Save, re-jar (zip), and replace stcejbweblogic.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 **WebLogic Server Console - Create a New StartupClass** on page 59.) Enter the following Values:

   Name: SeeBeyond_Startup

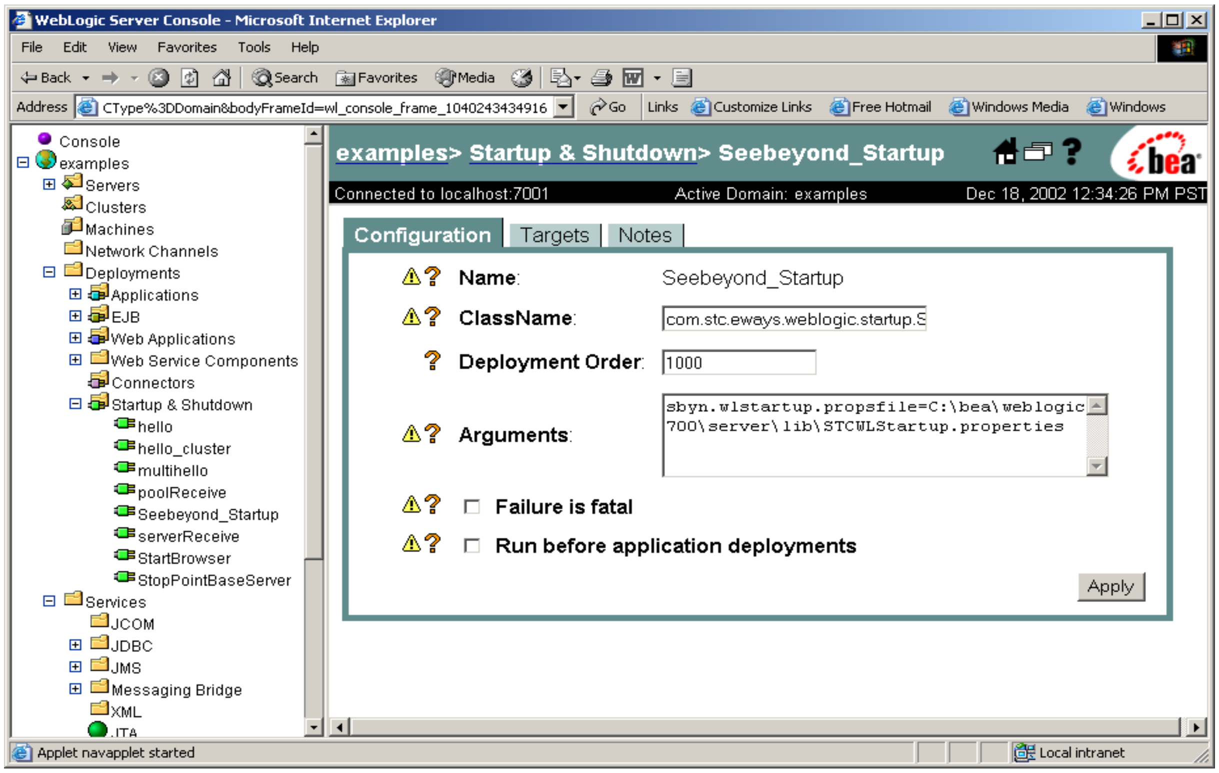
   CLASSNAME: com.stc.eways.weblogic.startup.STCWLStartup

   Deployment Order: 1000 (default)

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

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

Figure 25 WebLogic Server Console - Create a New StartupClass



Name: Seebeyond_Startup

ClassName: com.stc.eways.weblogic.startup.S

Deployment Order: 1000

Arguments:
```
sbyn.wlstartup.propsfile=C:\bea\weblogic
700\server\lib\STCWLStartup.properties
```

☐ Failure is fatal

☐ Run before application deployments

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.

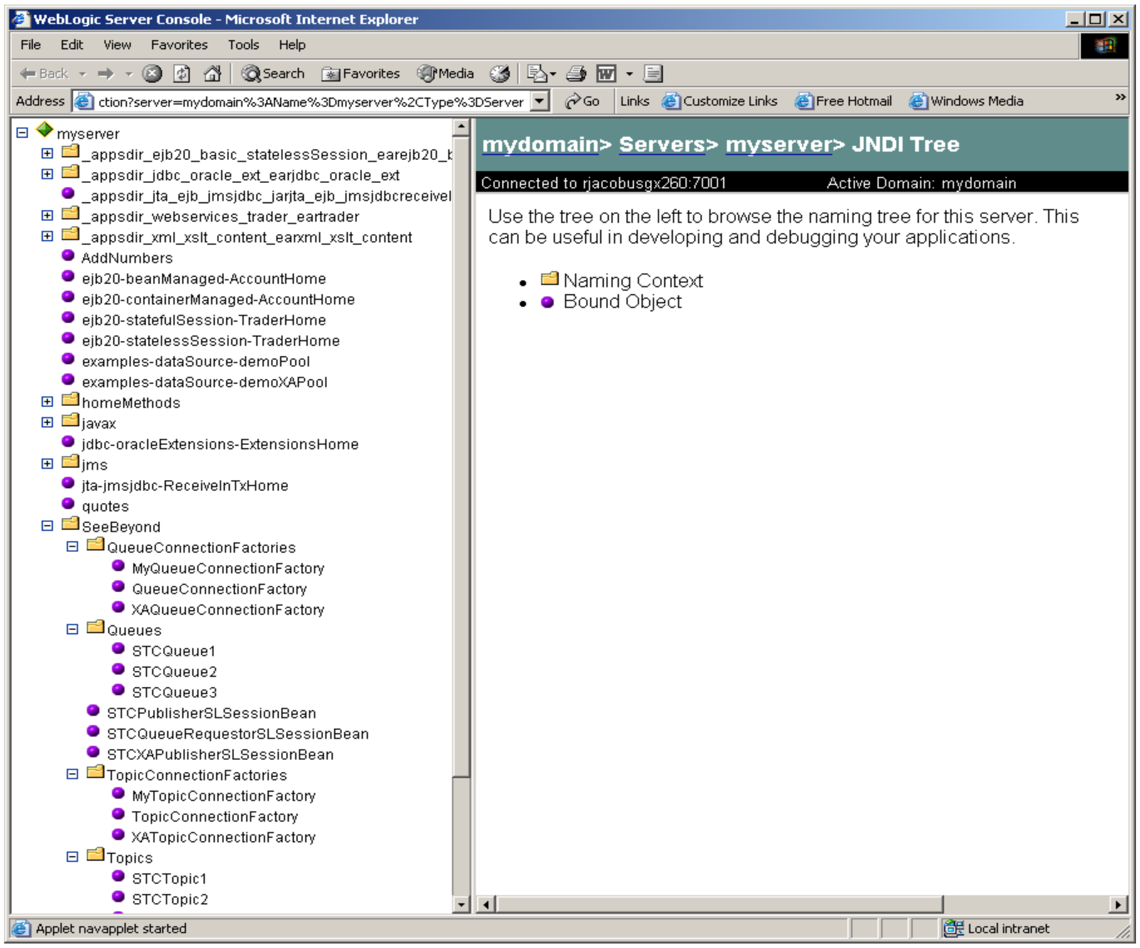
If the startup class is successfully invoked, you should see the following text in the Start Server command window:

```
STCWLStartup - SeeBeyond startup class invoked - STCWLStartup
STCWLStartup - Successfully invoked SeeBeyond 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 SeeBeyond node to verify that all SeeBeyond JMS objects are now available (see Figure 23).

Figure 26 View the JNDI Tree


WebLogic Server Console - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address ction?server=mydomain%3AName%3Dmyserver%2CType%3DServer
Go Links Customize Links Free Hotmail Windows Media
myserver
_appsdir_ejb20_basic_statelessSession_earejb20_b
_appsdir_jdbc_oracle_ext_earjdbc_oracle_ext
_appsdir_jta_ejb_jmsjdbc_jarjta_ejb_jmsjdbcreceivel
_appsdir_webservices_trader_eartrader
_appsdir_xml_xslt_content_earxml_xslt_content
AddNumbers
ejb20-beanManaged-AccountHome
ejb20-containerManaged-AccountHome
ejb20-statefulSession-TraderHome
ejb20-statelessSession-TraderHome
examples-dataSource-demoPool
examples-dataSource-demoXAPool
homeMethods
javax
jdbc-oracleExtensions-ExtensionsHome
jms
jta-jmsjdbc-ReceiveInTxHome
quotes
SeeBeyond
QueueConnectionFactories
MyQueueConnectionFactory
QueueConnectionFactory
XAQueueConnectionFactory
Queues
STCQueue1
STCQueue2
STCQueue3
STCPublisherSLSessionBean
STCQueueRequestorSLSessionBean
STCXAPublisherSLSessionBean
TopicConnectionFactories
MyTopicConnectionFactory
TopicConnectionFactory
XATopicConnectionFactory
Topics
STCTopic1
STCTopic2
mydomain> Servers> myserver> JNDI Tree
Connected to rjacobusgx260:7001
Active Domain: mydomain
Naming Context
Bound Object
Applet navapplet started
Local intranet

Use the tree on the left to browse the naming tree for this server. This can be useful in developing and debugging your applications.

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 (see* ***Executing the Schema*** *on page 143). 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\stcejbweblogic.jar. With the file selected, click **Upload**.

B Under **Step 2**, find **stcejbweblogic.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**.

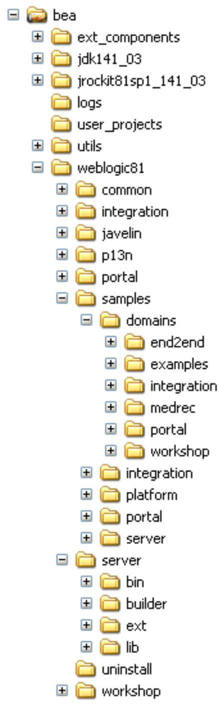
D Under **Step 4**, enter **stcejbweblogic** as the name for this application (EJB).

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

6.3 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 24).

Figure 27 WebLogic Server File Structure



1 Verify that the system classpath contains ejb.jar, weblogic.jar (with ejb.jar preceding weblogic.jar in order), and stcejbweblogic.jar.

2 Copy the following files to the **<BEA-HOME>\weblogic81\server\lib** directory:

- com.stc.jms.stcjms.jar
- stcejbweblogic.jar
- stcwlstartup.jar
- STCWLStartup.properties

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

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

stcejbweblogic.jar, stcwlstartup.jar, and STCWLStartup.properties can be found at:

```
edesigner\usrdir\modules\ext\weblogic
```

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 stcwlstartup.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\stcwlstartup.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\stcwls
tartup.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 stcejbweblogic.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 to **7001**, and if necessary, change **localhost** to the name of your specific computer.

C Save, re-jar (zip), and replace stcejbweblogic.jar.

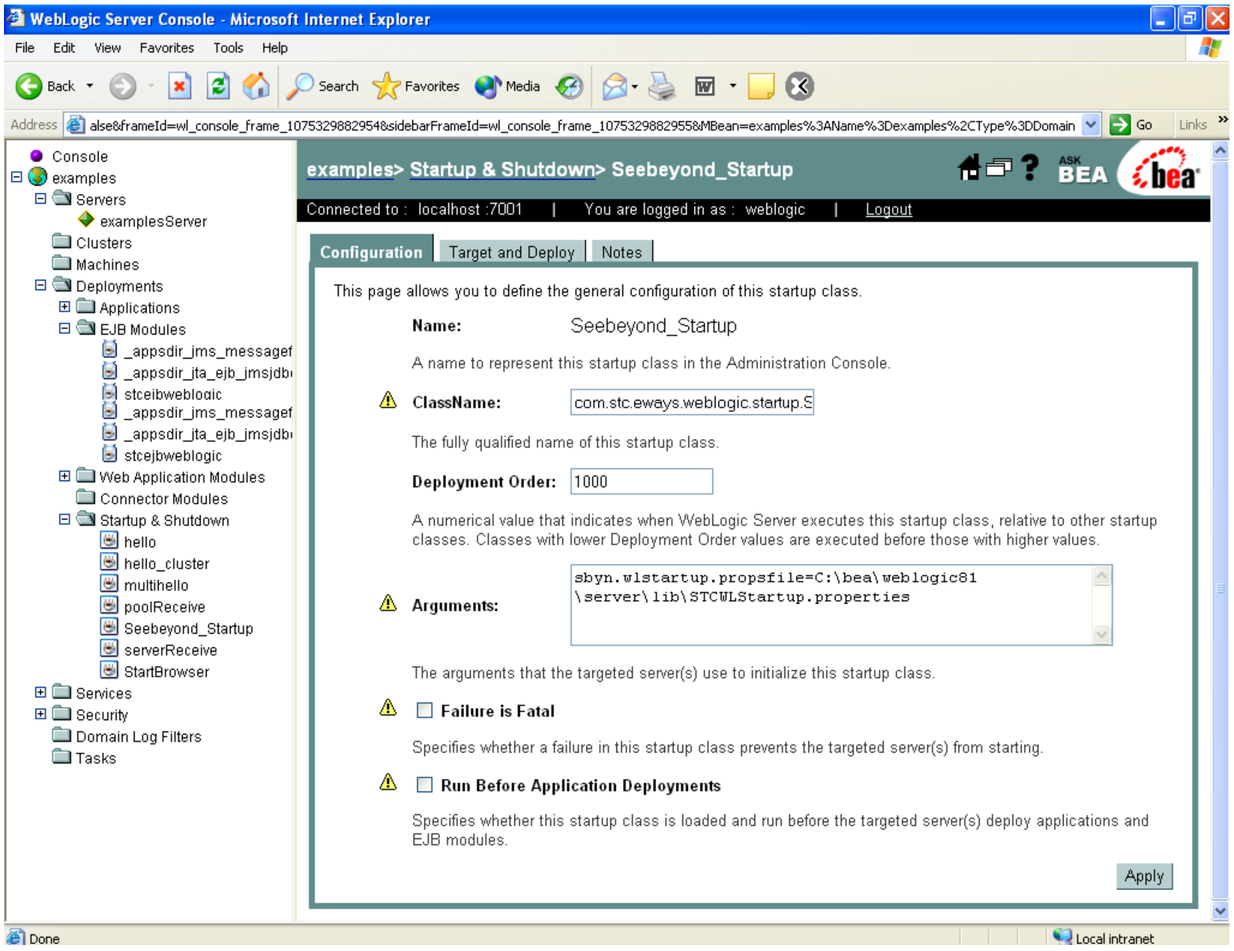
6 Start an instance of the application server (in this case, the user defined domain/server).

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 **WebLogic Server Console - Create a New StartupClass** on page 59.) Enter the following Values:

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

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

Figure 28 WebLogic Server Console - Create a New StartupClass



This page allows you to define the general configuration of this startup class.

Name: Seebeyond_Startup

A name to represent this startup class in the Administration Console.

ClassName: com.stc.eways.weblogic.startup.S

The fully qualified name of this startup class.

Deployment Order: 1000

A numerical value that indicates when WebLogic Server executes this startup class, relative to other startup classes. Classes with lower Deployment Order values are executed before those with higher values.

Arguments:

```
sbyn.wlstartup.propsfile=C:\bea\weblogic81
\server\lib\STCWLStartup.properties
```

The arguments that the targeted server(s) use to initialize this startup class.

☐ **Failure is Fatal**

Specifies whether a failure in this startup class prevents the targeted server(s) from starting.

☐ **Run Before Application Deployments**

Specifies whether this startup class is loaded and run before the targeted server(s) deploy applications and EJB modules.

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.

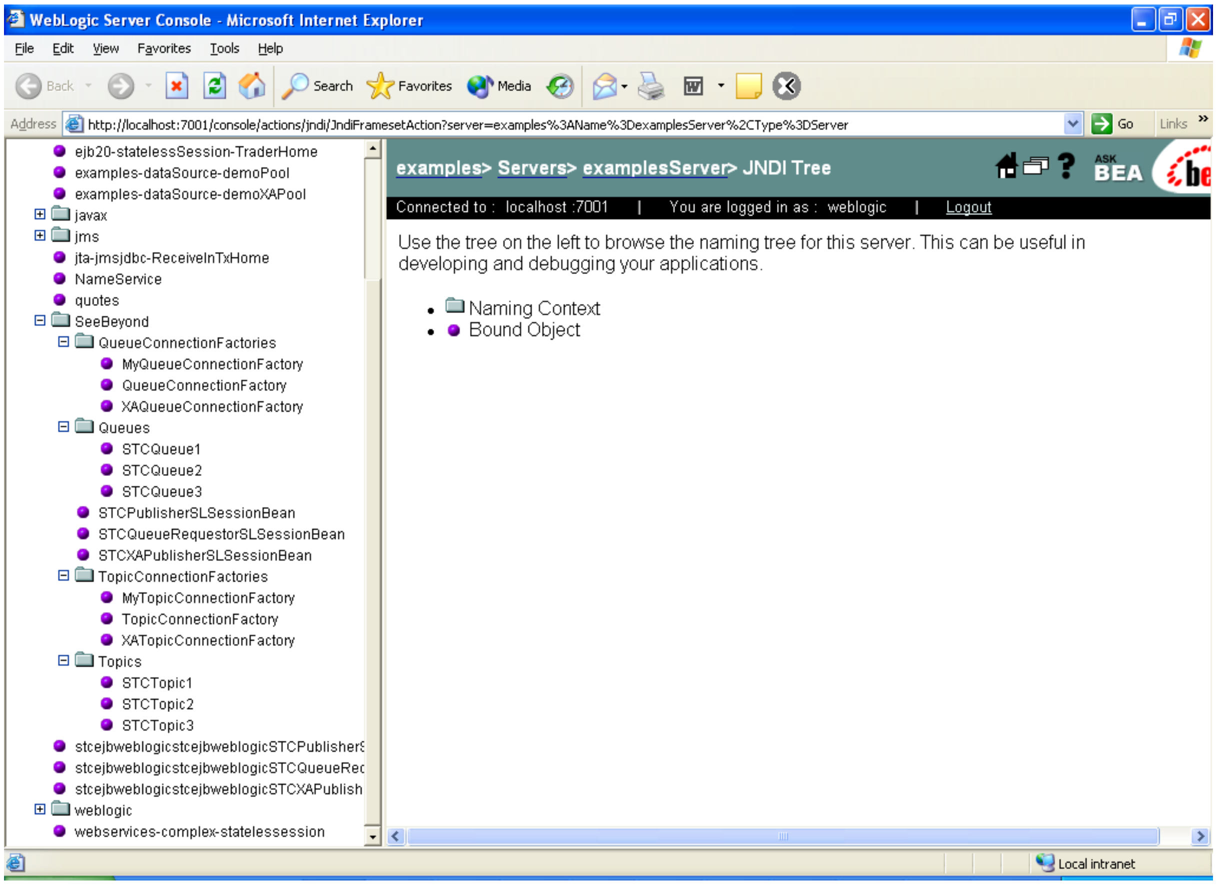
If the startup class is successfully invoked, you should see the following text in the Start Server command window:

```
STCWLStartup - SeeBeyond startup class invoked - Seebeyond_Startup
STCWLStartup - Topic name: Topic.Sample3
STCWLStartup - Topic name: Topic.Sample2
STCWLStartup - Topic name: Topic.Sample1
STCWLStartup - Queue name: Queue.Sample3
STCWLStartup - Queue name: Queue.Sample2
STCWLStartup - Queue name: Queue.Sample1
STCWLStartup - Successfully invoked SeeBeyond 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 SeeBeyond node to verify that all SeeBeyond JMS objects are now available (see Figure 23).

Figure 29 View the JNDI Tree


WebLogic Server Console - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address
Go
Links
examples> Servers> examplesServer> JNDI Tree
ASK BEA
Local intranet

http://localhost:7001/console/actions/jndi/JndiFramesetAction?server=examples%3AName%3DexamplesServer%2CType%3DServer

- ejb20-statelessSession-TraderHome
- examples-dataSource-demoPool
- examples-dataSource-demoXAPool
- javax
- jms
- jta-jmsjdbc-ReceiveInTxHome
- NameService
- quotes
- SeeBeyond
  - QueueConnectionFactories
    - MyQueueConnectionFactory
    - QueueConnectionFactory
    - XAQueueConnectionFactory
  - Queues
    - STCQueue1
    - STCQueue2
    - STCQueue3
  - STCPublisherSLSessionBean
  - STCQueueRequestorSLSessionBean
  - STCXAPublisherSLSessionBean
  - TopicConnectionFactories
    - MyTopicConnectionFactory
    - TopicConnectionFactory
    - XATopicConnectionFactory
  - Topics
    - STCTopic1
    - STCTopic2
    - STCTopic3
- stcejbweblogicstcejbweblogicSTCPublisherS
- stcejbweblogicstcejbweblogicSTCQueueRec
- stcejbweblogicstcejbweblogicSTCXAPublish
- weblogic
- webservices-complex-statelessession

Connected to : localhost :7001 | You are logged in as : weblogic | Logout

Use the tree on the left to browse the naming tree for this server. This can be useful in developing and debugging your applications.

- Naming Context
- Bound Object

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\stcejbweblogic.jar. With the file selected, click **Upload**.
5. Select the uploaded **stcejbweblogic.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 **stcejbweblogic** 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

This chapter describes how to build and use Object Type Definitions (OTDs) using the WebLogic OTD Wizard.

What's in This Chapter:

7.1 Java Methods for the OTD Wizard

Field nodes are added to the OTD based on the Tables in the external data source. Java method and parameter nodes are added to provide the appropriate JDBC functionality. For more information about the Java methods, refer to your JDBC developer's reference.

Note: *Java classes provided in the WebLogic OTD Wizard can contain APIs created using standard Sun JDK 1.3.x or JDK 1.4.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: *Consult the Sun Microsystems Javadoc for the latest API documentation for certain restrictions imposed by the J2EE specification.*

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.*

7.2 Creating the OTD

OTDs contain the data structure and rules that define an object. The OTD Wizard creates OTDs based on EJB archive files (and theirs dependent classes, if applicable).

