SeeBeyond ICAN Suite

e*Way Intelligent Adapter for WebLogic User's Guide

Release 5.0.5 for Schema Run-time Environment (SRE)

SEEBEYOND®

Contents

Chapter 5

Java Methods   116

Introduction

This document describes the integration between BEA WebLogic™ application Server and SeeBeyond e*Gate using the e*Way Intelligent Adapter for WebLogic (the WebLogic e*Way).

1.1 Intended Reader

The reader of this guide is presumed:

- to be a developer or system administrator with the responsibility of maintaining the e*Gate system.
- to have high-level knowledge of Windows or UNIX operations and administration.
- to be familiar with WebLogic Server functions.
- to have high-level knowledge of Java™, JMS™, and Enterprise JavaBeans™.
- to be thoroughly familiar with Windows-style GUI operations.

1.2 Overview

WebLogic Server

BEA's WebLogic Server is an application server used to build new applications with graphical interfaces or screens. These may be accounting applications, HR applications, shipping applications, and so forth.

The WebLogic application server is an architecture for building business logic in re-usable components so that a Web server can access this data easily. The application server talks (in the Java world) in terms of Enterprise JavaBeans. Enterprise JavaBeans (EJBs) are the units of work that an application server is responsible for and exposes to the external world. The interface between the presentation and real applications/real data is the EJB.

The WebLogic application server allows the user to build EJBs and deploy them, making them available to other applications on various machines. These EJBs are Java programs written by the developer and deployed to the application server. The application server offers services that users previously had to write themselves

including connectivity, business logic, re-usability, security, concurrency (access is serialized), and transactionallity (uses XA to assure a successful transfer/update or rollback).

The WebLogic application server performs pooling to conserve system resources. Object pooling reduces the number of allocations by placing objects in a pool so that the next request for the object does not require a re-allocation of memory. Thread pooling and connection pooling work in much the same way to save memory and connection resources. Clustering is another benefit of the EJB's. Clustering means that the applications are easily moved or distributed to other machines. The WebLogic application server streamlines the process of building distributed, scalable, highly available systems.

The Intelligent Adapter for WebLogic e*Way

The Intelligent Adapter for WebLogic e*Way (WebLogic e*Way) facilitates integration between applications built on the WebLogic platform and e*Gate, using J2EE's component model (EJB).

1.3 Using J2EE™ with e*Gate and the WebLogic e*Way

The e*Way Intelligent Adapter for WebLogic employs Java 2 Platform, Enterprise Edition™ (J2EE™) components and services. The following sections break down the JNDI™, JMS and EJB subsystems, and XA Transactions, with respect to the WebLogic integration strategy (as described in **WebLogic e*Way Component Overview** on page 14).

1.3.1. Java Naming and Directory Interface (JNDI)

Java Naming and Directory Interface™ (JNDI) is an API published by Sun. In short, this set of APIs allows a Java program to store objects and lookup objects using multiple naming services in a standard manner. A naming service may be LDAP, a file system, or an 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 in 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 T3 Naming Service

Any J2EE compliant Application Server, such as WebLogic, 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 Event Type Definition (ETD) to find the home interface of an EJB, JNDI must be

configured in the connection .def file and associated with the ETD. 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

Here's an example of code for 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 1), log onto the Administrative console for the server you want to administer (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 1   Administrative Console - View JNDI Tree
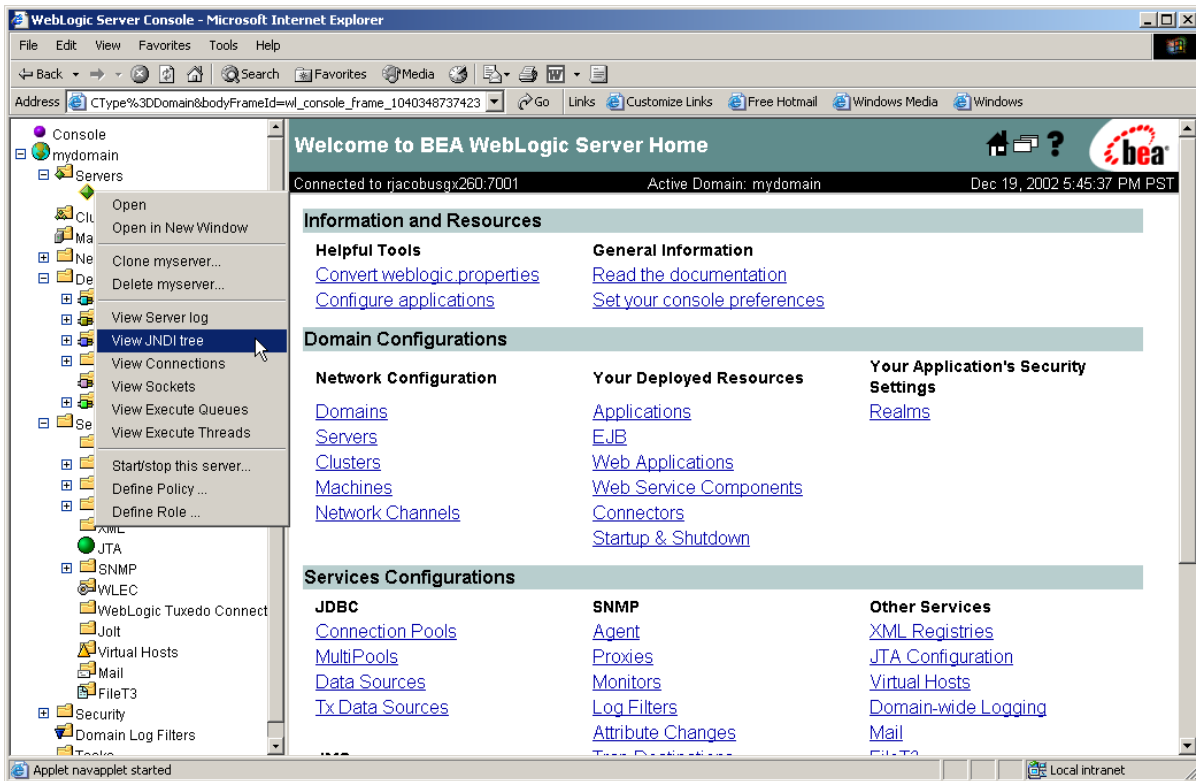
In the following example, (see **Figure 2 on page 11**) the JNDI tree Web page shows that the **SeeBeyond** subcontext was expanded in order to view the SeeBeyond JMS objects that were bound to the WebLogic JNDI. These objects are bound when the **STCWLStartup** class is loaded and run by the WebLogic Server. (See **SeeBeyond WebLogic Startup Class** on page 24 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 ETD to look up the home interface of the EJB.

Figure 2   Administrative Console - The JNDI Tree Web Page
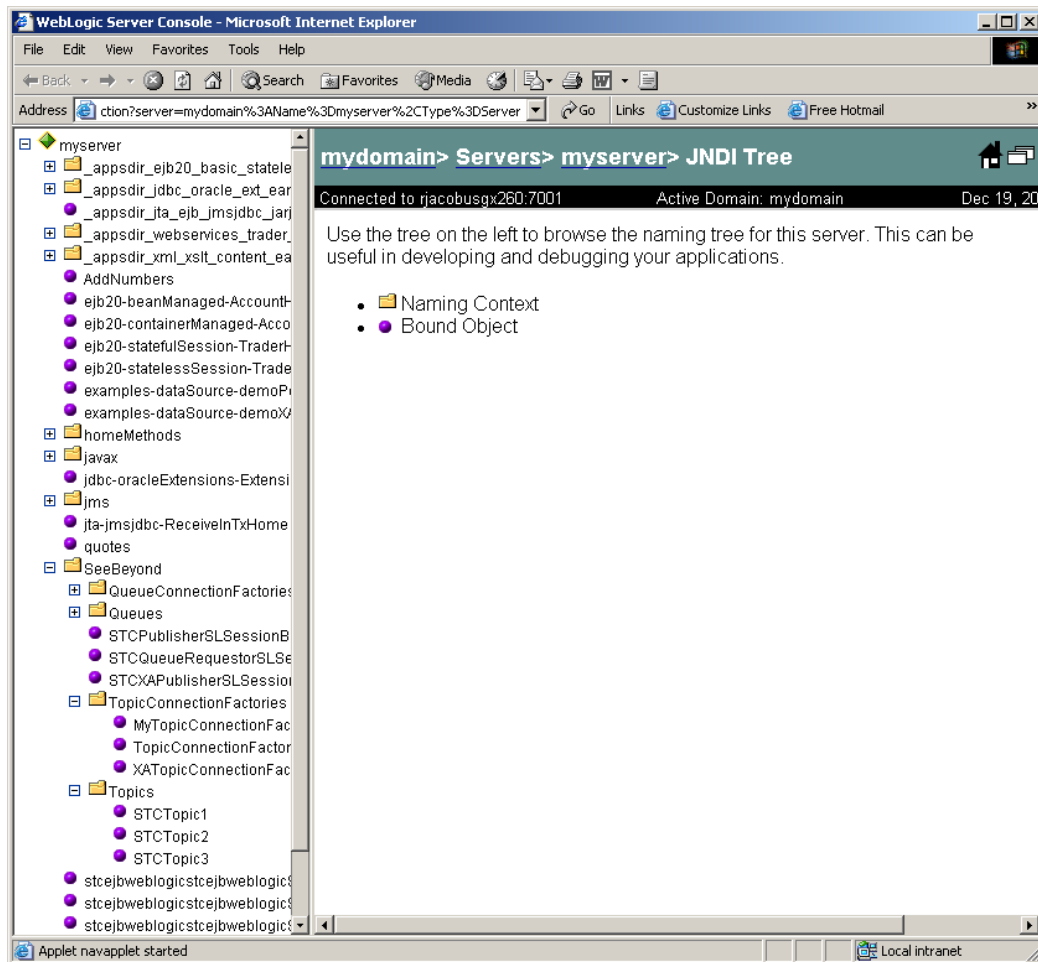
1.3.2. Java Messaging Service (JMS)

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 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 such quality of services as transactions (local or XA), persistence, durability, and others.

There are two possible destinations that a client sends messages to or receive messages from. They are **Topic** and **Queue** (see Figure 3 and Figure 4). 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 17).

Figure 3   Topic - The Publish-Subscribe Model.
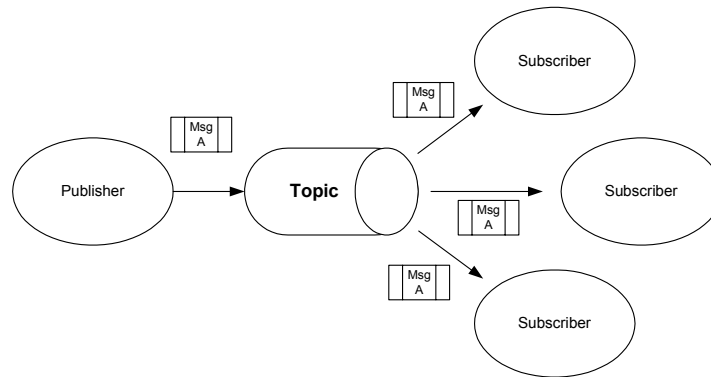
Figure 3 shows multiple subscribers receiving the same messages when the publisher publishes the message to a Topic. This is the Publish-Subscribe model.

Figure 4   Queue - The Point-to-Point Model
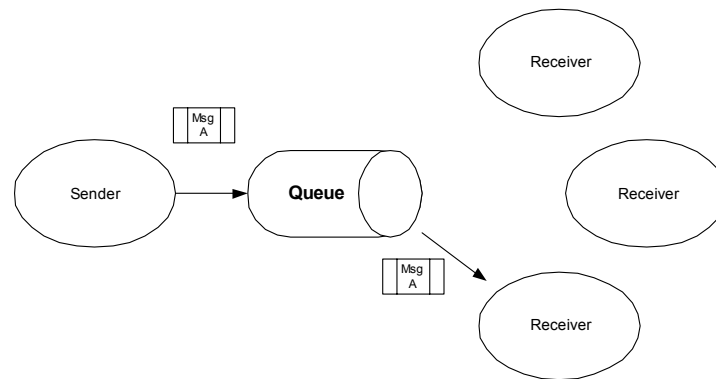
The Point-to-Point model (Figure 4), on the other hand, allows for only one of the receivers to get the message when a sender sends a message to a Queue.

1.3.3. Enterprise JavaBeans (EJBs)

What is Enterprise JavaBean Architecture?

Sun defines Enterprise JavaBean Architecture as follows: Enterprise JavaBean architecture is a component architecture for the development and deployment of component-based distributed business applications. Applications written using the Enterprise JavaBean architecture are scalable, transactional, and multi-user secure. These applications may be written once, and then deployed on any server platform that supports the Enterprise JavaBean specification.

Essentially, if a user writes an EJB, such that the EJB adheres to the EJB specification, the EJB can be deployed on any EJB container regardless of the software vendor that provided the container (application server). The EJB developer does not need to write any code relating to things such as transactions or threads. These services are provided by the container in which an EJB was deployed. The only responsibility of the EJB

developer (or EJB deployer) is to define the attributes of the EJB in its deployment descriptor in order to take advantage of these services offered by the container.

Message Driven Beans

A Message Driven Bean (MDB) is a type of EJB defined by Sun (in the EJB 2.0 specification) in order to deal with asynchronous subscription/publication of JMS messages in a different manner than Entity and Session Beans. An MDB is often compared to a Stateless Session Bean in that it does not have any state context. An MDB 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 implementation needs to implement 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.

Session Beans

A Session Bean is another type of EJB. The Session Bean consists 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 31.)

Entity Beans

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 **Accessing Entity Beans** on page 36.)

1.3.4. XA Transactions

The **X/Open XA** specification defines the interactions between the Transaction Manager (TM) and the Resource Manager (RM). 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 of a particular 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 RMs 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 RMs (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 RMs 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 Message Driven Beans** on page 36.)