Steps used to create an OTD include:

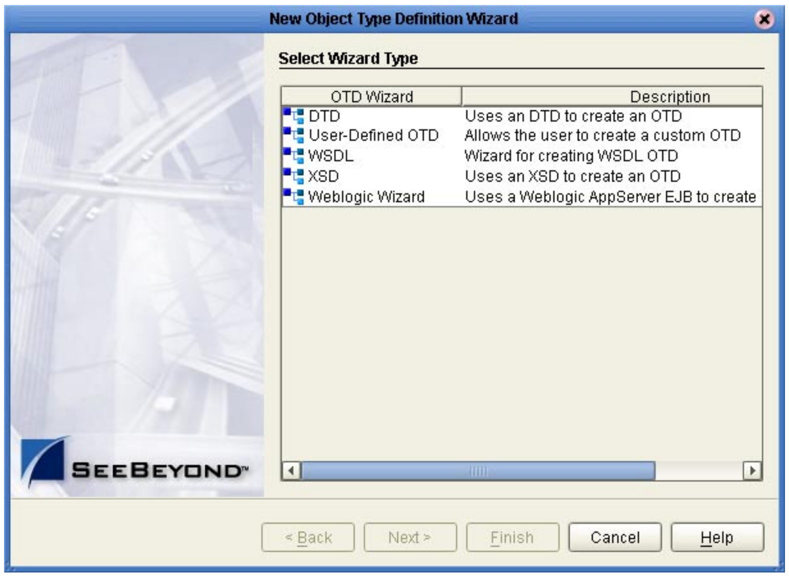
- **Select Wizard Type** on page 67
- **Specify OTD Name** on page 68
- **Select Interfaces** on page 68
- **Select Method Argument** on page 71
- **Select Class Path** on page 71
- **Review Selections** on page 74

7.2.1. Select Wizard Type

1 On the Enterprise Explorer, right click on the project and select **New > Object Type Definition** from the menu.

2 The **Select Wizard Type** window appears, displaying the available **OTD** wizards. See **Figure 30 on page 67**.

Figure 30 OTD Wizard Selection

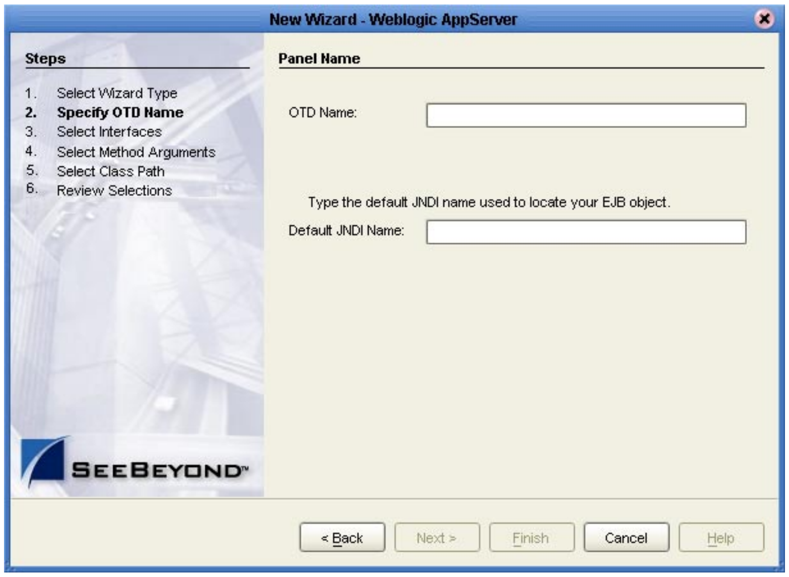


OTD Wizard	Description
DTD	Uses an DTD to create an OTD
User-Defined OTD	Allows the user to create a custom OTD
WSDL	Wizard for creating WSDL OTD
XSD	Uses an XSD to create an OTD
Weblogic Wizard	Uses a Weblogic AppServer EJB to create

3 From the list, select the **WebLogic Wizard** OTD and click **Next**. The **Specify OTD Name** window appears.

7.2.2. Specify OTD Name

Figure 31 Database Connection Information



4 Enter an OTD name and type the Java Naming and Directory Interface (JNDI) name used to locate your Enterprise Java Bean (EJB) object.

5 Click **Next**, the **Select Interfaces** window appears.

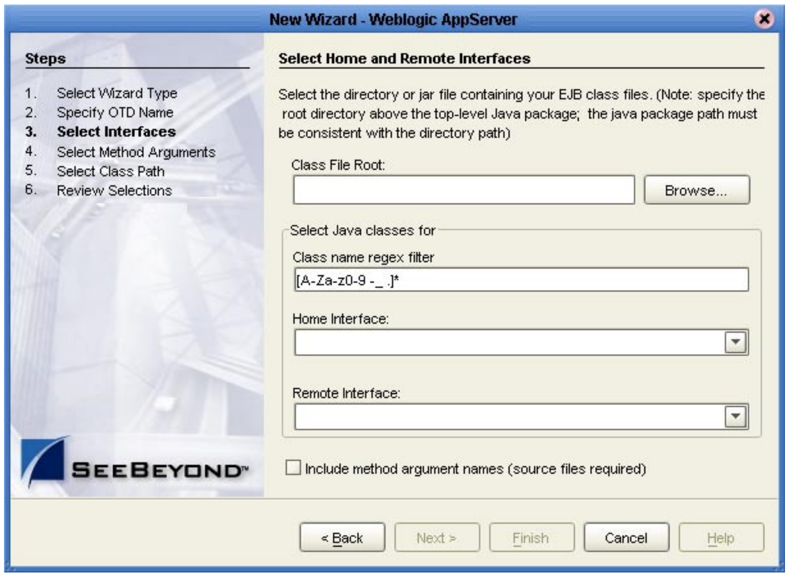
7.2.3. Select Interfaces

The Select Interfaces window requires locating and selecting the root directory or specific jar file containing the EJB class files.

Available Fields Include:

- **Class File Root** – contains the path of the EJB jar file or a directory path that includes the EJB, and other dependent jar files.
- **Class name regex filter** – used as an interface name pattern filter when locating EJB class files. A regular expression can be entered to describe home and remote interface name patterns for matching optimization. Wildcard-like (*) patterns, used as a suffix, are allowed in this field for matching purposes.
- **Home Interface** – used to define the enterprise bean’s life-cycle methods that are used to create, remove, and find beans.
- **Remote Interface** – used to define an enterprise bean’s business methods, which are used by bean clients to interact with the bean.

Figure 32 Select Interfaces window



Select the directory or jar file containing your EJB class files. (Note: specify the root directory above the top-level Java package; the java package path must be consistent with the directory path)

Class name regex filter: [A-Za-z0-9 -_ .]*

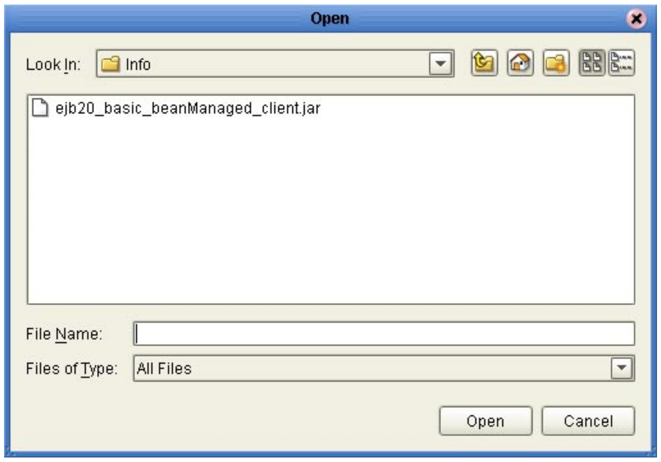
6 Click the **Browse** button next to the Class File Root text box, and in the Open window, navigate to the required EJB jar file. The EJB jar file must contain both Home and Remote Interface classes for the EJB.

If you provide an EJB class file that does not contain all necessary dependent class files required for the Home or Remote Interface, the WebLogic OTD Wizard prompts a Confirm Dependent Class window, allowing you to provide additional dependent class files. The Confirm Dependent Class window includes the following buttons:

- **Yes** – choose Yes to provide the path and name of required jar files.
- **No** – choose No to ignore a class that has a dependency.
- **Cancel** – choose Cancel to ignore all classes with dependencies.

Note: 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.

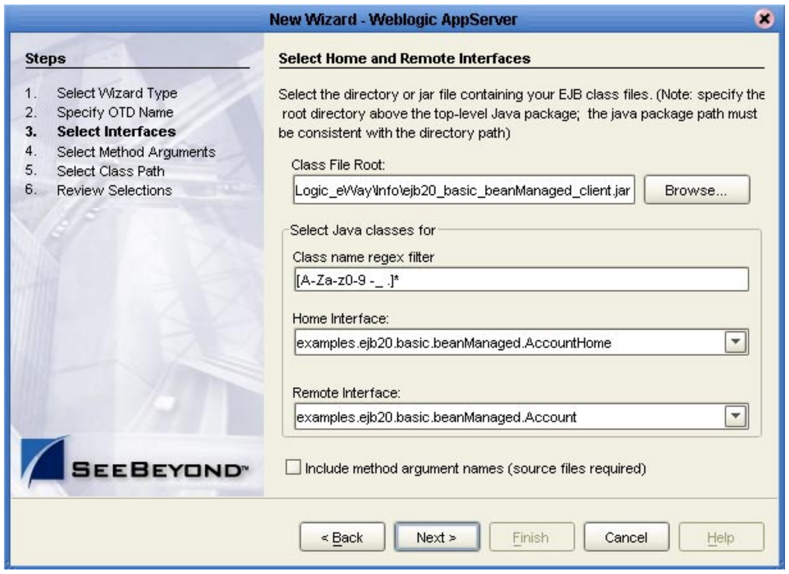
Figure 33 Class File Root Open window



Note: Only recognized Class File Root file names are accepted in the File Name text field.

7 Select the EJB jar file and click the **Open** button. Both the Home Interface and Remote Interface fields are automatically populated in the Select Interfaces window.

Figure 34 Completed Select Interfaces window



New Wizard - Weblogic AppServer

Steps

1. Select Wizard Type
2. Specify OTD Name
3. **Select Interfaces**
4. Select Method Arguments
5. Select Class Path
6. Review Selections

Select Home and Remote Interfaces

Select the directory or jar file containing your EJB class files. (Note: specify the root directory above the top-level Java package; the java package path must be consistent with the directory path)

Class File Root: Logic_eWay\Info\ejb20_basic_beanManaged_client.jar

Select Java classes for

Class name regex filter: [A-Za-z0-9 -_ .]*

Home Interface: examples.ejb20.basic.beanManaged.AccountHome

Remote Interface: examples.ejb20.basic.beanManaged.Account

Include method argument names (source files required)

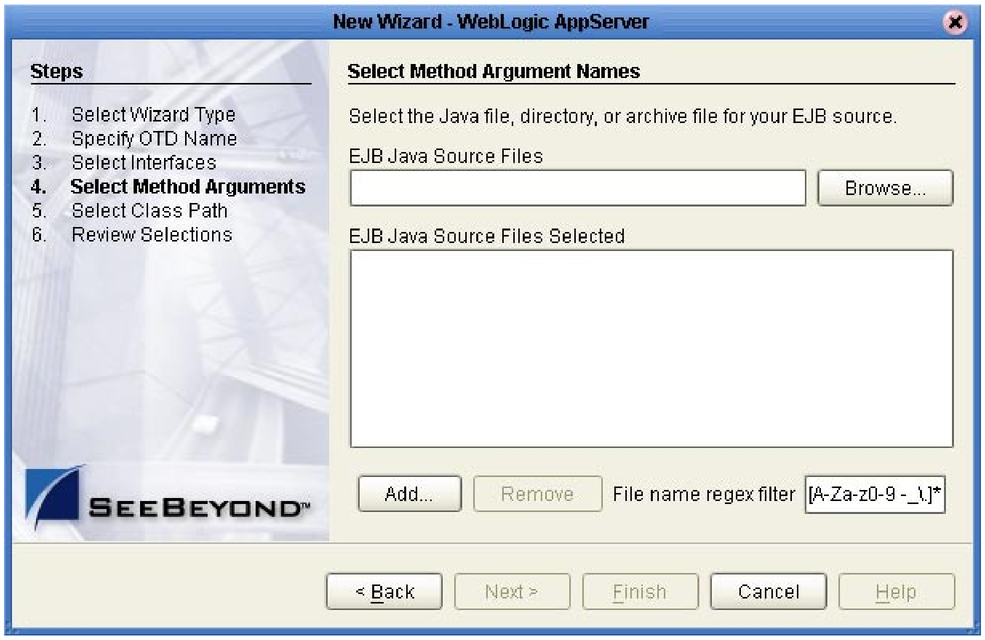
Note: Do not check the Include method argument names checkbox if you do not have the EJB source code.

8 If you selected the **Include Method argument names** checkbox, the **Select Method Arguments** window appears when you click **Next**. If you have not selected the checkbox, skip to **Select Class Path** on page 71

7.2.4. Select Method Argument

9 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 it searches only for .java files.

Figure 35 Select Method Argument



Select the Java file, directory, or archive file for your EJB source.

EJB Java Source Files

EJB Java Source Files Selected

File name regex filter [A-Za-z0-9 -_\.]*

10 Select the source files and click the **Open** button. The files will be populated in the **EJB Java Source Files Selected** window.