1.4 WebLogic e*Way Component Overview

The e*Way Intelligent Adapter for WebLogic interacts with the WebLogic Application Server using three modes.

1 **Synchronous Interaction, e*Gate to WebLogic.** Synchronous interaction means that e*Gate makes a request to WebLogic and waits for a response. This can be thought of as analogous to a phone call in which the caller makes the call and waits for a response.

2 **Asynchronous Interaction, WebLogic EJB to e*Gate (JMS).** Asynchronous interaction means that a request is sent but the sender does not wait for a response. It can be thought of as analogous to a mail message in which mail is sent and forgotten until sometime later when a response is received. The J2EE asynchronous model is the Java Messaging Service (JMS), which dictates how a client application talks to a Queue. The WebLogic EJBs publish to the e*Gate JMS IQ Manager.

3 **Asynchronous Interaction, e*Gate (JMS) to WebLogic Message Driven Bean.** The e*Gate JMS publishes to a WebLogic Application Server Message Driven Bean. A Message Driven Bean (MDB) is an specialized EJB that acts like a trigger which

executes whenever there is activity on a specific Queue. A message published to e*Gate's JMS causes an MDB stored in WebLogic to execute.

1.4.1. Synchronous Interaction, e*Gate to WebLogic Server

Synchronous interaction, in which a requester sends a request and waits while the service is executed before proceeding with the next request, is carried out by the WebLogic e*Way using two component parts, the EJB ETD Builder and the WebLogic e*Way Connection def file.

The **EJB ETD Builder** is used to generate Event Type Definitions (ETDs) from WebLogic's Session and Entity Beans EJB interface classes (Session and Entity Beans, not Message Driven Beans), that represent the methods of the EJB. These methods can then be called from within a Collaboration, making them accessible to the user. The EJB ETD 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 EJB ETD and ETDs for other applications, making the EJB methods available to that application.

The **WebLogic e*Way Connection def file** serves as the basis for configuration files that store the parameters for connecting to the JNDI directory service.

The EJB ETD

The EJB ETD, generated from a WebLogic interface, represents the methods from the EJB that can be called inside a Collaboration. The ETD can be divided into four portions that provide information about the EJB.

Figure 5 EJB ETD nodes represent both Home and Remote Interface methods
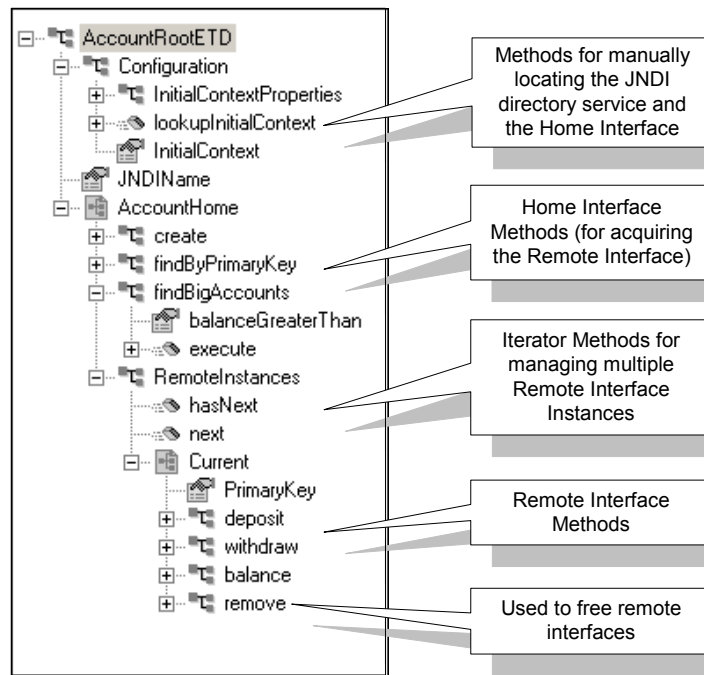
The first portion, the **initial context** and **JNDI Name**, is comprised of the methods for manually locating the home interface which allows the ETD to communicate with the directory service to connect to the EJBs Java objects. The defaults for the configuration are provided by the correct configuration file. The user uses these nodes to override the values in the configuration file. The JNDI Name is set to the default specified in the EJB ETD Builder wizard.

The second portion consists of home interface methods for acquiring the remote interface. The home interface allows the ETD to find and invoke EJB instances. For the example in Figure 5, the home interface method, **findBigAccounts()**, called with the argument balanceGreaterThan (100,000) finds all account EJBs with a balance over 100,000 and assign their remote interface to the Remote Instances ETD node.

The third portion contains the iterator methods **hasNext()** and **next()** for accessing the returned remote interfaces. For the sample in Figure 5, an example of this would be if **findBigAccounts()** returned ten accounts. This would be ten remote interfaces. The iterator methods allow the user to write a **while** loop to loop over multiple remote interfaces and access functionality based on the business logic.

The fourth portion contains remote interface methods that allow processes to be run on the current remote interface.

Remove() should be used with care. Remove(), when called from a Session Bean, frees resources on the server. The danger comes when calling Remove() from an Entity Bean, in which case the Entity Bean is deleted from the database/storage.

1.4.2. Asynchronous Interaction, WebLogic EJBs to e*Gate JMS and e*Gate JMS to WebLogic MDBs

Modes 2 and 3 incorporate asynchronous interaction between WebLogic Server and e*Gate's JMS.

The following sections describe in detail, the e*Way Intelligent Adapter for WebLogic's integration with WebLogic Server using the SeeBeyond implementation of JMS. The e*Way incorporates the SeeBeyond JMS into the WebLogic environment. Essentially, it incorporates the SeeBeyond JMS IQ Manager so that EJBs in the WebLogic container can receive messages from or send messages to e*Gate. There are two schemes: **Message Driven Beans subscribing to SeeBeyond JMS** and **Session Beans publishing/sending to SeeBeyond JMS**.

In order to implement the solutions, two other subsystems are used: the T3 naming service and the EJB container (for Session Beans and Message Driven Beans as defined in EJB 2.0). The naming service allows us to "bind" the following SeeBeyond JMS objects: **TopicConnectionFactory**, **QueueConnectionFactory**, **Topic**(s), and **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 6) illustrates the components involved:

Figure 6   WebLogic Server and WebLogic e*Way Components
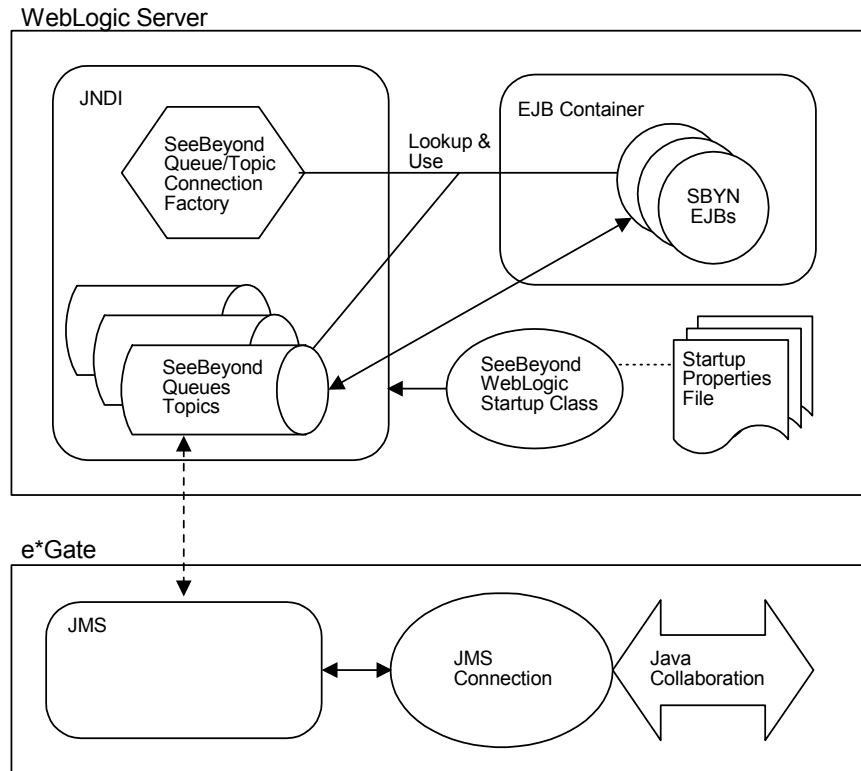
1.4.3. SeeBeyond JMS

As part of the WebLogic e*Way installation, SeeBeyond supplied startup classes install JMS objects into the T3 naming service. Four JMS ConnectionFactory objects are bound to the naming service, **TopicConnectionFactory**, **XATopicConnectionFactory**, **QueueConnectionFactory**, and **XAQueueConnectionFactory**. Moreover, installing the SeeBeyond supplied session and Message Driven Beans installs Topic and Queue objects into the naming service.

Message Flow from e*Gate to WebLogic

For message flow from e*Gate 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 7 shows the components involved in e*Gate to WebLogic mode. The arrows represent message flow.

Figure 7   Message Flow from e*Gate to WebLogic
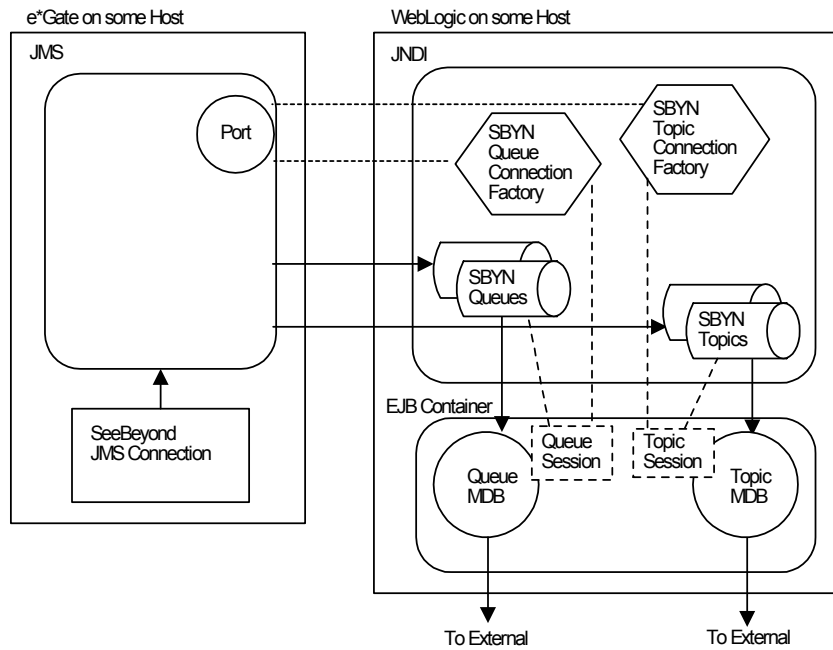
Figure 8 displays an example of the **ejb-jar.xml** for the Topic MDB which receives messages from a SeeBeyond JMS Topic.

Figure 8   ejb-jar.xml - Topic MDB

Figure 9 displays an example of the **weblogic-ejb-jar.xml** for the Topic MDB which receives messages from a SeeBeyond JMS Topic.

Figure 9   weblogic-ejb-jar.xml - Topic MDB
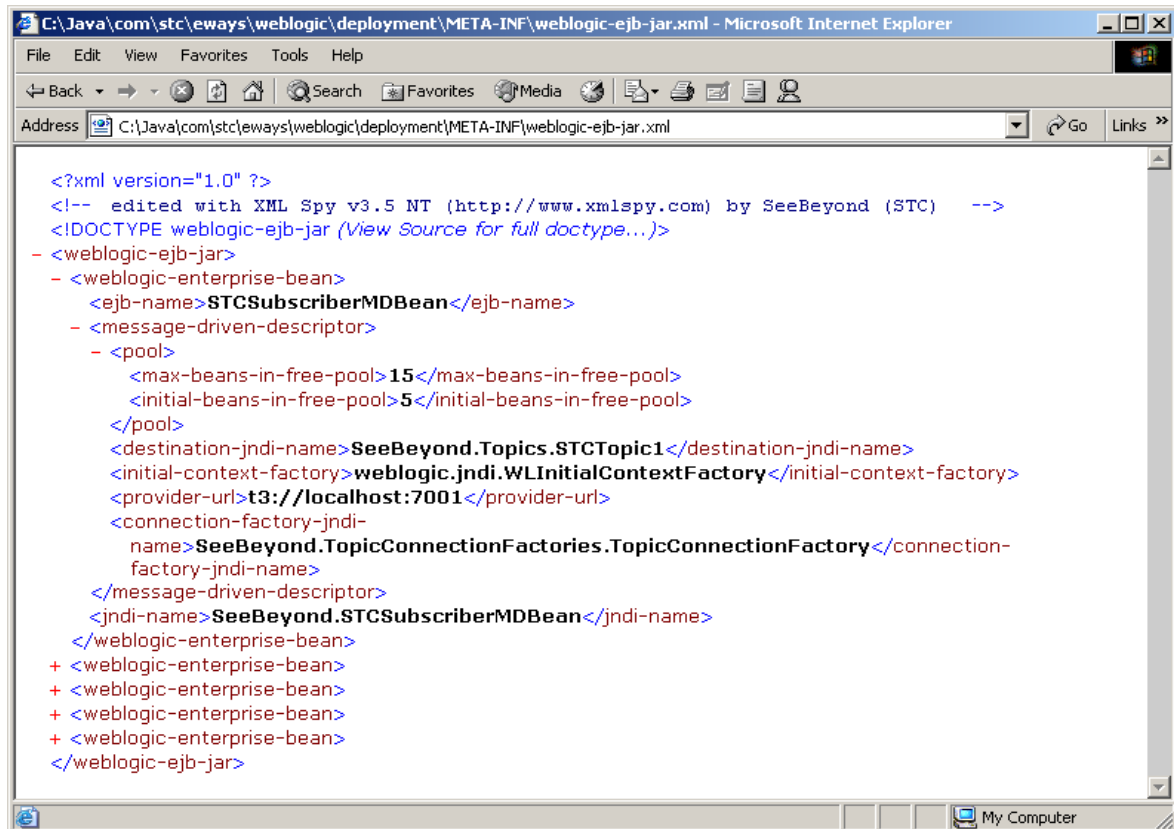
The *destination-jndi-name* of the Topic is **SeeBeyond.Topics.STCTopic1**; this is a SeeBeyond JMS Topic. Using the WebLogic T3 naming service, the two entries *initial-context-factory* and *provider-url* are **weblogic.jndi.WLInitialContextFactory** and **t3://localhost:7003** respectively. Since the container needs to use the SeeBeyond **JMS TopicConnectionFactory**, we specify the SeeBeyond **TopicConnectionFactory** with the entry *connection-factory-jndi-name* 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 the MDB can be deployed and used. The WebLogic Administrative Console does NOT allow the user to create 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 24 to see how the startup class works and how to configure and deploy it.) The three 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 e*Gate
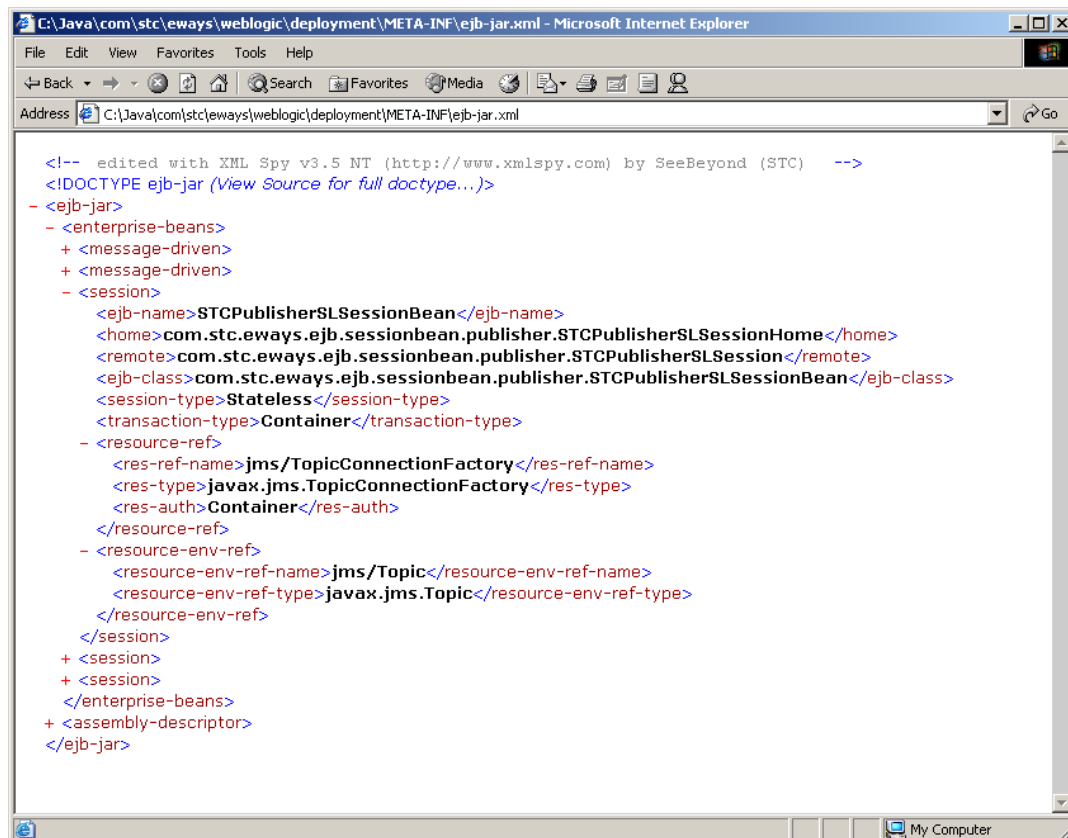
For message flow from WebLogic to e*Gate, Session Beans can publish/send JMS messages to SeeBeyond JMS Topics/Queues.

In addition to the connection factories, the Topic and Queue destinations are also bound to the T3 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 24 for details.) With access to these JMS objects via JNDI, the Session Beans utilize the JMS API's to send the JMS message to e*Gate.

How do the Session Beans know what the JNDI entries are for the connection factory and destinations? 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 10 displays the Session Bean **ejb-jar.xml** deployment descriptor.

Figure 10   Session Bean ejb-jar.xml deployment descriptor

Figure 11 displays the Session Bean **weblogic-ejb-jar.xml** deployment descriptor.

Figure 11   Session Bean weblogic-ejb-jar.xml deployment descriptor
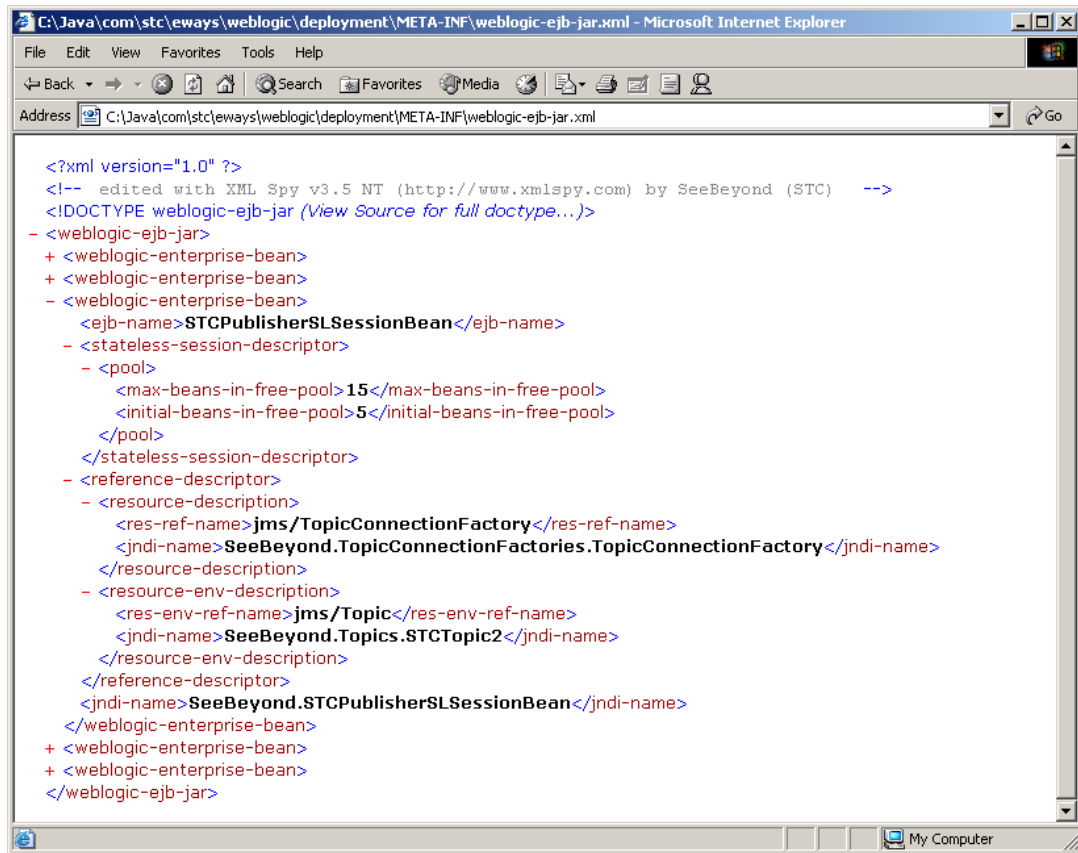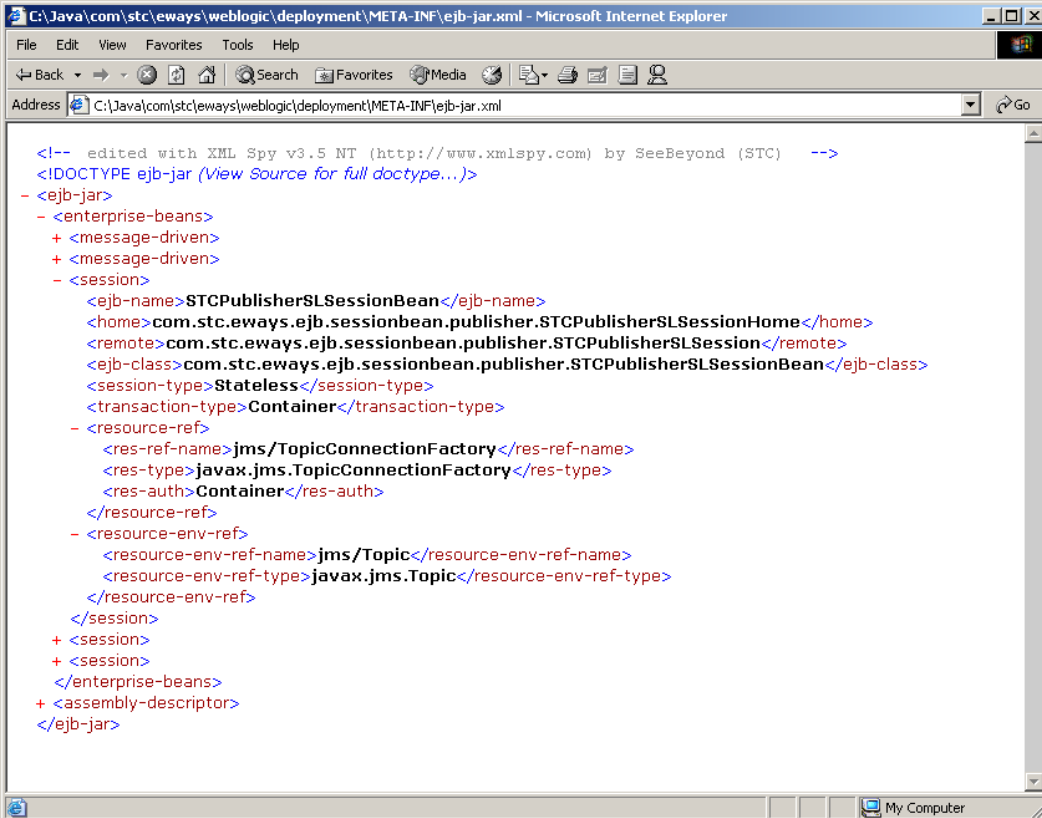
Figure 12 displays a diagram of the components involved for the WebLogic to e*Gate mode. The arrows represent the message flow.

Figure 12   Message Flow from WebLogic to e*Gate
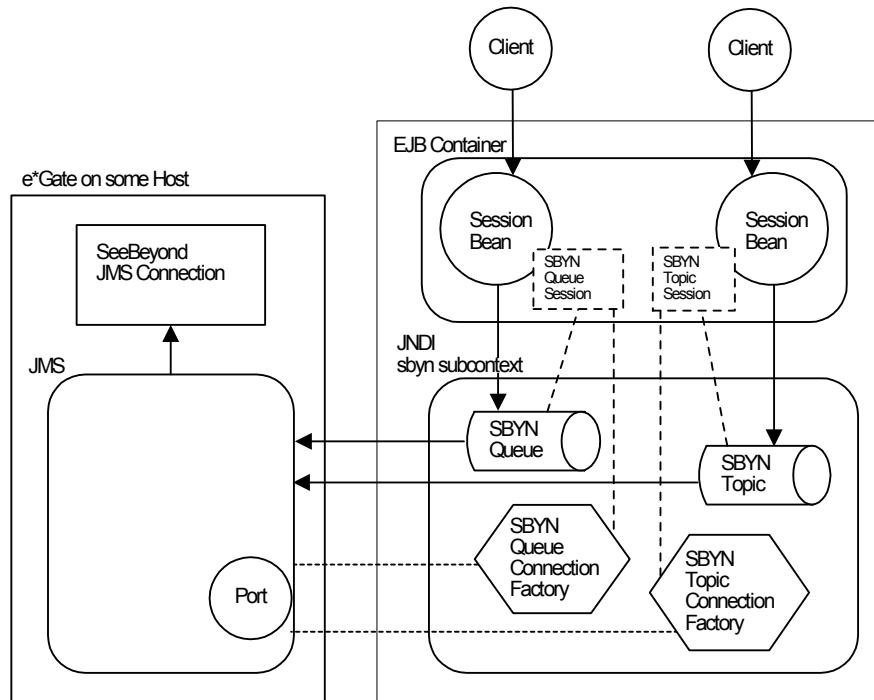
How do the Session Beans know what the JNDI entries are for the connection factory and destinations? Each Session Bean specifies the **TopicConnectionFactory** or **QueueConnectonFactory** with the <resource-ref> element in the **ejb-jar.xml** file. Moreover, the Session Bean specifies the destination 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.
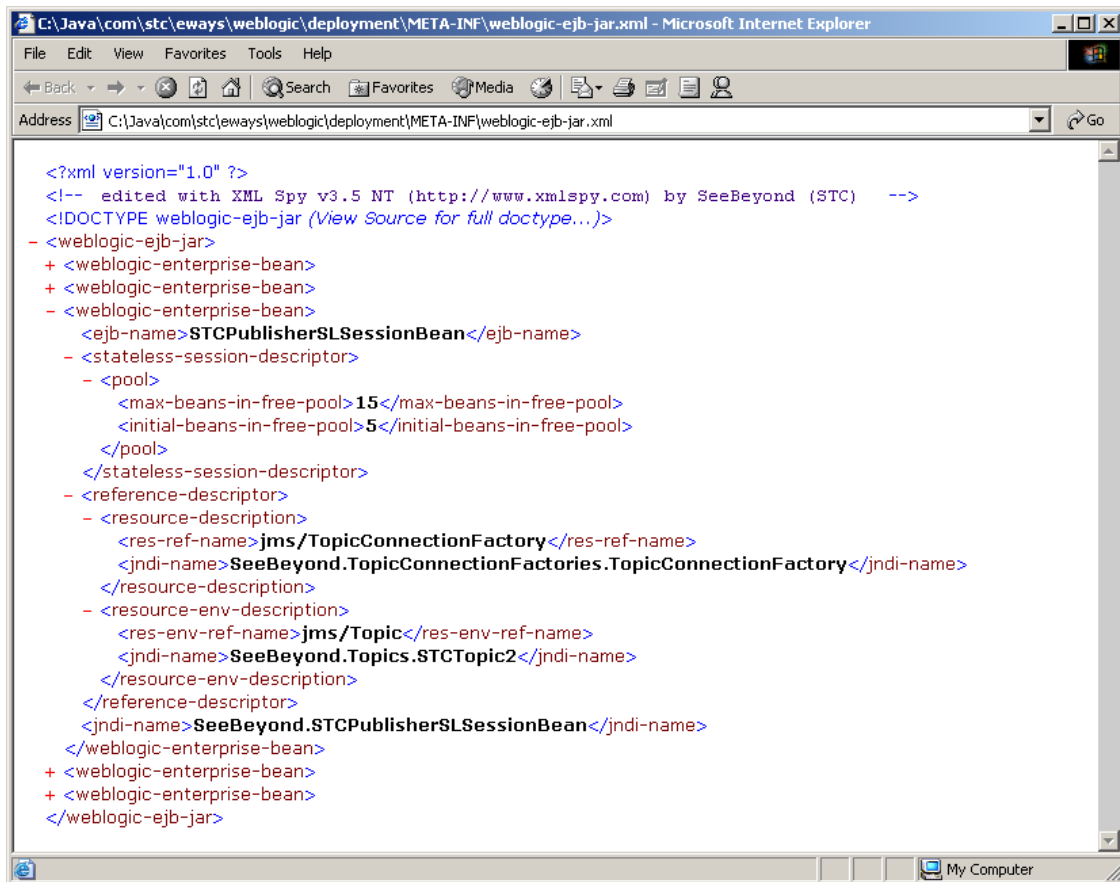
Figure 13 displays an example of the **ejb-jar.xml** deployment descriptor for the Session Bean publishing to a SeeBeyond JMS Topic:

Figure 13   ejbjar.xml deployment descriptor - Session Bean to SeeBeyond JMS Topic

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.

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 14.

Figure 14   weblogic-ejb-jar.xml defines the actual JNDI name

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.

SeeBeyond WebLogic Startup Class

To bind the SeeBeyond JMS objects into the WebLogic T3 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, a SeeBeyond **TopicConnectionFactory**, a **QueueConnectionFactory**, all the configured Topics, and all the configured JMS Queues are instantiated and bound to WebLogic's naming service. 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.

The startup class is called **STCWLStartup.class**. It implements the **weblogic.common.T3StartupDef interface**. **STCWLStartup.class** only needs to implement two methods: **setServices()** and **startup()**. 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.) 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: **name** which is of type **java.lang.String** and **args** which is of type **HashTable**. These two arguments are provided by the server. The name is the name of the startup class. The **args** argument contains name/value pairs that are passed to the startup as program "arguments." These program arguments are defined when the startup class is deployed in the server using the WebLogic Administrative Console.

The startup properties file is read by the startup class when the **startup()** method is invoked by the WebLogic Server. This file, **STCWLStartup.properties**, 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. The following sections describe each section in detail. Comment lines in the properties file start with either a '#' or a '!' character. The following section displays the default **STCWLStartup.properties** file.

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=24053
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**, and **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 e*Gate environment (the name of the event created with the Schema Designer). 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 e*Gate to WebLogic** on page 17 and **Message Flow from WebLogic to e*Gate** on page 20 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 e*Gate environment (the name of the event created with Schema Designer). 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 e*Gate to WebLogic** on page 17 and **Message Flow from WebLogic to e*Gate** on page 20 for more information on the EJB deployment descriptors.

1.4.4. SeeBeyond Sample Message Driven Beans

The previous sections, **Java Naming and Directory Interface (JNDI)** on page 8 and **Java Messaging Service (JMS)** on page 11 describe the JNDI and JMS subsystems. This section finally ties all the concepts that were previously discussed with those for the SeeBeyond MDBs.

There are two MDBs that are deployed in WebLogic: **MDB Subscribing to SeeBeyond Topic** and **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.xm**l. 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.

MDB Receiving from 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.

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>
```

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. 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**. Since this MDB is

subscribing to a SeeBeyond Topic, the <destination-type> is specified as
javax.jms.Topic. In order to create a durable subscriber MDB, the <subscription-durability> is specified as **Durable**. Finally, in the <container-transaction> tag of the
<assembly-descriptor>, we define the transactional mode for the MDB. This MDB does
not use a transaction, so **NotRequired** in the <trans-attribute> tag is specified.

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>
```

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

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. The <destination-jndi-name> tells the container the JNDI name of the
SeeBeyond Topic that this MDB is to subscribe. 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 24 for details.) The
container locates these JNDI objects in its own JNDI as specified by the <initial-context-factory> and <provider-url>.

MDB Subscribing to SeeBeyond Queue

This MDB subscribes to a SeeBeyond JMS Queue. It subscribes to ONLY ONE
SeeBeyond Queue and simply receives and displays the JMS Messages.

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>
```

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. 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**. Since this MDB is subscribing to a SeeBeyond Queue, the user must specify the <destination-type> as javax.jms.Queue. In order to create a durable subscriber MDB, the <subscription-durability> is specified as **Durable**. Finally, in the <container-transaction> tag of the <assembly-descriptor>, the transactional mode is defined for the MDB. This MDB does not use a transaction, so **NotRequired** in the <trans-attribute> tag is specified.

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>
```

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

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. The <destination-jndi-name> tells the container the JNDI name of the SeeBeyond Queue that this MDB is to subscribe. Also, 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 24 for details.) The container locates these JNDI objects in its own JNDI as specified by the <initial-context-factory> and <provider-url>.

Accessing Session Beans

Session Beans can be accessed from an e*Gate Collaboration by using the EJB ETD Builder to create an ETD for the Session Bean. this is done by using create on the home interface to create a remote instance, **hasNext()** and **next()** to access the instance, call methods on the remote instance and then free resources by calling **remove()** when finished.

SeeBeyond Sample Session Beans

There are two Stateless Session Beans available with the WebLogic e*Way: A Session Bean that publishes to a SeeBeyond JMS Topic and another Session Bean that uses the **STCQueueRequestor** to send and receive a message to and from SeeBeyond JMS.

In the sections to follow, there are references to two XML files. These files are used as the Session Bean's deployment descriptor; they 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 need to define in order to deploy the MDBs.

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.

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>
```

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. 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.

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>
```

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

Again, 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/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. The Topic and
TopicConnectionFactory must have already been created and registered with JNDI by
the startup class. (See **SeeBeyond WebLogic Startup Class** on page 24 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 to a SeeBeyond JMS Queue and get back a reply on
the request sent. It exposes the remote method, request(), which takes a String as an
argument. The Session Bean gets the message and sends it to a SeeBeyond JMS Queue.
The Session Bean then gets a reply from e*Gate.

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>
```

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. 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.STCPublisherSLSessionHom*e. 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. Notice that 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**. 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.

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>
```

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

As before, 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. Notice that the value for the jndi-name tag for the resource name **jms/QueueConnectionFactory** is **SeeBeyond.QueueConnectionFactories.QueueConnectionFactory** and the value for the *jndi-name* tag for the **jms/Queue** entry is **SeeBeyond.Queues.STCQueue2**. These 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 24 for details.) The container locates these JNDI objects in its own JNDI as specified by the <initial-context-factory> and <provider-url>.

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 e*Gate Collaboration by using the EJB ETD Builder to create an ETD for the Session Bean. this is done by using Creators or Finders on the home interface to create remote instances, **hasNext()** and **next()** to access the instance, 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).

1.4.5. SeeBeyond Sample XA Message Driven Beans

An 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 (ONE) 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 43 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 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.

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 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 * (which means all methods).

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**.

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. The <destination-jndi-name> tells the container the JNDI name of the
SeeBeyond Queue to which this MDB is to subscribe. Also, the <connection-factory-
jndi-name> 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 24 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.

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 <resource-ref>, <resource-env-ref>, <resource-ref-name>, and <resource-env-ref-name> tags of the Bean's deployment descriptor.

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 to "lazy loading" of the JMS objects.

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 43 for details.) This Session Bean exposes two remote methods, createAccountAndPublish() and getBalance(). createAccountAndPublish() 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 e*Gate 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 should see that this record is not inserted. Moreover, e*Gate does not receive the rollback message.

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>
```

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. 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. 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. So, 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 also, 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.

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).

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** and the value for the jndi-name tag for the jms/Topic entry is **SeeBeyond.Topics.STCTopic3**.   These

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 24 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.

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. Figure 15 displays the configuration file created for the SeeBeyond JMS IQ Manager:

Figure 15   SeeBeyond JMS IQ Manager - Trace Settings
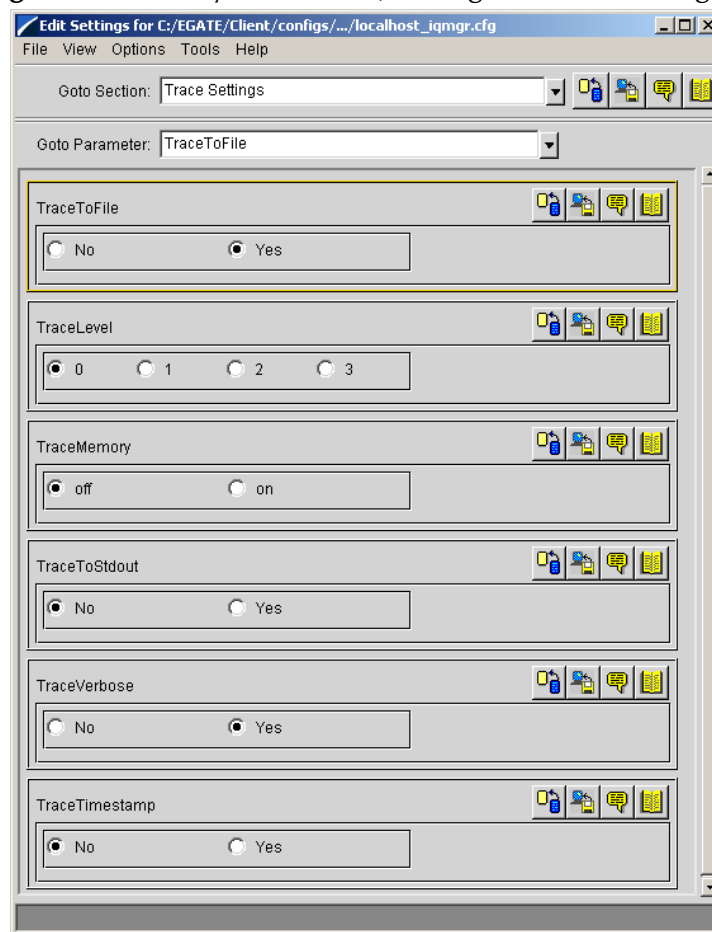
The JMS server log should appear something like this :

```
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:636f6d2e7365656265796f6e642e6a6d732e636c69656e742e53544358415265573
6f75726365
…
```

```
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, 2002 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, 2002 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, 2002 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, 2002 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, 2002 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, 2002 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, 2002 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, 2002 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, 2002 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <ResourceDescriptor[demoXAPool]: endResourceUse, Number of active
requests:0>
####<Apr 4, 2002 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, 2002 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, 2002 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, 2002 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, 2002 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <> <>
<000000> <5:fa80c71858e3d95b: XA.end(rm=demoXAPool, xar=demoXAPool, flags=TMSUCCESS)>
```

```
####<Apr 4, 2002 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, 2002 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, 2002 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <ResourceDescriptor[demoXAPool]: endResourceUse, Number of active
requests:0>
####<Apr 4, 2002 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, 2002 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, 2002 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <5:fa80c71858e3d95b: XA.prepare DONE:ok>
####<Apr 4, 2002 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, 2002 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <> <>
<000000> <5:fa80c71858e3d95b: XA.prepare(rm=demoXAPool, xar=demoXAPool>
####<Apr 4, 2002 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, 2002 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <5:fa80c71858e3d95b: XA.prepare DONE:ok>
####<Apr 4, 2002 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <>
<5:fa80c71858e3d95b> <000000> <ResourceDescriptor[demoXAPool]: endResourceUse, Number of active
requests:0>
####<Apr 4, 2002 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, 2002 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, 2002 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, 2002 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, 2002 5:49:53 PM PST> <Debug> <JTA> <localhost> <examplesServer> <Thread-3> <> <>
<000000> <XAResource[demoXAPool].commit(xid=5:fa80c71858e3d95b,onePhase=false)>
####<Apr 4, 2002 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, 2002 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, 2002 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

examples-dataSource-demoXAPool

As part of its examples server, WebLogic pre-installs a pre-configured datasource named examples-dataSource-demoXAPool (see **Figure 16 on page 44**) and associates it with the pre-installed connection pool named demoXAPool (see **Figure 17 on page 44**). 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 e*Way. Use the figures below to verify that the WebLogic examples server is properly set up to work with the sample e*Gate schemas/EJBs discussed in this document.