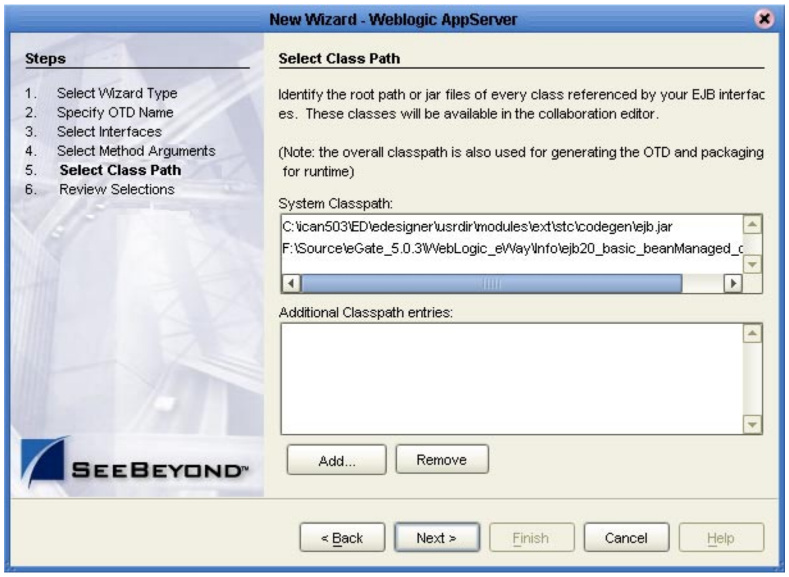
The **Class name regex filter** field is used as a name filter when locating EJB source files. A regular expression can be entered to describe the file name pattern for matching optimization. Wildcard-like (*) patterns, used as suffix, are allowed in this field for matching purposes.

11 Once the desired EJB source files have been added, click **Next**. The **Select Class Path** window appears. At least 2 files must be supplied -- one for the home interface and one for the remote interface (or one EJB bean implementation source).

7.2.5. Select Class Path

The Select Class Path contains a list of system based jar files referenced by the EJB. You must identify the root path and jar files of every class referenced by the EJB interfaces.

Figure 36 Select Class Path window



Select Class Path

Identify the root path or jar files of every class referenced by your EJB interfaces. These classes will be available in the collaboration editor.

(Note: the overall classpath is also used for generating the OTD and packaging for runtime)

System Classpath:

```
C:\ican503\ED\edesigner\usrdir\modules\ext\stc\codegen\ejb.jar
F:\Source\eGate_5.0.3\WebLogic_eWay\Info\ejb20_basic_beanManaged_c
```

Additional Classpath entries:

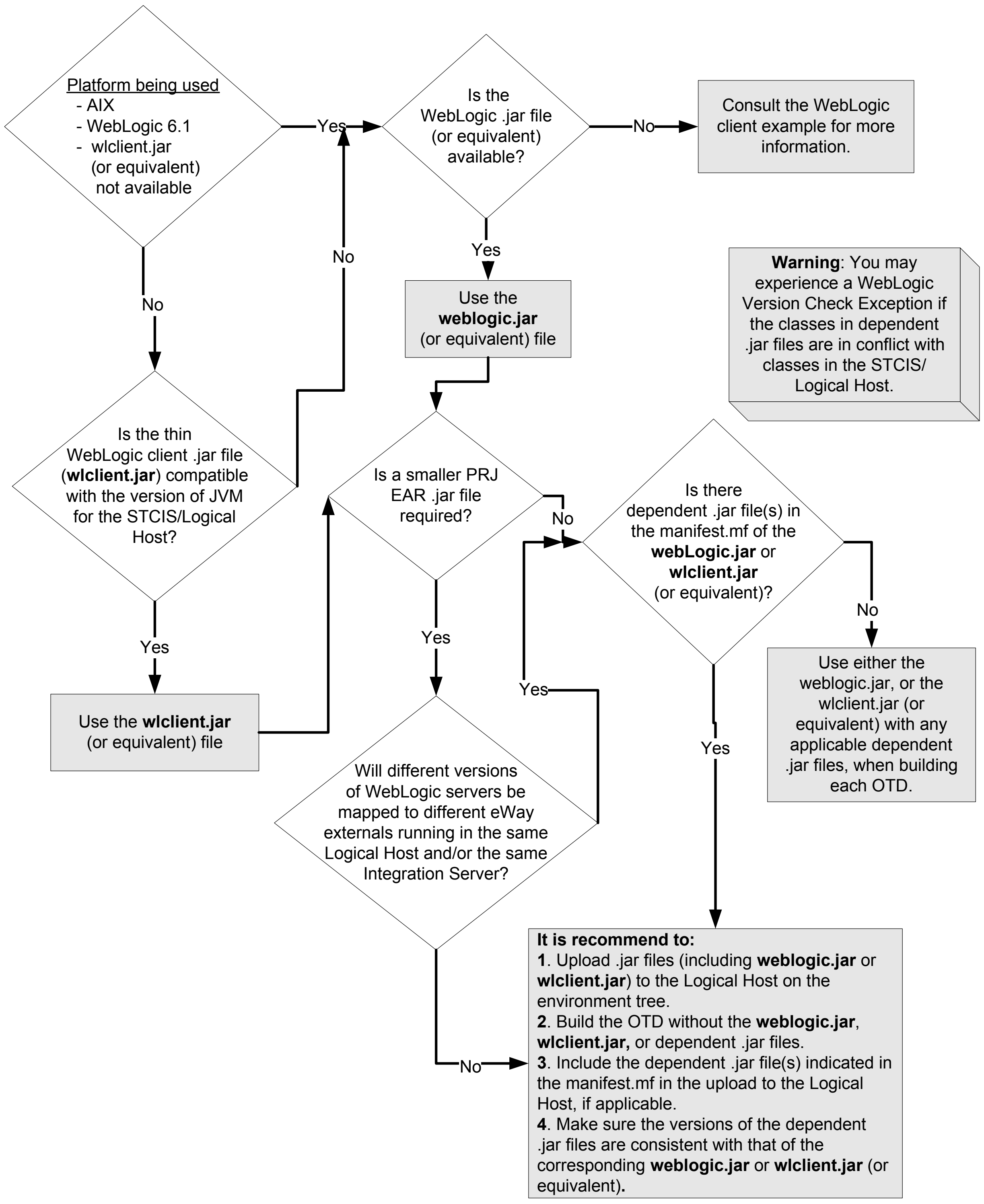
12 Click the **Add** button, and in the Open window, navigate to and select the required jar files. You will be reminded to supply, at the very least, the wlclient.jar or weblogic.jar file which is WebLogic's client jar file.

You may be prompted to provide any dependent archive file(s) if the home and remote interface classes require them for runtime with the logicalhost or for use in any Java Collaboration Editor instance. Please notice that if wlclient.jar is supplied, you must ensure that the version of the JVM, for the underlined logicalhost in use, is compatible with the corresponding wlclient.jar. In most cases, when including the absence of a wlclient.jar (for an older version of the WebLogic server), it is advised to use the self-contained weblogic.jar, even though it will result in larger ICAN project file.

13 You must at least select the **wlclient.jar** file which is WebLogic's client jar file. For more information on indicating the .jar file(s) to be used, refer to **Figure 37 on page 73**

Note: *Use the weblogic.jar file if the wlclient.jar file is not available, or if you are running on an AIX platform.*

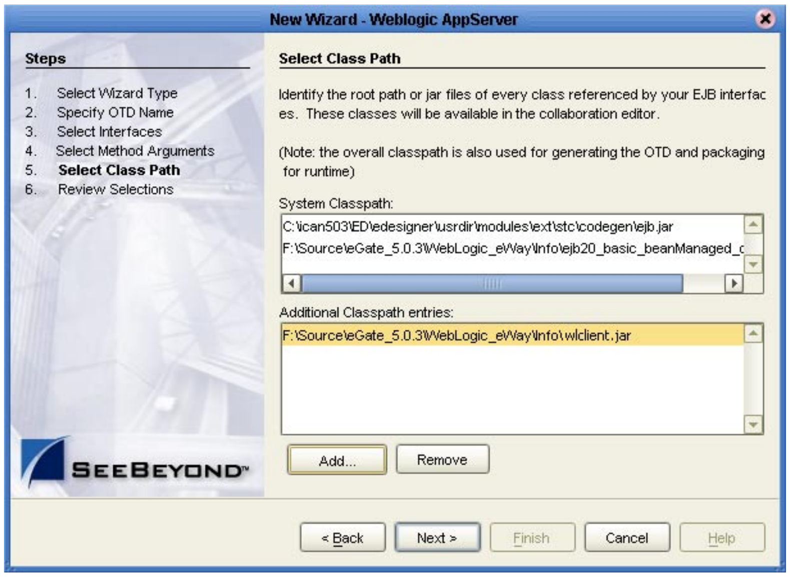
Figure 37 Indicating the .jar file to be used


Platform being used
- AIX
- WebLogic 6.1
- wlclient.jar (or equivalent) not available
Yes
No
Is the WebLogic .jar file (or equivalent) available?
No
Consult the WebLogic client example for more information.
Yes
Use the weblogic.jar (or equivalent) file
Warning: You may experience a WebLogic Version Check Exception if the classes in dependent .jar files are in conflict with classes in the STCIS/ Logical Host.
No
Is the thin WebLogic client .jar file (wlclient.jar) compatible with the version of JVM for the STCIS/Logical Host?
Is a smaller PRJ EAR .jar file required?
No
Is there dependent .jar file(s) in the manifest.mf of the weblogic.jar or wlclient.jar (or equivalent)?
No
Yes
Yes
Yes
Use the wlclient.jar (or equivalent) file
Use either the weblogic.jar, or the wlclient.jar (or equivalent) with any applicable dependent .jar files, when building each OTD.
Yes
Will different versions of WebLogic servers be mapped to different eWay externals running in the same Logical Host and/or the same Integration Server?
No

It is recommend to:
1. Upload .jar files (including **weblogic.jar** or **wlclient.jar**) to the Logical Host on the environment tree.
2. Build the OTD without the **weblogic.jar**, **wlclient.jar,** or dependent .jar files.
3. Include the dependent .jar file(s) indicated in the manifest.mf in the upload to the Logical Host, if applicable.
4. Make sure the versions of the dependent .jar files are consistent with that of the corresponding **weblogic.jar** or **wlclient.jar** (or equivalent)**.**

14 Click the **Open** button to close the window and return to the **Select Class Path** window. SeeFigure 38.

Figure 38 Select Class Path - Add window



Select Class Path

Identify the root path or jar files of every class referenced by your EJB interfac es. These classes will be available in the collaboration editor.