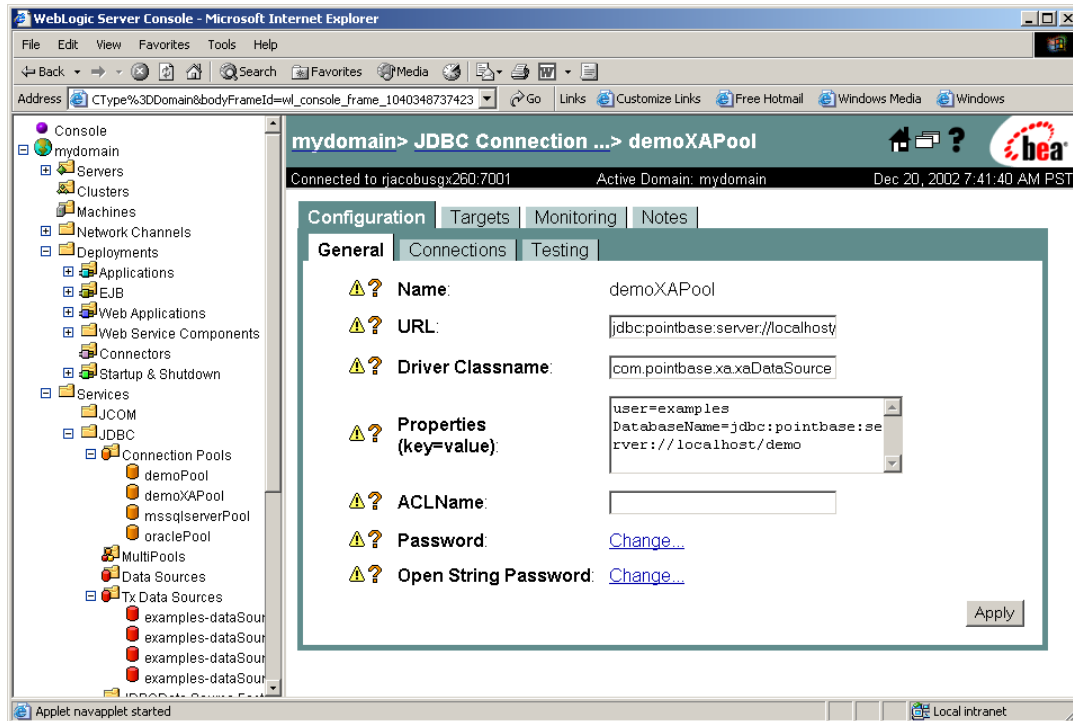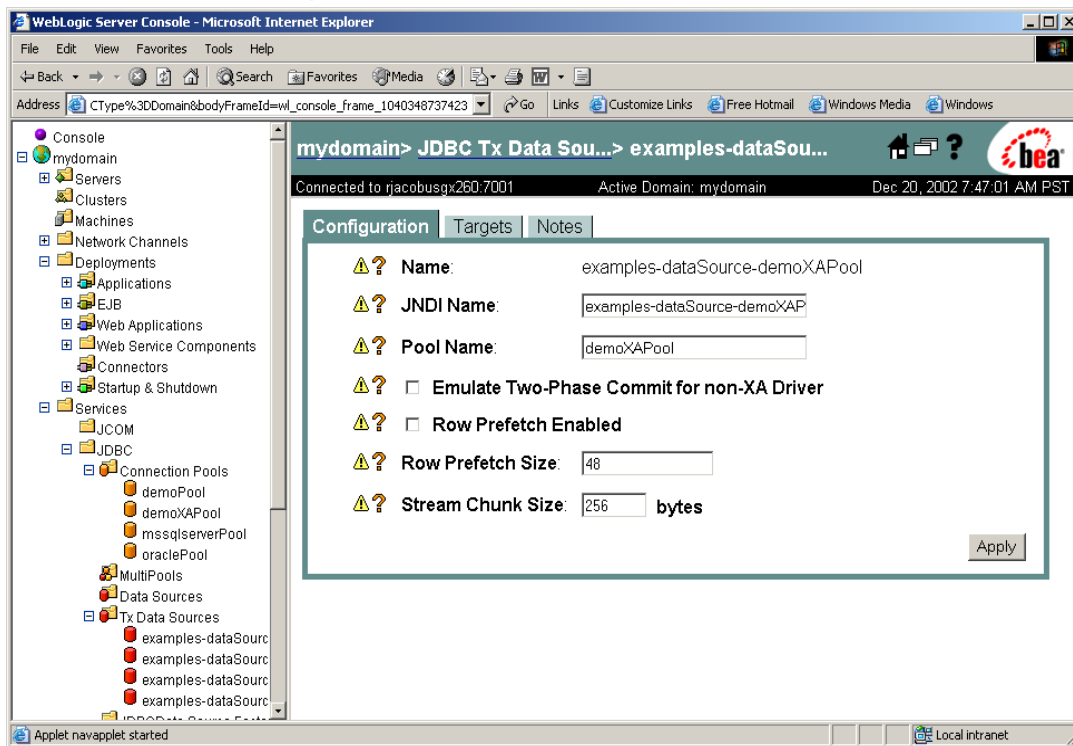**Figure 16**   WebLogic (7.0) Administrative Console - demoXAPool

Figure 17   WebLogic (7.0) Administrative Console - demoXAPool

1.5 Supported Operating Systems

The WebLogic e*Way is available on the following operating systems:

- Windows 2000, Windows XP, and Windows Server 2003
- HP-UX 11.0 and 11i (PA-RISC)
- IBM AIX 5.1L and 5.2
- Sun Solaris 8 and 9
- Korean Windows 2000, Windows XP, and Windows Server 2003
- Korean HP-UX 11.0 and 11i (PA-RISC)

1.6 System Requirements

To use the WebLogic e*Way, you need the following:

- An e*Gate Participating Host.
- A TCP/IP network connection.
- Additional disk space for e*Way executable, configuration, library, and script files. The disk space is required on both the Participating and the Registry Host. Additional disk space is required to process and queue the data that this e*Way processes; the amount necessary varies based on the type and size of the data being processed, and any external applications performing the processing.

*Note: Open and review the **Readme.txt** for the WebLogic e*Way for any additional requirements prior to installation. The Readme.txt is located on the Installation CD_ROM at setup\addons\ewweblogic.*

1.6.1. External System Requirements

- BEA WebLogic Server 6.1, 7.0, or 8.1.

Installation

This chapter describes the procedures for installing the WebLogic e*Way.

- **"Windows" on page 46**
- **"UNIX" on page 47**
- **"Files Created by the Installation" on page 48**

2.1 Windows

2.1.1. Pre-installation

- Exit all Windows programs before running the setup program, including any antivirus applications.
- You must have Administrator privileges to install this e*Way.

2.1.2. Installation Procedure

To install the WebLogic e*Way on a Windows system

1 Log in as an Administrator to the workstation on which you are installing the e*Way.

2 Insert the e*Way installation CD-ROM into the CD-ROM drive.

3 If the CD-ROM drive's Autorun feature is enabled, the setup application launches automatically; skip ahead to step 4. Otherwise, use the Windows Explorer or the Control Panel's **Add/Remove Applications** feature to launch the file **setup.exe** on the CD-ROM drive.

4 The InstallShield setup application launches. Follow the installation instructions until you come to the **Please choose the product to install** dialog box.

5 Select **e*Gate Integrator**, then click **Next**.

6 Follow the on-screen instructions until you come to the second **Please choose the product to install** dialog box.

7 Clear the check boxes for all selections except **Add-ons**, and then click **Next**.

8 Follow the on-screen instructions until you come to the **Select Components** dialog box.

9 Highlight (but do not check) **e*Ways**, and then click the **Change** button. The **SelectSub-components** dialog box appears.

10 Select the **WebLogic e*Way**. Click the continue button to return to the Select Components dialog box, then click **Next**.

11 Follow the rest of the on-screen instructions to install the WebLogic e*Way. Be sure to install the e*Way files in the suggested client installation directory. The installation utility detects and suggests the appropriate installation directory. *Unless you are directed to do so by SeeBeyond support personnel, do not change the suggested installation directory setting.*

Important: *ebj.jar and weblogic.jar (with ejb.jar preceding weblogic.jar) must be added to the classpath prior to using the EJB ETD Builder.*

Note: *Once you have installed and configured this e*Way, you must incorporate it into a schema by defining and associating the appropriate Collaborations, Collaboration Rules, IQs, and Event Types before this e*Way can perform its intended functions. For more information about any of these procedures, please see the online Help.*

*For more information about configuring e*Ways or how to use the e*Way Editor, see the **e*Gate Integrator User's Guide**.*

2.2 UNIX

2.2.1. Pre-installation

You do not require root privileges to install this e*Way. Log in under the user name that you wish to own the e*Way files. Be sure that this user has sufficient privileges to create files in the e*Gate directory tree.

2.2.2. Installation Procedure

To install the WebLogic e*Way on a UNIX system

1 Log in on the workstation containing the CD-ROM drive, and insert the CD-ROM into the drive.

2 If necessary, mount the CD-ROM drive.

3 At the shell prompt, type:

cd /cdrom

4 Start the installation script by typing:

setup.sh

5 A menu of options appears. Select the **Install e*Way** option. Then, follow the additional on-screen directions.

Note: *Be sure to install the e*Way files in the suggested **client** installation directory. The installation utility detects and suggests the appropriate installation directory. **Unless you are directed to do so by SeeBeyond support personnel, do not change the suggested "installation directory" setting.***

6 After installation is complete, exit the installation utility and launch the Schema Designer.

Important: *ebj.jar and weblogic.jar (with ejb.jar preceding weblogic.jar) must be added to the classpath prior to using the EJB ETD Builder.*

Note: *Once you have installed and configured this e*Way, you must incorporate it into a schema by defining and associating the appropriate Collaborations, Collaboration Rules, IQs, and Event Types before this e*Way can perform its intended functions. For more information about any of these procedures, please see the online Help system.*

*For more information about configuring e*Ways or how to use the e*Way Editor, see the **e*Gate Integrator User's Guide**.*

2.3 Files/Directories Created by the Installation

The WebLogic e*Way installation process installs the following files (see Table 1) within the e*Gate directory tree. Files are installed within the **egate\client** tree on the Participating Host and committed to the **default** schema on the Registry Host.

Table 1 Files Created by the Installation

e*Gate Directory	File(s)
	stcewweblogic.ctl
\external\ewweblogic\classes\	stcwlstartup.jar
\external\ewweblogic\configs\startup\	STCWLStartup.properties
\configs\ejbetd\	weblogic.def

Configuration

This chapter describes how to configure the components of the WebLogic e*Way and WebLogic Server. Configuration for the WebLogic e*Way differs for the Synchronous using the EJB ETD Builder and the Asynchronous Implementations which use JMS.

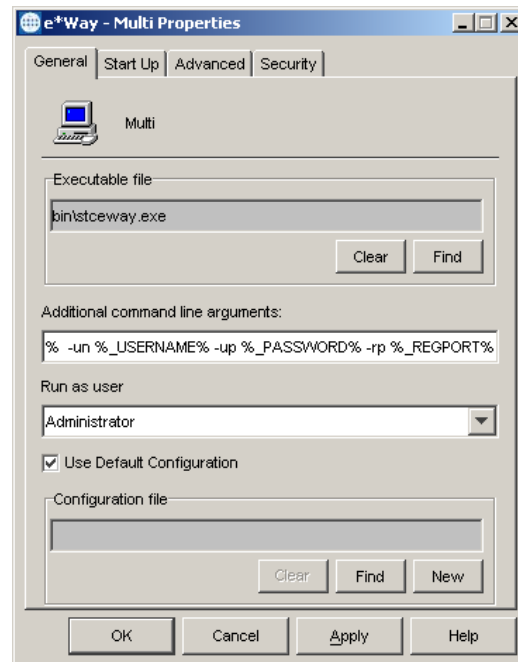## 3.1 Configuring the Components for Synchronous Interaction Implementation using the EJB ETD Builder

If you have not already done so, launch the Schema Designer, selecting a sample schema for Synchronous Interaction using the EJB ETD Builder. The configuration for this implementation differs from that of the Asynchronous Interaction implementations in that JMS is not used.

3.1.1. Multi-Mode e*Way Configuration Parameters (Synchronous Interaction)

e*Way configuration parameters are set using the e*Way Editor.

To change Multi-Mode e*Way configuration parameters

1 In the Schema Designer's Component editor, select the e*Way you want to configure and display its properties (see Figure 18). The Executable file for Multi-Mode e*Ways is **stceway.exe**.

Figure 18   Multi-mode e*Way Properties

2 In the **Additional Command Line Arguments** box, type any additional command line arguments that the e*Way may require, taking care to insert them *at the end* of the existing command-line string. Be careful not to change any of the default arguments unless you have specific need to do so.

3 Under **Configuration File**, click **New** to create a new file, **Find** to select an existing configuration file, or **Edit** to edit the currently selected file. The Editor opens to edit settings for the Multi-Mode e*Way. The Edit Settings dialog box opens.

4 Configure the e*Way as needed for your system. Any necessary settings for a specific sample are provided in the Implementation Chapter.

For more information about how to use the e*Way Editor, see the e*Way Editor's online Help or the *e*Gate Integrator User's Guide*.

For more information about the Multi-Mode e*Way, see the *Standard e*Way Intelligent Adapter User's Guide*.
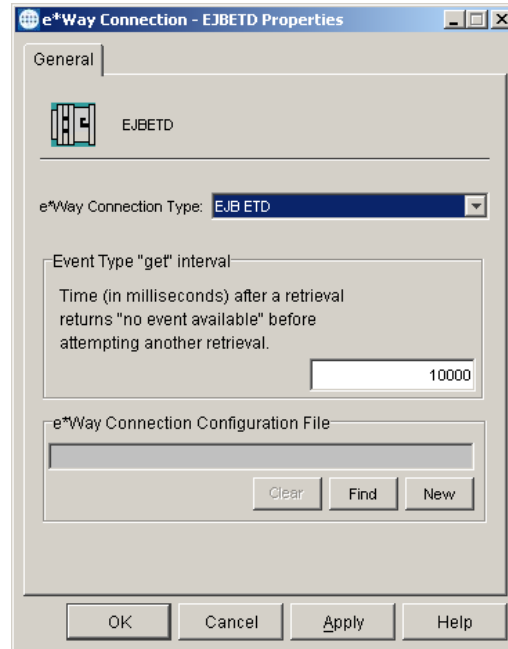
3.1.2. EJB ETD e*Way Connection

The EJB ETD e*Way Connection Type provides the specific parameters necessary for JNDI and EJB access. To create and configure an EJB ETD e*Way Connection do the following:

To create the e*Way Connection

1 In the Schema Designer's **Component** editor, select the **e*Way Connections** folder and on the palette, click on the **Create a New e*Way Connection** button.

2 Enter a name (for this sample, EJBETD) and create the **e*Way Connection**.

3   Double-click the new **e*Way Connection**. The **e*Way Connection Properties** dialog box opens (seeFigure 19).

Figure 19   e*Way Connection Properties

4   From the **e*Way Connection Type** drop-down box, select **EJB ETD**. The **Event Type "get" interval** is not used in this case. Use the default setting.

5   Click **New** under the e*Way Connection Configuration File field.

6   The **Edit Settings** dialog box opens. Enter the correct parameters for your e*Way Connection as defined in the following pages. When all parameters have been entered, from the **File** menu, click **Save** and **Promote to Run Time**.

Configuring the ETD e*Way Connection

The EJB ETD e*Way connection parameters provide information for locating EJBs using JNDI, as well as security and connection functionality. At a minimum the parameters **java.naming.provider.url** and **java.naming.factory.initial** must be set before the e*Way can be used. For additional information regarding JNDI specific parameters go to **http://java.sun.com/products/jndi**. For further information on WebLogic specific parameters go to **http://e-docs.bea.com/wls/docs61////jndi/jndi.html**.

The EJB ETD WebLogic e*Way configuration parameters are organized into the following sections.

   ▪ **General Settings** on page 52

   ▪ **JNDI InitialContext Settings** on page 52

3.1.3. General Settings

This section contains the following parameters:

- **Type** on page 52

- **Class** on page 52

- **Property.Tag** on page 52

Type

Description

Specifies the connector type. The default value should always be used.

Required Value

EJB ETD is always the configured default for EJB ETD connections.

Class

Description

Specifies the class name of the EJB ETD connector object. The default value should always be used.

Required Value

com.stc.eways.ejbetd.EJBETDConnector is the configured default for EJB ETD connections.

Property.Tag

Description

Specifies the data source identity. This parameter is required by the current EBobConnectorFactory.

Required Value

A valid data source package name.

3.1.4. JNDI InitialContext Settings

This section contains the following parameters:

- **java.naming.provider.url** on page 53

- **java.naming.dns.url** on page 53

- **java.naming.factory.initial** on page 53

- **java.naming.factory.object** on page 53

- **java.naming.factory.state** on page 54

- **java.naming.factory.control** on page 54

- **java.naming.factory.url.pkgs** on page 54

- **java.naming.security.protocol** on page 54

- **java.naming.security.authentication** on page 54

- **java.naming.security.principal** on page 55

- **java.naming.security.credentials** on page 55

- **java.naming.authoritative** on page 55
- **java.naming.batchsize** on page 55
- **java.naming.referral** on page 56
- **java.naming.language** on page 56
- **weblogic.jndi.createIntermediateContexts** on page 56
- **weblogic.jndi.delegate.environment** on page 56
- **weblogic.jndi.pinToPrimaryServer** on page 56
- **weblogic.jndi.provider.rjvm** on page 57
- **weblogic.jndi.replicateBindings** on page 57
- **weblogic.jndi.ssl.client.certificate** on page 57
- **weblogic.jndi.ssl.client.key_password** on page 57
- **weblogic.jndi.ssl.root.ca.fingerprints** on page 57
- **weblogic.jndi.ssl.server.name** on page 58
- **weblogic.jndi.use.iiop.service.provider** on page 58

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.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.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 class names of 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 class names of 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.control

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

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

java.naming.factory.url.pkgs

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

Required Value
Package prefixes used to load URL context factories, separated by a colon. com.sun.jndi.url is always added to end of list.

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.

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.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

A user **name** or **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.credentials

Description

Specifies the principal's (user's) credentials for the authentication scheme determined by the authentication scheme 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

A password, certificate (file), or 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.authoritative

Description

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

Required 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 is a suggestion to the provider to return the results of operations in batches of a specified size, so that the provider can optimize 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.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

"**follow**", "**ignore**", or "**throw**". If not specified it defaults to the service provider default.

java.naming.language

Description

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

Required Value

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.

weblogic.jndi.createIntermediateContexts

Description

Specifies the how 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

"**true**" or "**false**". If not specified it defaults to the service provider default.

weblogic.jndi.delegate.environment

Description

Specifies the JNDI environment to use for connecting to a third-party naming service through the WebLogic Server. When specified 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.pinToPrimaryServer

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
"**true**" or "**false**". The configured default is false.

weblogic.jndi.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.replicateBindings

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 grants that any modification to the naming tree is replicated across the cluster, This ensures that any server can act as a naming server for the entire cluster.

Required Value
"**true**" or "**false**". The default value is true.

weblogic.jndi.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 DER_encoded X.509 certificates. Other than 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.ssl.client.key_password

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

Required Value
A password.

weblogic.jndi.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.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.use.iiop.service.provider

Description

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.

3.2 Configuring Components for Asynchronous Interaction Implementation using SeeBeyond JMS

If you have not already done so, launch the Schema Designer, selecting a sample schema for Asynchronous Interaction using the SeeBeyond JMS. The configuration for this implementation differs from that of the Synchronous Interaction implementations in that JMS is used.

3.2.1. JMS IQ Manager

Verify that the IQ Manager Type is set to **SeeBeyond JMS** (see Figure 20).

Since the WebLogic e*Way publishes Events to JMS, the IQ Manager type in your Participating Host must be set to SeeBeyond JMS.

Figure 20   SeeBeyond JMS IQ Manager
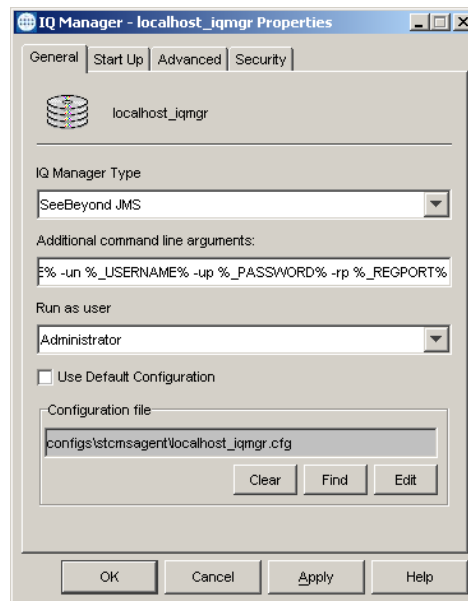
3.2.2. Multi-Mode e*Way Configuration Parameters (asynchronous interaction)

e*Way configuration parameters are set using the e*Way Editor.

To change Multi-Mode e*Way configuration parameters

1 In the Schema Designer's Component editor, select the e*Way you want to configure and display its properties. The Executable file for Multi-mode e*Ways is **stceway.exe**.

2 In the **Additional Command Line Arguments** box, type any additional command line arguments that the e*Way may require, taking care to insert them *at the end* of

the existing command-line string. Be careful not to change any of the default arguments unless you have specific need to do so.

3   Under **Configuration File**, click **New** to create a new file, **Find** to select an existing configuration file, or **Edit** to edit the currently selected file. The Editor opens to edit settings for the Multi-Mode e*Way. The Edit Settings dialog box opens.

4   Configure the e*Way as needed for your system. Any necessary settings for a specific sample are provided in the Implementation Chapter.

For more information about how to use the e*Way Editor, see the e*Way Editor's online Help or the *e*Gate Integrator User's Guide*.

For more information about the Multi-Mode e*Way, see the *Standard e*Way Intelligent Adapter User's Guide*.

3.2.3. e*Way Connection

Create and configure an e*Way Connection. The connection type should be set to "**SeeBeyond JMS**". (For the sample, the e*Way Connection is referred to as "JMSQueueConsumer".) Set the Event Type "get" interval to 5000.

To create and configure an SeeBeyond JMS e*Way Connection do the following:

Create the e*Way Connection

1   In the Schema Designer's **Component** editor, select the **e*Way Connections** folder and on the palette, click on the **Create a New e*Way Connection** button.

2   Enter a name (for this sample, conJMSQueueConsumer) and create the **e*Way Connection**.

3   Double-click the new **e*Way Connection**. The **e*Way Connection Properties** dialog box opens.

4   From the **e*Way Connection Type** drop-down box, select **SeeBeyond JMS**.Set the Event Type "get" interval to 5000.

Click **New** under the e*Way Connection Configuration File field. The **New JMS e*Way Connect** dialog box opens. Indicate whether the e*Way Connection is intended for:

▪ External: Connect to JMS IQ Manager which is not in the current schema

▪ Internal: Connect to JMS IQ Manager within this schema

If **External** is selected, the user must configure e*Way Connection, including ServerNamer, Hostname, and Port Number.If **Internal** is selected, the user selects a JMS IQ Manager from the drop-down, and the ServerName, Hostname, and Port Number are read in from the Registry.(see Figure 21).

Figure 21 JMS e*Way Connection properties
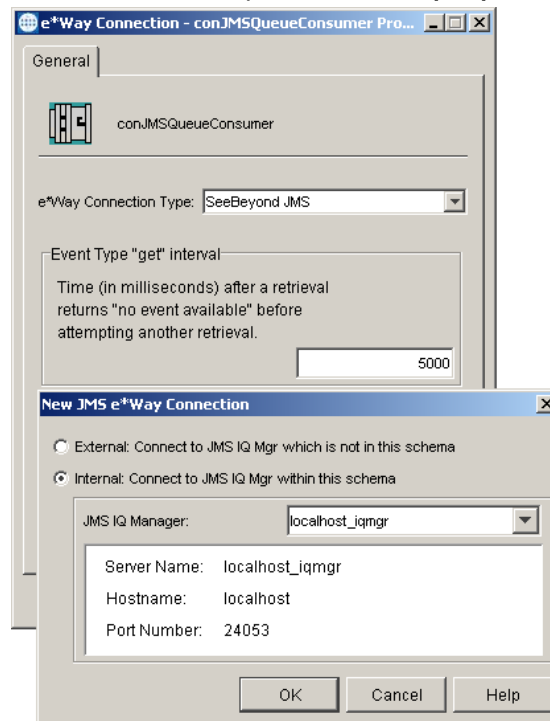
5 Click **Edit** under the e*Way Connection Configuration File field. **The Edit Settings** dialog box opens. Enter the correct parameters for your e*Way Connection as defined in the following pages. When all parameters have been entered, from the **File** menu, click **Save** and **Promote to Run Time**.

Configuring the JMS e*Way Connection parameters

For more information about the JMS e*Way Connections, see the *SeeBeyond JMS IQ Manager User's Guide*.

This section describes the JMS e*Way configuration parameters. For SeeBeyond JMS, the e*Way Connection configuration parameters are organized into two sections:

- **General Settings** on page 61
- **Message Service** on page 63

General Settings

The General Settings control overall properties of the e*Way Connection. This section contains the following parameters:

- **Connection Type** on page 62
- **Transaction Type** on page 62
- **Delivery Mode** on page 62
- **Maximum Number of Bytes to read** on page 63
- **Default Outgoing Message Type** on page 63

- **Message Selector** on page 63

- **Factory Class Name** on page 63

Connection Type

Description
Specifies the type of connection to be established.

For classic publication/subscription behavior, where each message is delivered to all current subscribers to the Topic, select **Topic**.

For point-to-point behavior (equivalent to "subscriber pooling" for conventional IQs), where each message is delivered to only one recipient in the pool, select **Queue.**

Required Values
Topic or **Queue**.

Transaction Type

Description
Specifies the type of transaction to be instantiated.

Important: *XA transactions for the WebLogic e*Way are managed by the WebLogic TransactionManager, NOT the e*Gate TransactionManager. For XA transactions make sure that the XAConnectionFactory(ies) are configured for the startup class.*

In **Internal** (one-phase transactional) style, a commit is necessary: The message is not saved until the either a commit or a rollback is received.

In **XA-compliant** (two-phase transactional style) a two-phase commit is done: The sender sends a prepare, and the commit occurs if and only if all receivers are prepared. Collaborations that use Guaranteed Exactly Once Delivery (GEOD) of Events require XA-compliant transaction types. *Note: This does not affect XA Transactions for the WebLogic e*Way. Read "Important" above.*

In **Non-Transactional** mode, the message is automatically saved on the server; no commit is necessary.

Required Values
Internal, **non-transactional,** or **XA-compliant**.

Delivery Mode

Description
Setting **Delivery Mode** to **Persistent** guarantees that the JMS IQ Manager stores each message safely to disk. Setting it to **Non-Persistent** does not guarantee that the message is stored safely to disk. **Non-Persistent** provides better performance but no recovery.

Required Values
Non-Persistent or **Persistent**.

Important: *If the JMS IQ Manager halts when in **Non-Persistent** mode, undelivered messages are lost.*

Maximum Number of Bytes to read

Description

Your setting for this parameter depends on the size of your messages. For example, if you can anticipate that very large messages will be read, set this parameter accordingly.

Required Values

1 to **200000000.** The default is **5000.**

Default Outgoing Message Type

Description

For messages that carry no payload, or carry only a simple TextMessage payload (such as XML documents), you can set this option to **Text.**

For messages whose payload is known to be incompatible with other messaging systems, or whose payload is unknown, keep this option set to **Bytes**.

Required Values

Bytes or **Text**.

Message Selector

Description

Specifies the Message Selector to be used for subscriptions.

Required Values

A string. The maximum length of query is set to 512 characters, including a null terminator.

Note: This parameter does not check syntax. If the syntax is incorrect, the selector is ignored and the subscriber is not created.

Factory Class Name

Description

For SeeBeyond e*Way Connections, keep the default setting:
com.stc.common.collabService.SBYNJMSFactory

Required Values

Default: **com.stc.common.collabService.SBYNJMSFactory**

Message Service

The parameters in this section specify the low-level information required to establish the JMS. This section contains the following parameters:

- **Server Name** on page 64
- **Host Name** on page 64
- **Port Number** on page 64
- **Maximum Message Cache Size** on page 64

Server Name

Description
Specifies the name of the server (JMS IQ Manager) with which e*Gate communicates.