(Note: the overall classpath is also used for generating the OTD and packaging for runtime)

System Classpath:

C:\ican503\ED\edesigner\usrdir\modules\ext\stc\codegen\ejb.jar
F:\Source\eGate_5.0.3\WebLogic_eWay\Info\ejb20_basic_beanManaged_c

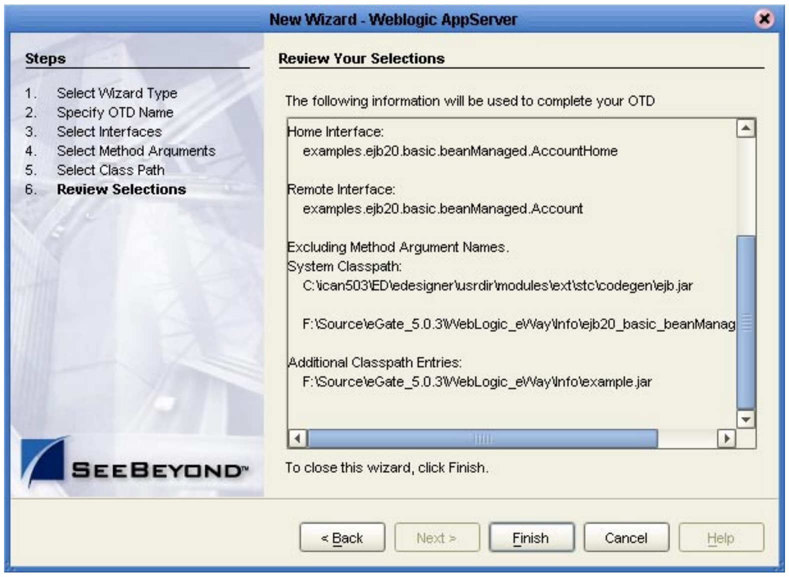
Additional Classpath entries:

F:\Source\eGate_5.0.3\WebLogic_eWay\Info\wlclient.jar

15 Click the **Next** button. The **Review Selections** window appears.

7.2.6. Review Selections

Figure 39 Review Your Selections window



Review Your Selections

The following information will be used to complete your OTD

```
Home Interface:
   examples.ejb20.basic.beanManaged.AccountHome

Remote Interface:
   examples.ejb20.basic.beanManaged.Account

Excluding Method Argument Names.
System Classpath:
   C:\ican503\ED\edesigner\usrdir\modules\ext\stc\codegen\ejb.jar

   F:\Source\eGate_5.0.3\WebLogic_eWay\Info\ejb20_basic_beanManag

Additional Classpath Entries:
   F:\Source\eGate_5.0.3\WebLogic_eWay\Info\example.jar
```

To close this wizard, click Finish.

16 Review your selections and click the **Finish** button to close the window and create the OTD.

Chapter 8

Implementing the WebLogic eWay

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:

- **"Synchronous Communication—eGate to WebLogic Server" on page 75**
- **"Asynchronous Communication—WebLogic EJB to eGate JMS" on page 76**
- **"Asynchronous Communication—eGate JMS to a WebLogic Message Driven Bean" on page 77**

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 WebLogic_BPEL Sample Project" on page 81**
- **"The WebLogic_JCE Sample Project" on page 84**

8.1.2 Asynchronous Communication—WebLogic EJB to eGate JMS

Asychronous Communication from the WebLogic EJB to eGate JMS requires the implementation of SeeBeyond's 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 85**
- **"The JMSTopicSubscribe Sample" on page 86**
- **"The JMSXATopicSubscribe Sample" on page 88**

Preparing WebLogic

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

1. **Configure WebLogic**

   Configure WebLogic to create JNDI entries for SeeBeyond JMS on WebLogic Server instance startup.

2. **Create a new Session Bean**

   Create an EJB that can publish to SeeBeyond 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. **Packaging and Deployment**

   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 SeeBeyond 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 SeeBeyond's JMS IQ Manager into the WebLogic environment.

Sample projects using synchronous communication in this manner include:

- **"The JMSQueueSend Sample" on page 90**
- **"The JMSTopicPublish Sample" on page 91**
- **"The JMSXAQueueSend Sample" on page 92**

Preparing WebLogic

The following steps are required to prepare the WebLogic MDB:

1. **Configure WebLogic**

   Configure WebLogic to create JNDI entries for SeeBeyond 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. **Create a new MDB**

   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. **Packaging and Deployment**

   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 Locating and Importing the Sample Projects

The eWay sample projects are included in the **WebLogiceWay.sar** file. This file is uploaded separately from the WebLogiceWay sar file during installation. For additional information, refer to section called **Sample Projects** on page 13.

Once the **WebLogiceWay.sar** file is uploaded to the Repository, you can begin downloading the sample projects using the **DOCUMENTATION** tab in the Enterprise Manager to a folder of your choosing.

Before using the sample project, you must first import it into the SeeBeyond Enterprise designer using the Enterprise Designer project Import utility.

Note: *eInsight is a Business Process modeling tool. If you have not purchased eInsight, contact your sales representative for information on how to do so.*

To Import a Sample Project:

1. From the Enterprise Designer's Project Explorer pane, right-click the Repository and select **Import**.
2. In the **Import Manager** window, browse to the directory that contains the sample project zip file.
3. Select the sample file and then click **Open**.
4. Click the **Import** button. If the import is successful, then click the **OK** button on the **Import Status** window.
5. Close the Import Manger window.

8.3 Running the Sample Projects

The steps required to run the sample project include:

- Setting the Properties
- Creating the Environment Profile
- Deploying the Project
- Running the Sample

8.3.1 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 File eWays:

1. On the Connectivity Map, double-click the **Inbound File eWay**
2. Select **Inbound File eWay** in the Templates dialog box and click **OK**.
3. The **Properties** window for the Inbound File eWay opens. Modify the parameter settings for your system. Change the Directory and Input file name to match the location and name of the sample data file.
4. Click **OK** to close the **Properties** window.
5. On the Connectivity Map, double-click the Outbound File eWay, select **Outbound File eWay** in the templates dialog box and click **OK**. The **Properties** window for the Inbound File eWay opens.
6. Modify the required parameter settings for your system, including the target Directory and Output file name.
7. Click **OK** to close the **Properties** window.

To Configure the Outbound WebLogic eWay:

1. On the Connectivity Map, double-click the WebLogic eWay.
2. Select **Outbound WebLogic eWay** and click **OK**. The **Properties** window for the WebLogic eWay opens.
3. Modify the parameter settings for your system. Click **OK** to close the **Properties** window.

8.3.2 Creating the Environment Profile

An eGate Environment represents the physical system required to implement a project. A typical Environment contains several components, including Logical Hosts, Integration Servers, Message Servers, and External Systems. Environments are created using the Enterprise Designer’s Environment Explorer

To Create a New Environment:

1. On the Environment Explorer, highlight and right-click the eWay profile.
2. Select Properties, and enter the configuration information required for the eWay. for more information, see **Configuring the Environment Properties** on page 17.

8.3.3 Deploying the Project

For instruction on the steps required to deploy a project, see the *eGate Integrator User's Guide*.

8.3.4 Running the Sample

For instruction on the steps required to run the Sample project, see the *eGate Integrator Tutorial*.

8.4 Using Sample Projects in eInsight

This section describes how to use sample projects with the ICAN Suite's 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*.

Before running a sample project, you must:

- Import the sample project
- Create an Environment for the sample project
- Configure the eWay properties for your specific system (see **Configuring the WebLogic eWay Properties** on page 15)
- Create a Deployment Profile

Note: *While several key steps are required to create, activate, and deploy a project, only the steps containing information relevant to the sample project in eInsight are included in this chapter. For more detailed information on how to complete a sample project, see the eInsight Enterprise Service Bus Installation Guide.*

8.4.1 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.

8.4.2 The WebLogic_BPEL Sample Project

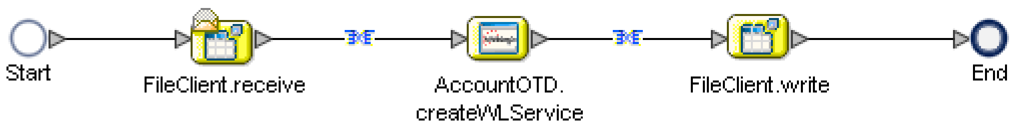
The eInsight sample project **WebLogic_BPEL** demonstrates a synchronous interaction between the WebLogic EJB and the eGate JMS. As mentioned previously, synchronous communication is considered an unbuffered process, requiring complete data transmission and reply or confirmation of message transmission 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 WebLogic_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:

CreateAccount_BP

This business process describes the account creation process seen in Figure 40.

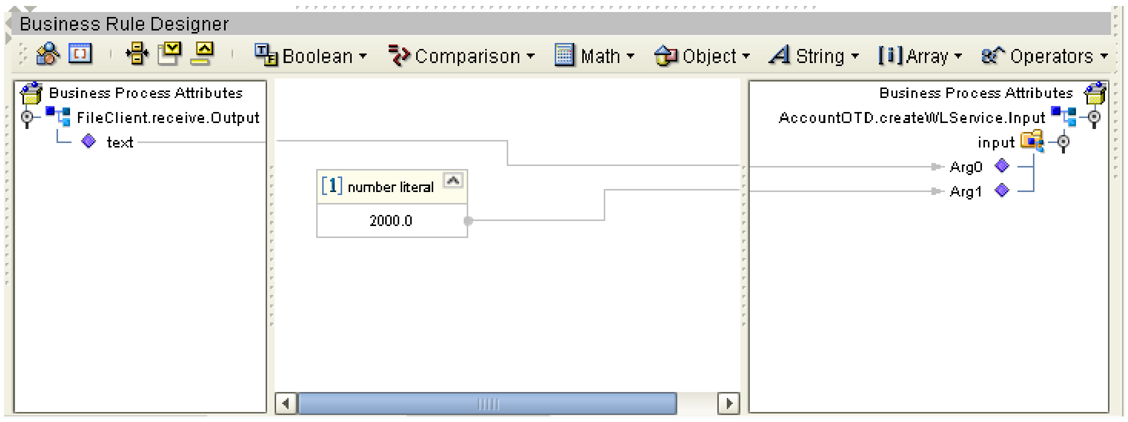
Figure 40 Account Creation Business Process



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 41) before sending the combined data to the FileClient.write container.

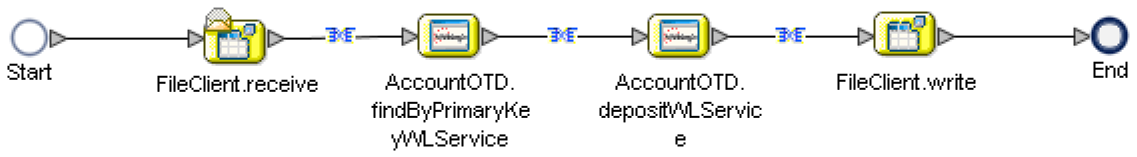
Figure 41 Account Creation


Business Rule Designer
Boolean
Comparison
Math
Object
String
Array
Operators
Business Process Attributes
FileClient.receive.Output
text
number literal
2000.0
Business Process Attributes
AccountOTD.createWLService.Input
input
Arg0
Arg1

DepositAmound_BP

This business process describes the account deposit process as seen in Figure 42.

Figure 42 Account Deposit Business Process

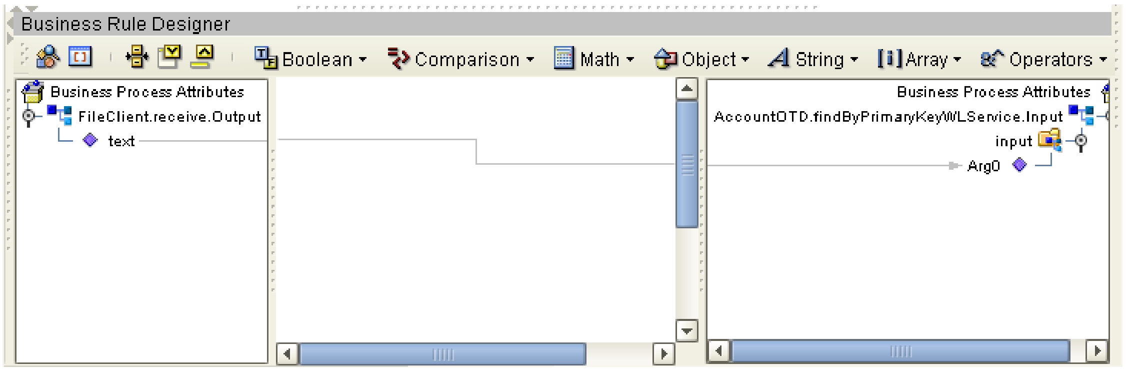


Business Process:

1 The file eWay subscribes to an external directory and picks up a text request for specific account data.

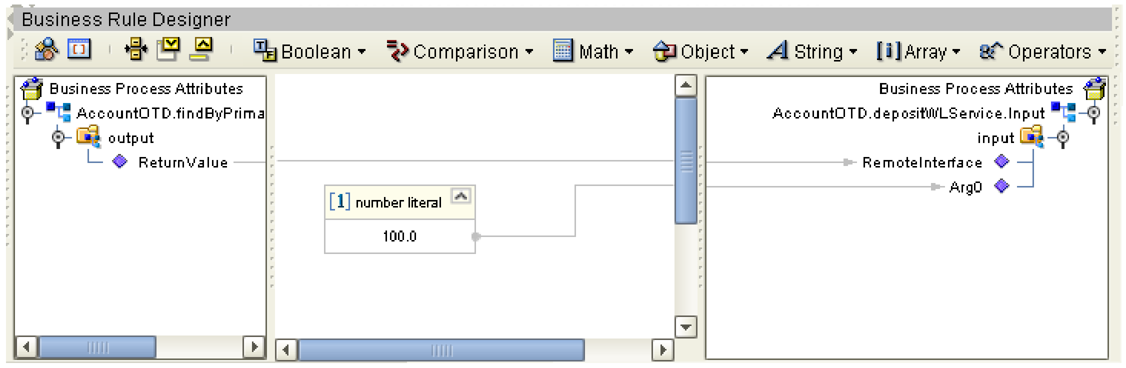
2 Account data is copied to the input container FileClient.receive.Output, using the AccountOTD.findByPrimaryKeyWLService BPEL service. This service also requests the specific data (Account ID and amount) using the primary key from WebLogic Server, as seen in Figure 43.

Figure 43 Retrieving Account Data


Business Rule Designer
Boolean
Comparison
Math
Object
String
Array
Operators
Business Process Attributes
FileClient.receive.Output
text
Business Process Attributes
AccountOTD.findByPrimaryKeyWLService.Input
input
Arg0

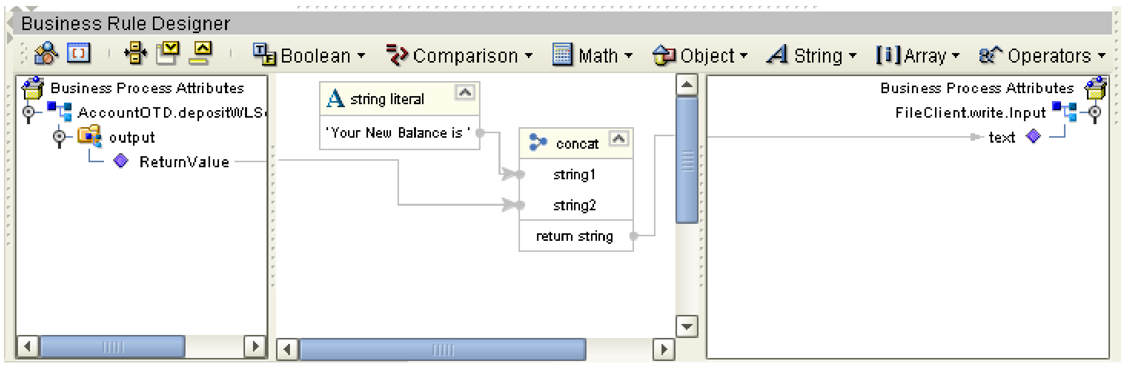
3 The AccountOTD.depositWLService OTD is then invoked, adding the number literal of 100 to the account balance listed in WebLogic Server (see Figure 44).

Figure 44 Making the Deposit



4 Combined data is sent to the FileClient.write container, as seen in Figure 45.

Figure 45 Returning the Response



8.5 Using the Sample Projects in eGate

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

Before running a sample project, you must:

- Import the sample project
- Create an environment
- Configure the eWay properties for your specific system (see **Configuring the WebLogic eWay Properties** on page 15)
- Create a Deployment Profile

Note: *While several key steps are required to create, activate, and deploy a project, only the steps containing information relevant to the WebLogic eWay are included in this chapter. For more detailed information on how to complete a sample project, see the eGate Integrator Tutorial.*

8.5.1 The WebLogic_JCE Sample Project

The **WebLogic_JCE** sample project demonstrates synchronous communication from eGate to WebLogic. The sample Project is identical to the **WebLogic_BPEL** sample project described in section **8.4.2**, with the exception that there is no eInsight Business Process.

The Connectivity Maps for this sample appear as follows:

Figure 46 WebLogic_JCE Connectivity Map—Create Account

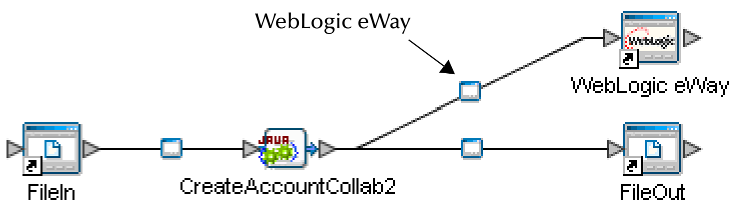
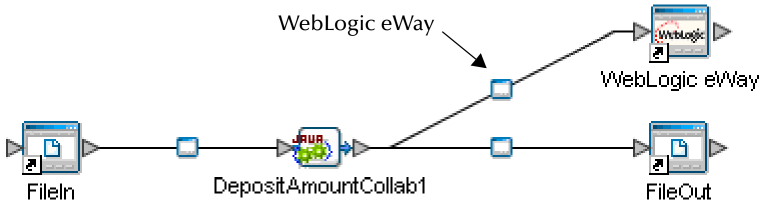


Figure 47 WebLogic_JCE Connectivity Map—Deposit



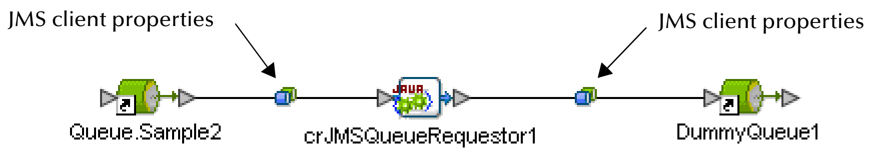
This sample project demonstrates creating and then depositing funds to an account using two collaborations (CreateAccountCollab and DepositAmountCollab). The first collaboration subscribes to the FileIn (File eWay), and then picks up an account name (String). Both the account name and a string literal 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.

8.5.2 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.

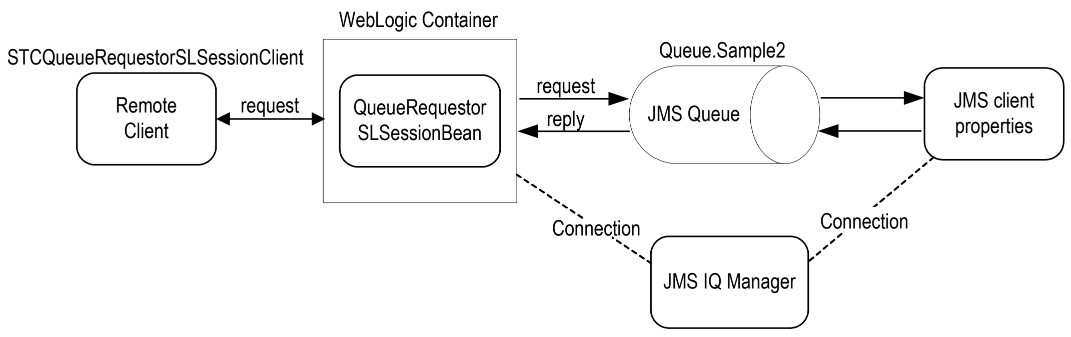
The Connectivity Map for this sample project appears as follows:

Figure 48 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 SeeBeyond 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 49 JMSQueueRequestor Sample Components