Required Values
A valid server name.

Host Name

Description
Specifies the name of the host on which with which the server (JMS IQ Manager) running.

Required Values
A valid host name.

Port Number

Description
Specifies the port number on which the JMS IQ Manager is running.

Required Values
A valid port number between **2000** and **1000000000**.

Maximum Message Cache Size

Description
Specifies the maximum size of the message cache in bytes.

Required Values
An integer between **1** and **2147483647**.

Configure the e*Way as needed for your system.

3.3   Configuring the WebLogic Server Components

The following section provides directions for configuring WebLogic Server for asynchronous interaction (modes 2 and 3) with e*Gate (JMS). Setup directions are provided for both WebLogic version 6.1 and 7.0.

- **Configuration for WebLogic 6.1** on page 65
- **Configuration for WebLogic 7.0** on page 69

3.3.1.   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 open the **wlserver6.1** directory, then open the **config** directory. Sample servers created on WebLogic Server are located in the config directory (see Figure 22).

Figure 22   WebLogic Server 6.1 File Structure
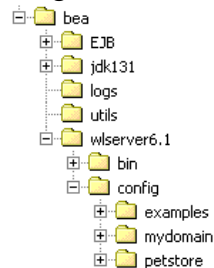
1   Verify that the system classpath contains ebj.jar and weblogic.jar (with ejb.jar preceding weblogic.jar).

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

stcejbweblogic.jar
stcwlstartup.jar
STCWLStartup.properties

- stcejbweblogic.jar can be found on the Installation CD-ROM in the sample folder at:

      samples\ewweblogic

- stcwlstartup.jar can be found at:

      eGate\Server\registry\repository\default\external\ewweblogic
      \classes

- STCWLStartup.properties can be found at:

      eGate\Server\registry\repository\default\external\ewweblogic
      \configs\startup\

3   Modify **startExamplesServer.cmd** and **setExamplesServer.cmd** located at <WL-HOME/config/examples. Append 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\stcjms.jar;.\lib\stcwlstartup.jar
```

For setExampleEnv.cmd

```
setCLASSPATH=%CLASSPATH%;%WL_HOME%\lib\stcjms.jar;%WL_HOME%\lib\st
cwlstartup.jar
```

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

4 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, do the following to modify the deployment descriptors. Extract stcejbweblogic.jar and edit **META-INF\weblogic-ejb-jar.xml**. For each Bean that is run, find the Provider_URL tag of the deployment descriptor and change the port number from 7003 to 7001. Then re-jar (zip) stcejbweblogic.jar.

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

6 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 67.) 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 23   WebLogic Server Console - Create a New StartupClass
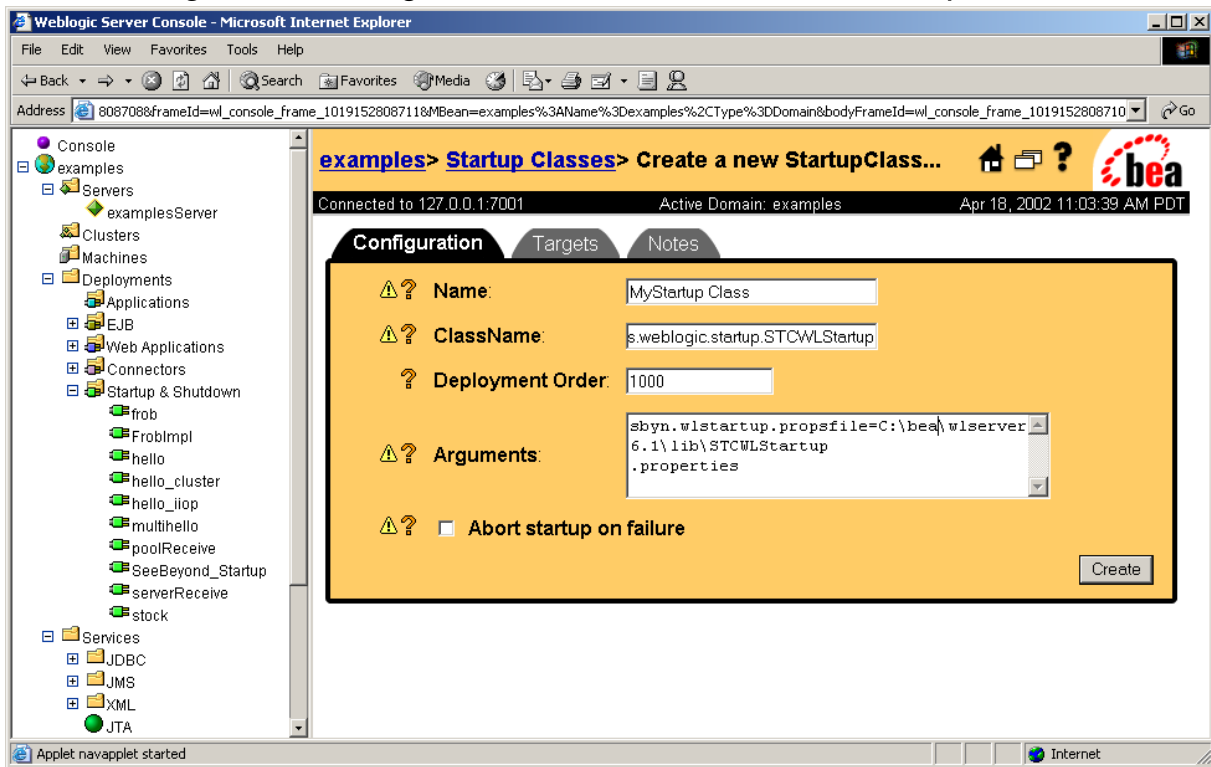
7   Click on the **Targets** tab and move the new server instance from Available to Chosen using the arrow button. Click **Apply**.

8   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
```

9   Start the Default Console.

10   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 24).

Figure 24 View the JNDI Tree
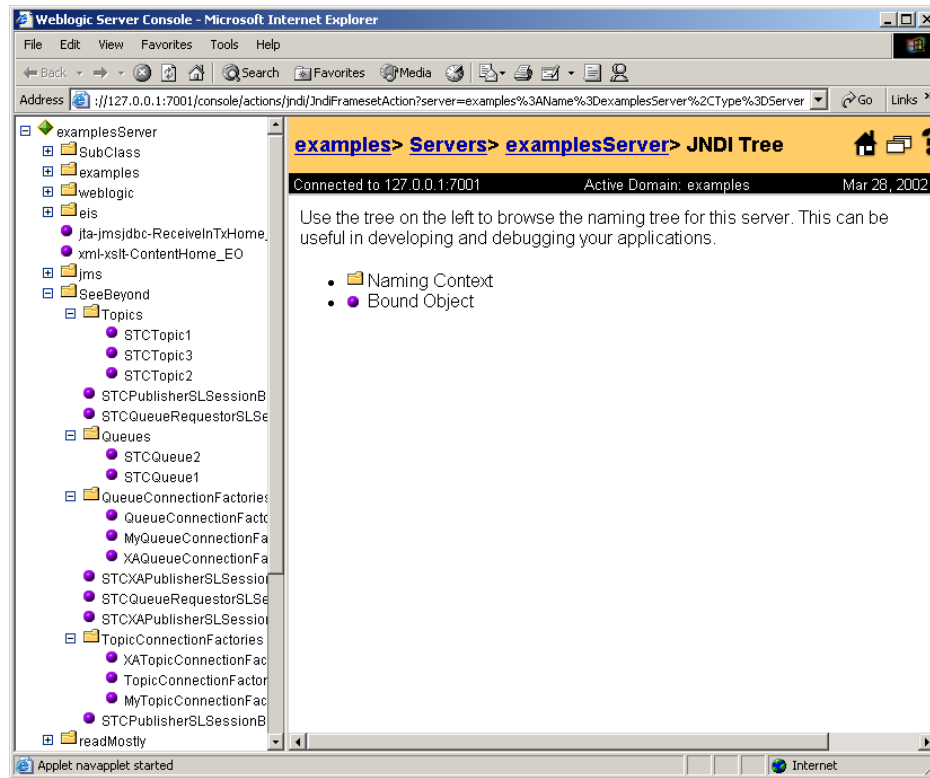
11 On the Console, click on Examples, Deployments, EJB. Click on **Install a new EJB**. Browse to and select <WL-HOME>\wlserver6.1\lib**stcejbweblogic.jar**. Click **Upload** to install it on the WebLogic Administration Server.

3.3.2. 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 25).

Figure 25   WebLogic Server File Structure
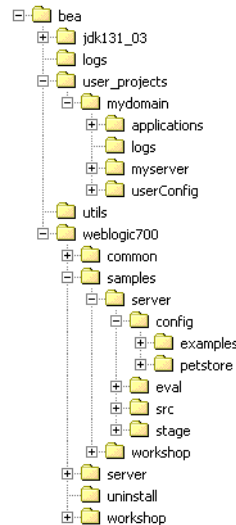
1   Import the **JMSAsynchProducersConsumers** sample schema into e*Gate (see **Installing a Sample Schema** on page 85).

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

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

stcejbweblogic.jar
stcwlstartup.jar
STCWLStartup.properties
stcjms.jar

▪ stcejbweblogic.jar can be found on the Installation CD-ROM in the sample folder at:

```
samples\ewweblogic
```

▪ stcwlstartup.jar can be found at:

```
eGate\Server\registry\repository\default\external\ewweblogic
\classes
```

▪ STCWLStartup.properties can be found at:

```
eGate\Server\registry\repository\default\external\ewweblogic
\configs\startup
```

▪ stcjms.jar can be found at:

```
eGate\server\regestry\repository\default\classes
```

4 Modify **startExamplesServer.cmd** and **setExamplesServer.cmd** located at <BEA-HOME>\user_projects\<domain name>, appending 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\stcjms.jar;C:\bea\weblog
ic700\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\stcjms.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, do the following to modify the deployment descriptors.

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 71.) 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 26   WebLogic Server Console - Create a New StartupClass
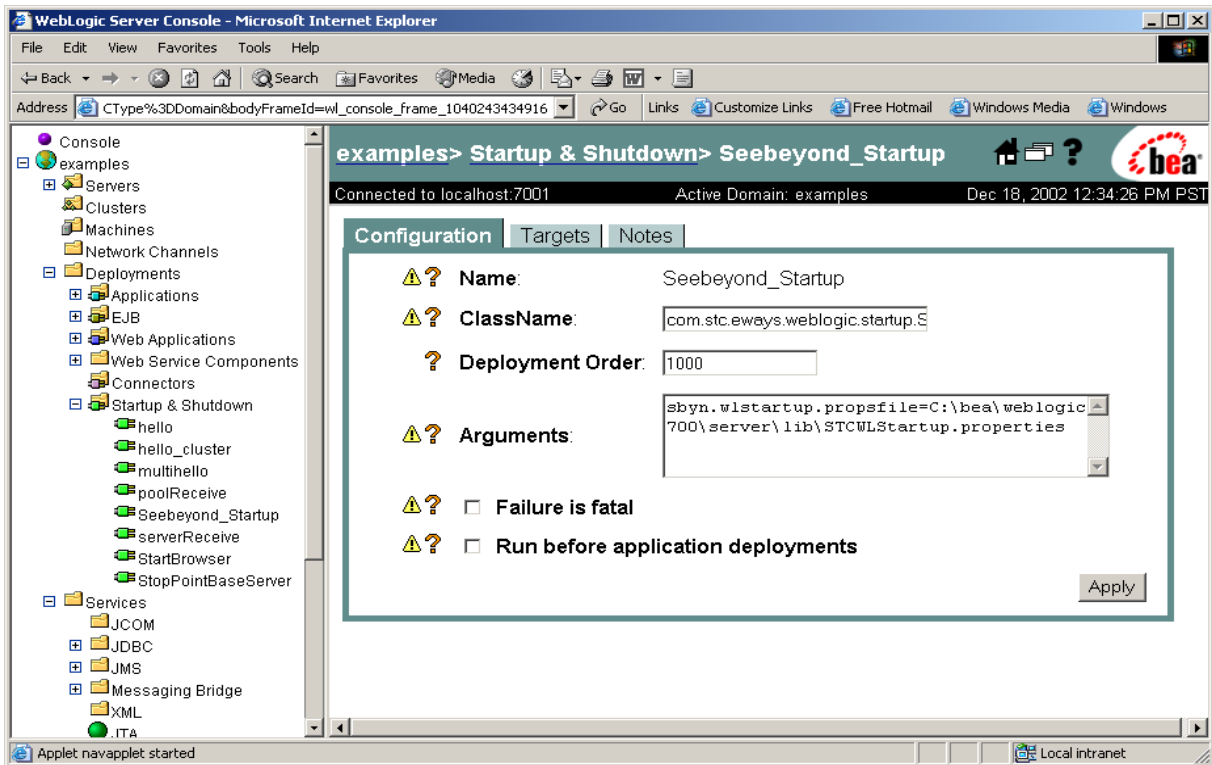
8   Click on the **Targets** tab and move the new server instance from Available to Chosen using the arrow button. Click **Apply**.

9   Stop and restart the server. To stop and restart the server do the following:

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** and **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 24).