As seen in Figure 49, 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 (SeeBeyond.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 SeeBeyond.STCQueueRequestorSLSessionBean "This is a text message." msg
```

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

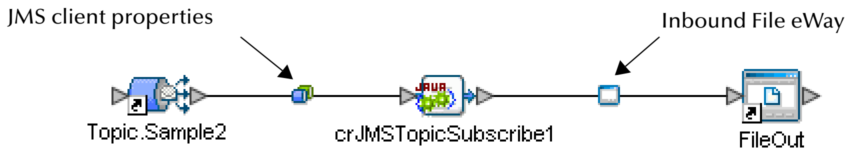
```
java com.stc.eways.ejb.sessionbean.publisher.STCPublisherSLSessionClient t3://localhost:7001
SeeBeyond.STCPublisherSLSessionBean c:\temp\testfile.txt file
```

8.5.3 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.

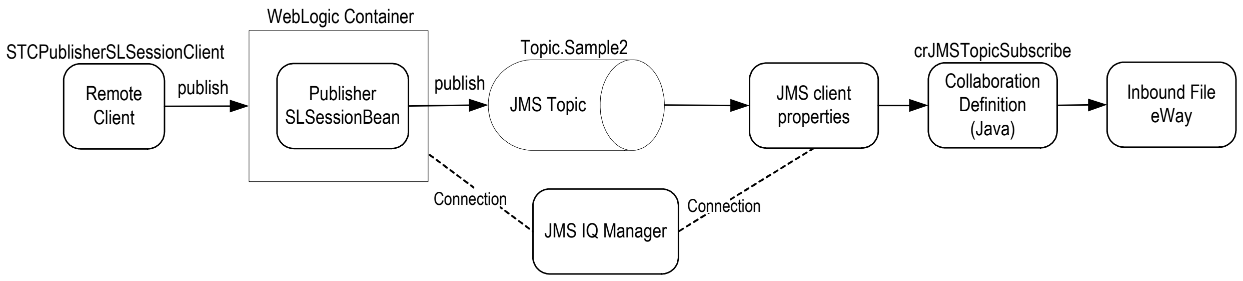
The Connectivity Map for this sample project appears as follows:

Figure 50 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 SeeBeyond JMS Topic. The Collaboration (crJMSTopicSubscribe) seen on the Connectivity Map is configured to use the internal SeeBeyond 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 51 JMSTopicSubscribe Sample Components



As seen in Figure 51, 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 (SeeBeyond.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
SeeBeyond.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
SeeBeyond.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 stcejbweblogic.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.

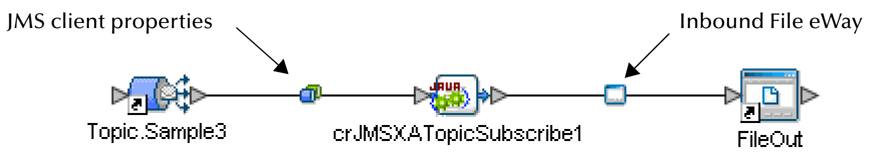
Note: *For more information on eWay Connection Configuration Parameters for JMS see* **Configuring the WebLogic eWay Properties** *on page 15*

8.5.4. 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:

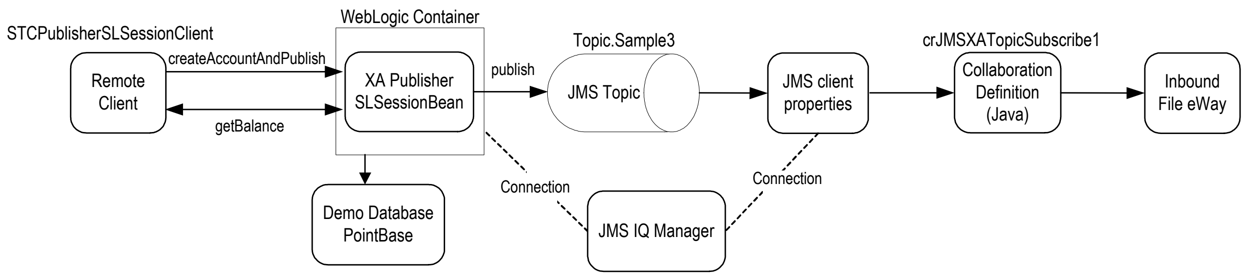
Figure 52 JMSXATopicSubscribe Sample



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 SeeBeyond 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 SeeBeyond JMS Topic.

Figure 53 JMSXATopicSubscribe Sample Components



The stand-alone remote client, com.stc.eways.ejb.sessionbean.xapublisher.STCPublisherSLSessionClient, 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 (SeeBeyond.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
SeeBeyond.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 stcejbweblogic.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 SeeBeyond JMS XA Resources are being used, look at the weblogic.log and SeeBeyond 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 112.

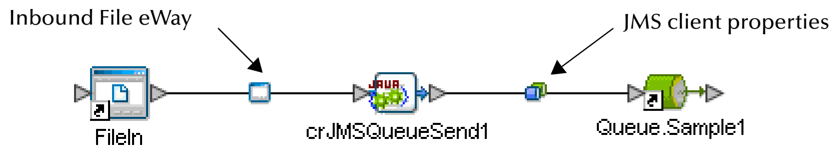
Important: *XA transactions for the WebLogic eWay are managed by the WebLogic TransactionManager, NOT the eGate TransactionManager or in the eWay Connection parameters. For XA transactions make sure that the XAConnectionFactory(ies) are configured for the startup class.*

8.5.5 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.

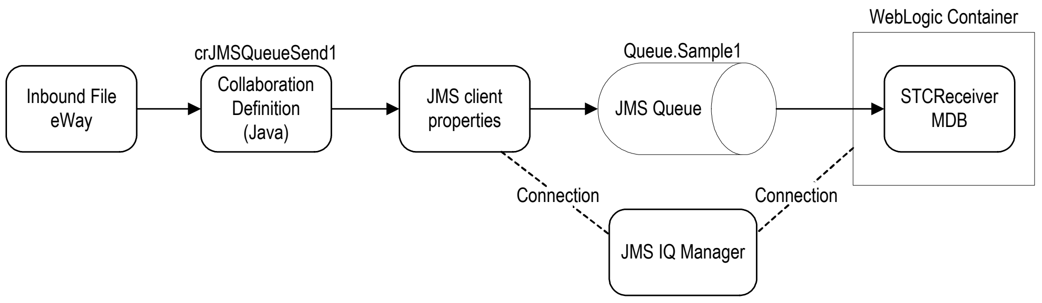
The Connectivity Map for this sample project appears as follows:

Figure 54 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 SeeBeyond 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 55 JMSQueueSend Sample Components



As seen in Figure 55, 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 SeeBeyond JMS IQ Manager to drive the MDB, see* **SeeBeyond JMS** *on page 38.*

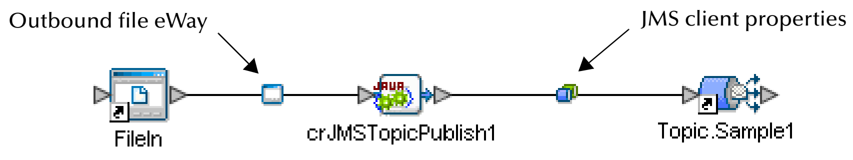
For more information on configuring Environment Properties for the sample project, see **Configuring the Environment Properties** *on page 17*

8.5.6. 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.

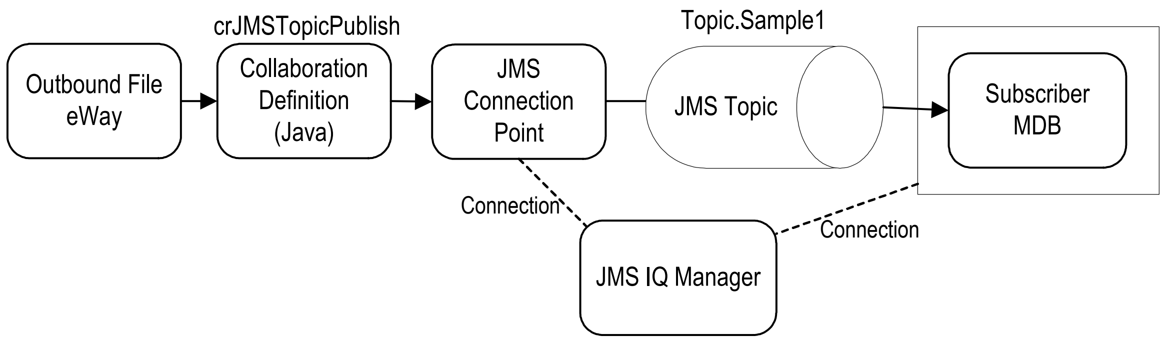
The Connectivity Map for this sample appears as follows:

Figure 56 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 SeeBeyond 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 57 JMSTopicPublish Sample Components



As seen in Figure 57, 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).

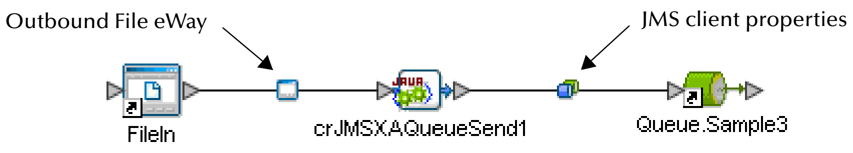
Note: *For more information on how to configure/deploy the MDB to use the SeeBeyond JMS IQ Manager to drive the MDB, see* **SeeBeyond JMS** *on page 38.*

8.5.7. 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.

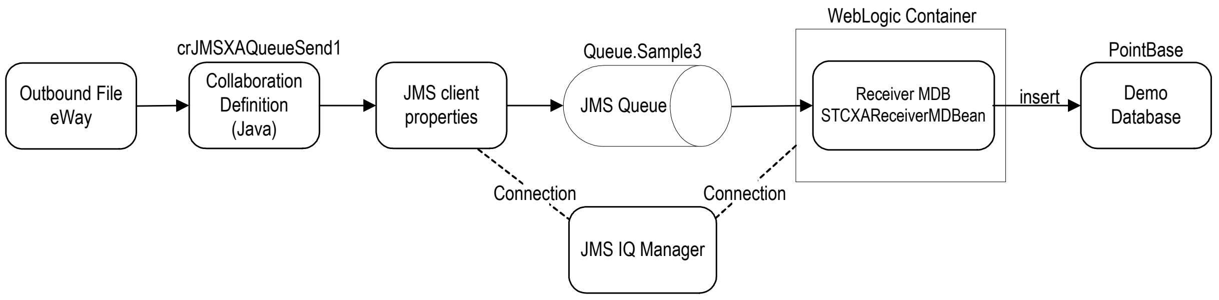
The Connectivity Map for tis sample project appears as follows:

Figure 58 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 SeeBeyond JMS IQ Manager as the JMS server, and is responsible for copying data from the source event to the output event.

Figure 59 JMSXAQueueSend Sample Components



As seen in Figure 59, 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 SeeBeyond JMS IQ Manager to drive the MDB, see **SeeBeyond JMS** on page 38.)

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 SeeBeyond JMS XAResource and the PointBase sample demoXAPool to receive messages from SeeBeyond 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 SeeBeyond 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 110. XA prepares and commits should be called on both database and SeeBeyond 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 112. For details on the format of the input message for the feeder eWay see **SeeBeyond Sample Message Driven Beans** on page 94.

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 stcejbweblogic.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 stcejbweblogic.jar file and the associated deployment descriptor files.*

Chapter 9

SeeBeyond Sample Message Driven Beans

The previous sections, **Java Naming and Directory Interface (JNDI)** on page 26 and **Java Messaging Service (JMS)** on page 29 describe the JNDI and JMS subsystems. This chapter relates the concepts that were discussed in the previous sections with those regarding the SeeBeyond Message Driven Beans (MDBs).

There are two MDBs that are deployed in WebLogic:

- MDB Subscribing to SeeBeyond Topic
- MDB Subscribing to SeeBeyond 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.

9.1 MDB Subscribing to a SeeBeyond Topic

This MDB subscribes to a SeeBeyond JMS Topic. It receives from ONLY ONE SeeBeyond 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>Container</transaction-type>
                  <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>STCSubscriberMDBean</ejb-name>
                        <method-name>*</method-name>
                  </method>
                  <trans-attribute>NotSupported</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 SeeBeyond 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**.

<container-transaction> Tag

In the <container-transaction> tag of the <assembly-descriptor>, we define the transactional mode for the MDB. Because this MDB does not use a transaction, the **NotRequired** tag in <trans-attribute> is specified.

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>SeeBeyond.Topics.STCTopic1</destination-jndi-name>
                  <initial-context-factory>weblogic.jndi.WLInitialContextFactory</initial-context-
factory>
                  <provider-url>t3://localhost:7003</provider-url>
                  <connection-factory-jndi-
name>SeeBeyond.TopicConnectionFactories.TopicConnectionFactory</connection-factory-jndi-name>
            </message-driven-descriptor>
            <jndi-name>SeeBeyond.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 SeeBeyond 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 **SeeBeyond WebLogic Startup Class** on page 47 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 SeeBeyond Queue

This MDB subscribes to only one SeeBeyond 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>Container</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>STCReceiverMDBean</ejb-name>
                        <method-name>*</method-name>
                  </method>
                  <trans-attribute>NotSupported</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 SeeBeyond 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**.

<container-transaction> Tag

In the <container-transaction> tag of the <assembly-descriptor>, the transactional mode is defined for the MDB. Because this MDB does not use a transaction, the **NotRequired** tag in <trans-attribute> is specified.

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>SeeBeyond.Queues.STCQueue1</destination-jndi-name>
                  <initial-context-factory>weblogic.jndi.WLInitialContextFactory</initial-context-
factory>
                  <provider-url>t3://localhost:7003</provider-url>
                  <connection-factory-jndi-
name>SeeBeyond.QueueConnectionFactories.QueueConnectionFactory</connection-factory-jndi-name>
            </message-driven-descriptor>
            <jndi-name>SeeBeyond.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 SeeBeyond 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 **SeeBeyond WebLogic Startup Class** on page 47 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. SeeBeyond Sample Session Beans

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

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

In the following sections, there are references to two XML files: **jb-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 SeeBeyond Topic

This Stateless Session Bean publishes to a SeeBeyond 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 SeeBeyond 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.STCPublisherSLSessionHome</home>
                  <remote>com.stc.eways.ejb.sessionbean.publisher.STCPublisherSLSession</remote>
                  <ejb-class>com.stc.eways.ejb.sessionbean.publisher.STCPublisherSLSessionBean</
ejb-class>
                  <session-type>Stateless</session-type>
                  <transaction-type>Container</transaction-type>
                  <resource-ref>
                        <res-ref-name>jms/TopicConnectionFactory</res-ref-name>
                        <res-type>javax.jms.TopicConnectionFactory</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>
            …

</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 **jsm/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.

ejb-jar.xml

In addition to the **ejb-jar.xml** file, the Session Bean also needs to be included in the **weblogic-ejb-jar.xm**l 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>SeeBeyond.TopicConnectionFactories.TopicConnectionFactory</
jndi-name>
                  </resource-description>
                  <resource-env-description>
                        <res-env-ref-name>jms/Topic</res-env-ref-name>
                        <jndi-name>SeeBeyond.Topics.STCTopic2</jndi-name>
                  </resource-env-description>
            </reference-descriptor>
            <jndi-name>SeeBeyond.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.xm**l.

<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:

SeeBeyond.TopicConnectionFactories.TopicConnectionFactory

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

SeeBeyond.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 **SeeBeyond WebLogic Startup Class** on page 47 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 SeeBeyond Queue

This Stateless Session Bean sends a request to the SeeBeyond 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 SeeBeyond 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.STCQueueRequestorSLSessionHome</home>

<remote>com.stc.eways.ejb.sessionbean.queuerequestor.STCQueueRequestorSLSession</remote>
                  <ejb-
class>com.stc.eways.ejb.sessionbean.queuerequestor.STCQueueRequestorSLSessionBean</ejb-class>
                  <session-type>Stateless</session-type>
                  <transaction-type>Container</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>Container</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 **jsm/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>SeeBeyond.QueueConnectionFactories.QueueConnectionFactory</
jndi-name>
                  </resource-description>
                  <resource-env-description>
                        <res-env-ref-name>jms/Queue</res-env-ref-name>
                        <jndi-name>SeeBeyond.Queues.STCQueue2</jndi-name>
                  </resource-env-description>
            </reference-descriptor>
            <jndi-name>SeeBeyond.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:

SeeBeyond.QueueConnectionFactories.QueueConnectionFactory

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

SeeBeyond.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 **SeeBeyond WebLogic Startup Class** on page 47 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 SeeBeyond 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 SeeBeyond JMS Topic.
     *
     * @param       message             The text message to send to a JMS Topic.
     *
     * @throws      EJBException        Upon error.
     *
     * @author      SeeBeyond
     */
    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 SeeBeyond Sample XA Message Driven Beans

A MDB can subscribe to a SeeBeyond 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 SeeBeyond JMS Queue Transactionally

The MDB subscribes to a (single) SeeBeyond 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 SeeBeyond 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 112 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 SeeBeyond 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>
            <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>SeeBeyond.Queues.STCQueue3</destination-jndi-name>
                  <initial-context-factory>weblogic.jndi.WLInitialContextFactory</initial-
context-factory>
                  <provider-url>t3://localhost:7003</provider-url>
                  <connection-factory-jndi-
name>SeeBeyond.QueueConnectionFactories.XAQueueConnectionFactory</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>SeeBeyond.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 SeeBeyond 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 **SeeBeyond WebLogic Startup Class** on page 47 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. SeeBeyond Sample XA Session Beans

A Session Bean (Stateless or Stateful) can publish a message to a SeeBeyond JMS Topic or send a message to a SeeBeyond JMS Queue in an XA transaction. The Session Bean accesses the SeeBeyond **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 SeeBeyond JMS XA Resource to WebLogic TransactionManager. The enlistment must be done to the current XA transaction created by the WebLogic container.

How To Enlist SeeBeyond 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 SeeBeyond 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 SeeBeyond Startup Class (see SeeBeyond 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 SeeBeyond JMS Topic Transactionally

This Stateless Session Bean publishes to a SeeBeyond 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 SeeBeyond JMS XAResource and the demo XA database resource pool.(See **examples-dataSource-demoXAPool** on page 112 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 SeeBeyond 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 SeeBeyond 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 SeeBeyond 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.STCXAPublisherSLSessionHome</home>
                  <remote>com.stc.eways.ejb.sessionbean.xapublisher.STCXAPublisherSLSession</
remote>
                  <ejb-
class>com.stc.eways.ejb.sessionbean.xapublisher.STCXAPublisherSLSessionBean</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 **jsm/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>SeeBeyond.TopicConnectionFactories.XATopicConnectionFactory</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>SeeBeyond.Topics.STCTopic3</jndi-name>
                  </resource-env-description>
            </reference-descriptor>
            <jndi-name>SeeBeyond.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:

SeeBeyond.TopicConnectionFactories.XATopicConnectionFactory

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

SeeBeyond.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 **SeeBeyond WebLogic Startup Class** on page 47 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 SeeBeyond 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 6.1,** modify the server startup script (i.e., startExamplesServer.cmd) to include the following Java properties in the command line:

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

For **WebLogic 7.0**, 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.seebeyond.jms.client.STCXAResource,
xar=com.seebeyond.jms.client.STCXAResource@82e1a, flags=TMNOFLAGS)>
####<Apr 4, 20xx 5:49:52 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <> <>
<000000> <ResourceDescriptor[com.seebeyond.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.seebeyond.jms.client.STCXAResource, xar=com.seebeyond.jms.client.STCXAResource@82e1a>
####<Apr 4, 20xx 5:49:52 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <ResourceDescriptor[com.seebeyond.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.seebeyond.jms.client.STCXAResource,
xar=com.seebeyond.jms.client.STCXAResource@82e1a, flags=TMSUCCESS)>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <> <>
<000000> <ResourceDescriptor[com.seebeyond.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.seebeyond.jms.client.STCXAResource, xar=com.seebeyond.jms.client.STCXAResource@82e1a>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <ResourceDescriptor[com.seebeyond.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.seebeyond.jms.client.STCXAResource,
xar=com.seebeyond.jms.client.STCXAResource@82e1a>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <> <>
<000000> <ResourceDescriptor[com.seebeyond.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.seebeyond.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.seebeyond.jms.client.STCXAResource].commit(xid=5:fa80c71858e3d95b,onePhase=false)>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <> <>
<000000> <ResourceDescriptor[com.seebeyond.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.seebeyond.jms.client.STCXAResource, xar=com.seebeyond.jms.client.STCXAResource@82e1a>
####<Apr 4, 20xx 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <ResourceDescriptor[com.seebeyond.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

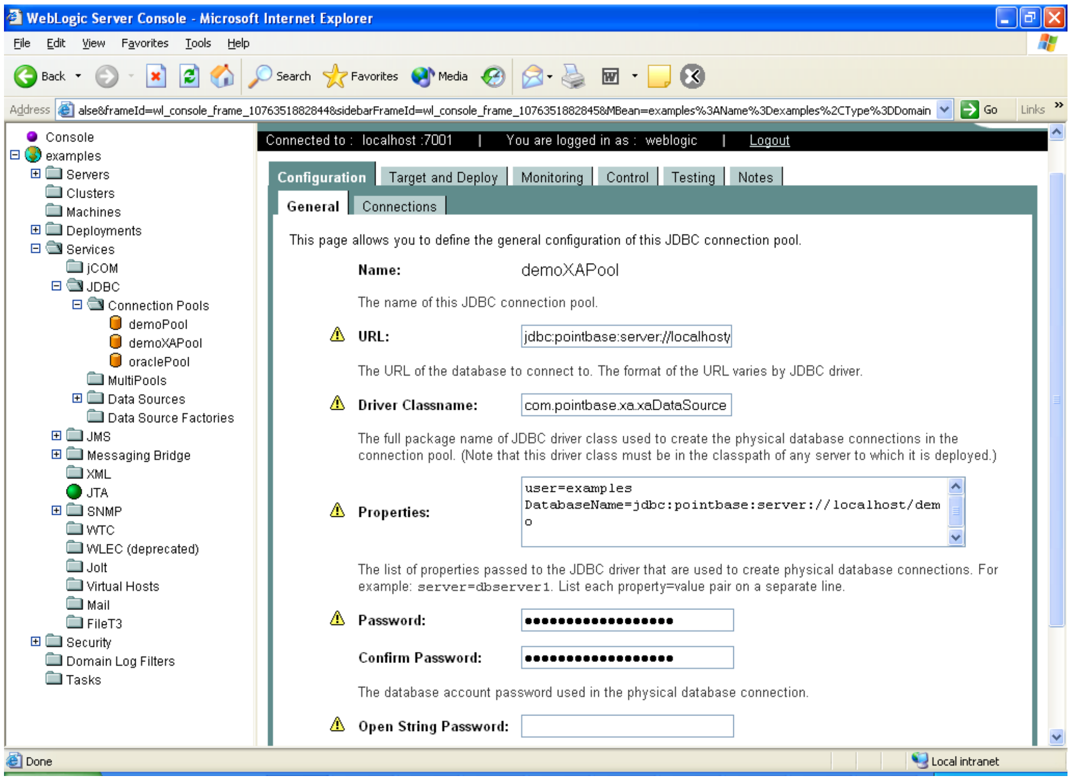
Additional logging and monitoring of JTA and JMS XA can be configured for WebLogic Server 7.0 through the Administrator Console. From the navigation pane on the left, expand the Servers node and select the appropriate server. Configure monitoring and logging in the following locations:

- 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 60 on page 112**) and associates it with the pre-installed connection pool named demoXAPool (see **on page 113**). 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 60 WebLogic (8.1) Administrative Console - demoXAPool



Connected to : localhost :7001 | You are logged in as : weblogic | Logout

This page allows you to define the general configuration of this JDBC connection pool.

Name: demoXAPool

The name of this JDBC connection pool.

URL: jdbc:pointbase:server://localhost/

The URL of the database to connect to. The format of the URL varies by JDBC driver.

Driver Classname: com.pointbase.xa.xaDataSource

The full package name of JDBC driver class used to create the physical database connections in the connection pool. (Note that this driver class must be in the classpath of any server to which it is deployed.)

Properties:
```
user=examples
DatabaseName=jdbc:pointbase:server://localhost/dem
o
```

The list of properties passed to the JDBC driver that are used to create physical database connections. For example: `server=dbserver1`. List each property=value pair on a separate line.

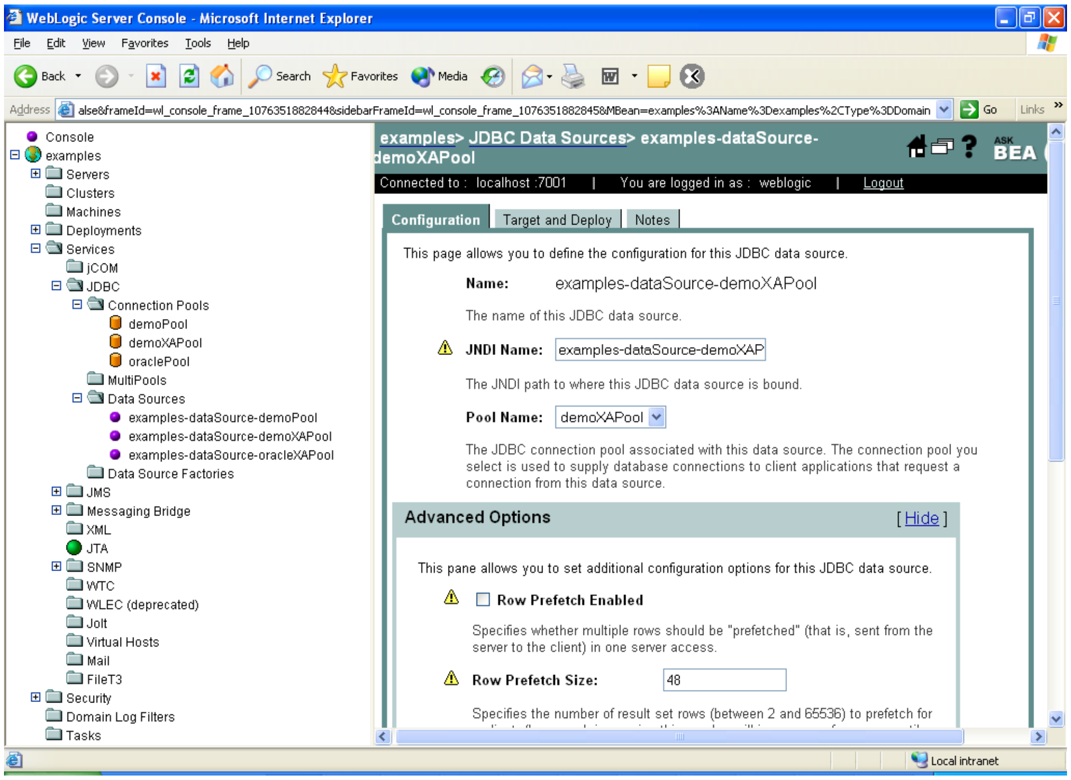
Password: ••••••••••••••••••

Confirm Password: ••••••••••••••••••

The database account password used in the physical database connection.

Open String Password:

WebLogic (8.1) Administrative Console - demoXAPool



This page allows you to define the configuration for this JDBC data source.

Name: examples-dataSource-demoXAPool

The name of this JDBC data source.

JNDI Name: examples-dataSource-demoXAP

The JNDI path to where this JDBC data source is bound.

Pool Name: demoXAPool

The JDBC connection pool associated with this data source. The connection pool you select is used to supply database connections to client applications that request a connection from this data source.

This pane allows you to set additional configuration options for this JDBC data source.

☐ **Row Prefetch Enabled**

Specifies whether multiple rows should be "prefetched" (that is, sent from the server to the client) in one server access.

Row Prefetch Size: 48

Specifies the number of result set rows (between 2 and 65536) to prefetch for

Index