Figure 27 View the JNDI Tree
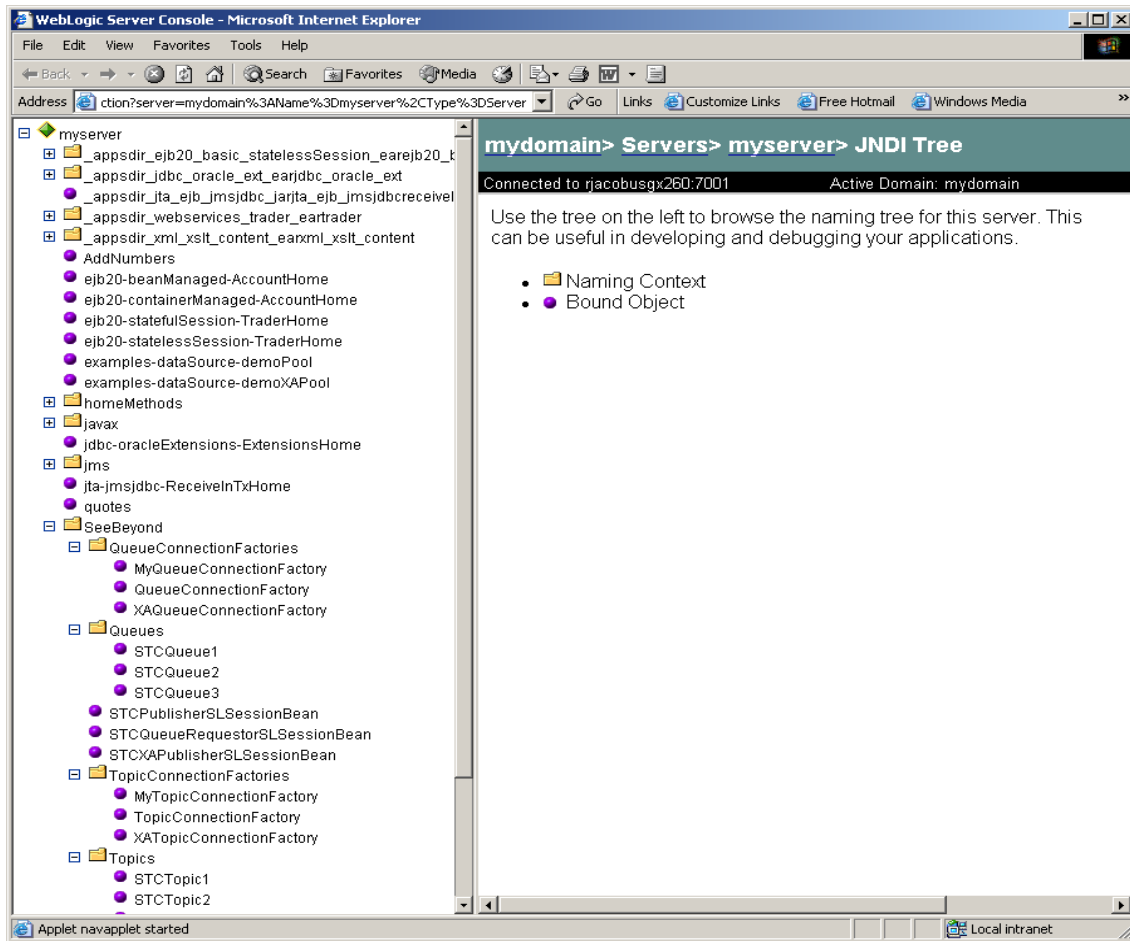
12 From the Navigator pane on the left, click on Examples, Deployments, EJB. Click on **Configure a new EJB**.

Note: *Before deploying the EJB, make sure that the JMS IQ Manager is running (see* **Executing the Schema** *on page 115). 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.

F Repeat steps 1-5 (A-E) for **AddNumbersEJB.jar**.

3.4 Append Classpaths for All Collaboration Rules

This step applies to both **Synchronous** and **Asynchronous** implementations of the WebLogic e*Way. Before running a schema, open e*Gate Schema Designer to the schema. Open the Collaboration Rules folder and open each of the Collaboration Rules in the Collaboration Rules Editor. From the menu bar of the editor, select Tools, Options, and in the Java Classpaths dialog box, select **Append** and click **OK** (see Figure 28). This must be done for each of the Collaboration Rules for each schema.

Figure 28   Appending the Classpath for each Collaboration Rule
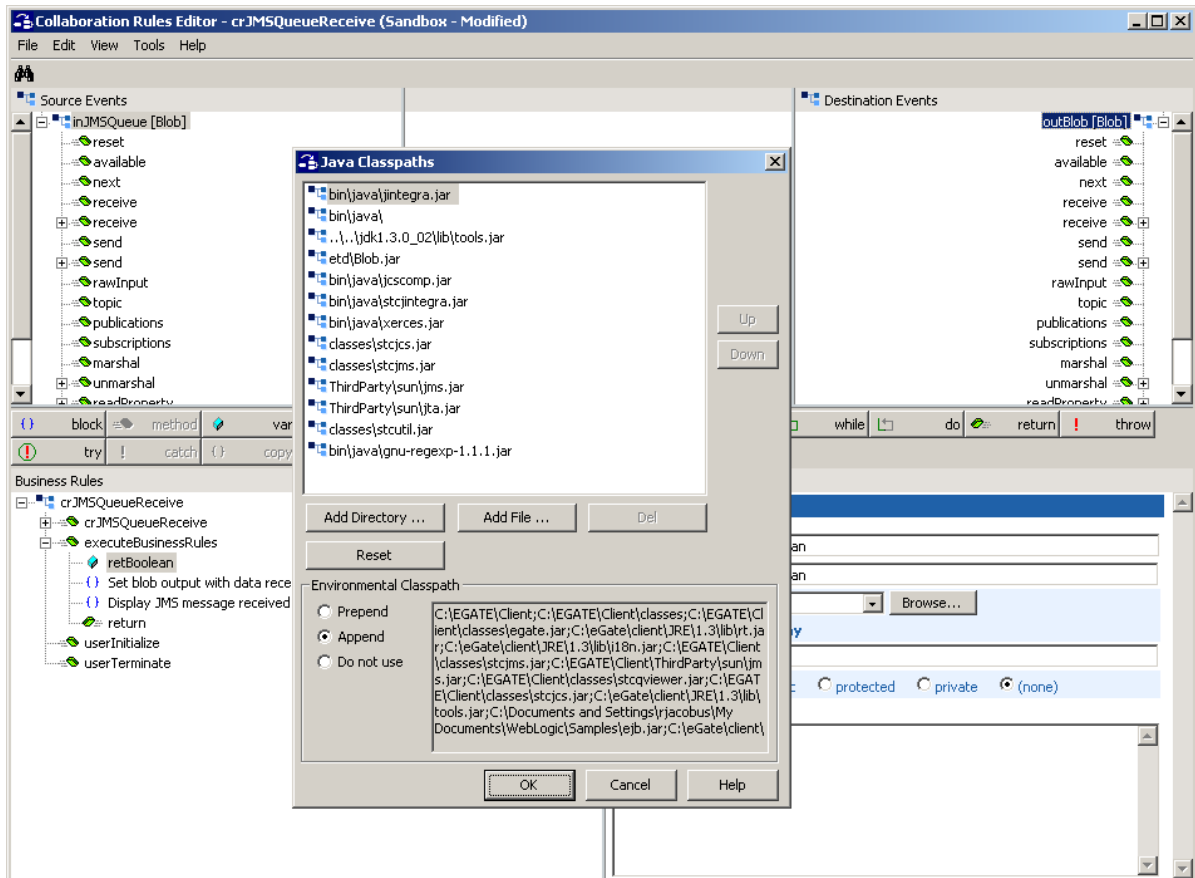
Implementation

This chapter contains basic information for implementing the WebLogic e*Way in a production environment. Examples are given for creating and configuring the necessary components for the WebLogic e*Way sample schemas. For more information on creating and configuring e*Way components see the *e*Gate Integrator User's Guide* or the *e*Gate Schema Designer's online Help system.*

4.1 Implementation Process: Overview

The WebLogic e*Way is an *application specific* e*Way that allows e*Gate to connect with WebLogic. When the e*Way Intelligent Adapter for WebLogic is installed with e*Gate Integrator, schema's can be created and configured using the e*Gate Schema Designer. A schema is an organization scheme containing the parameters for the components that control, route, and transform data as it moves through e*Gate in a predefined system configuration.

The process overview presents the steps involved in creating an e*Way schema. For the most part, these steps have already been implemented for the imported sample schemas. To implement the WebLogic e*Way within an e*Gate system requires the following:

- Install the WebLogic e*Way

- Create one or more e*Way components and configure their properties and parameters.

- Define the necessary e*Way Connections and configure their properties and parameters.

- Define Event Type Definitions (ETDs) to package the data being exchanged with the WebLogic application server.

- Configure the IQ Manager (and/or IQs, for the EJB ETD implementations) to suit the schemas specific needs.

- Define Collaboration Rules to extract selected information from a source Event and process it according to the Collaboration Service associated with the Collaboration Rules.

- Define Collaborations to receive and process Event Types and then forward the output to other e*Gate components.

▪ Configure any other components necessary to complete the schema.

▪ Test the schema and make any necessary adjustments.

For additional information on creating or modifying any component within the e*Gate Schema Designer, see the *e*Gate Schema Designer's online Help system*.

4.2 Sample Implementations

The following pages contain sample implementations that serve to explain how the components for the WebLogic e*Way are created for each mode.

▪ Mode 1: **Synchronous Interaction, e*Gate to WebLogic Server** on page 76.

▪ Mode 2: **Asynchronous Interaction, WebLogic EJB to e*Gate JMS** on page 82.

▪ Mode 3: **Asynchronous Interaction, e*Gate JMS to WebLogic Message Driven Bean** on page 83.

The section, **"Creating the Sample Schemas" on page 84** describes the various sample schemas for the WebLogic e*Way available on the installation CD-ROM.

The Host and Control Broker are automatically created and configured during the e*Gate installation. The default name for each is the name of the host on which the e*Gate Schema Designer GUI is installed.

Note: *For more information about creating or modifying any component within the e*Gate Schema Designer, see the e*Gate Schema Designer's online Help system.*

4.3 Considerations

▪ **Add ejb.jar and weblogic.jar to the system classpath**. **ejb.jar** and **weblogic.jar** (with ejb.jar preceding weblogic.jar in order) must be added to the system classpath prior to using the EJB ETD Builder for the ETD to be generated successfully. The ejb.jar file can be found at **http://java.sun.com/products/ejb/docs.html** and selecting Download Class Files.

▪ **Classes in the default package cannot be used by the EJB ETD Builder**. Users cannot generate ETDs for EJBs in the unnamed default package. An error message to this effect appears if this is attempted.

▪ **XA transactions for the WebLogic e*Way** are managed by the WebLogic TransactionManager, NOT the e*Gate TransactionManager or in the e*Way Connection parameters. For XA transactions make sure that the XAConnectionFactory(ies) are configured for the startup class.

▪ **weblogic.jar and the EJB interface classes** must be located on the Participating Host that runs the Collaborations using those EJBs, or can be mapped to as a remote connection.

- **Entries in the STCWLStartup.properties file** must not include any spaces in the values or property keys. Spaces are interpreted as unrecognizable characters.

- **Generating ETDS for EJB local interfaces** is not currently supported.

- A **Readme.txt** is available at ..\setup\addons\ewweblogic\readme.txt on the installation CD-ROM, that provides the latest information on required ESRs and recent changes to the e*Way. An additional **Readme.html** is available for the WebLogic e*Way samples at ..\samples\ewweblogic\Readme.htm that contains supplementary information on implementing the sample schemas.

4.4   Synchronous Interaction, e*Gate to WebLogic Server

Implementing the WebLogic e*Way schema in mode 1: Synchronous Interaction requires the following four steps:

- Step 1: Build the ETD from the interface classes.
- Step 2: Configure the e*Way Connections.
- Step 3: Build Collaboration Rules to instantiate the remote interfaces.
- Step 4: Bind the e*Way Connection to the Collaboration Rules.

Step 1: Build the ETD from the interface classes

The following procedures describe how to create an Event Type Definition (ETD) using the EJB ETD Builder.

Note: *ejb.jar and weblogic.jar (with ejb.jar preceding weblogic.jar in order) must be placed in the system classpath for the ETD to be generated successfully.*

Important: **Classes in the default package cannot be used by the EJB ETD Builder.** Users cannot generate ETDs for EJBs in the unnamed default package. An error message to this effect appears if this is attempted.

1 Select the **Event Types** folder on the **Components** tab of the e*Gate Navigator and click the **Create a New Event Type** button on the palette.

2 Enter the name of the **Event Type** in the **New Event Type Component** window, then click **OK**. (For this sample, the Event Type is defined as "**AddNumbers**".)

3 Double-click the new Event Type to open the **Event Type's Properties** dialog box. Click the **New** button under the Event Type Definition field. The ETD Editor opens.

4 From the ETD Editor **File** menu, click **New**. The **New Event Type Definition** window opens displaying e*Gate's ETD Wizards. Select the **EJB Wizard** and click OK (see Figure 29).

Figure 29 New Event Type Definition - EJB ETD Wizard
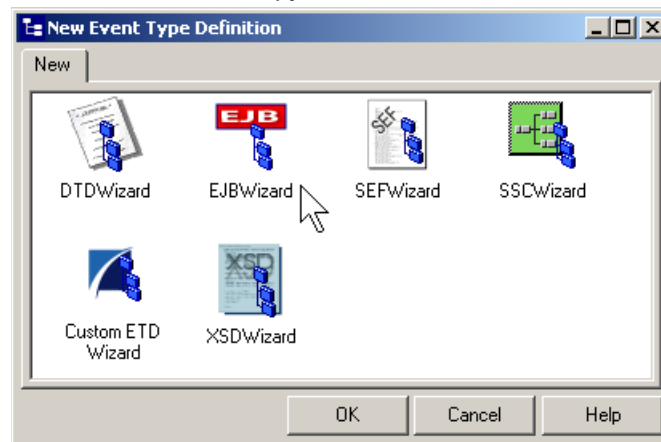
5 The **EJB ETD Builder Wizard** opens. Click **Next** to continue.

6 In the Package Name field, enter the last segment of the Java package name. For instance, the package name for com.stc.ejbetd.AddNumbers1 would be AddNumbers1. (See Figure 30)

7 Enter the Root Node Name. This name appears as the root node of the new ETD.

Figure 30 New Event Type Definition - EJB ETD Wizard
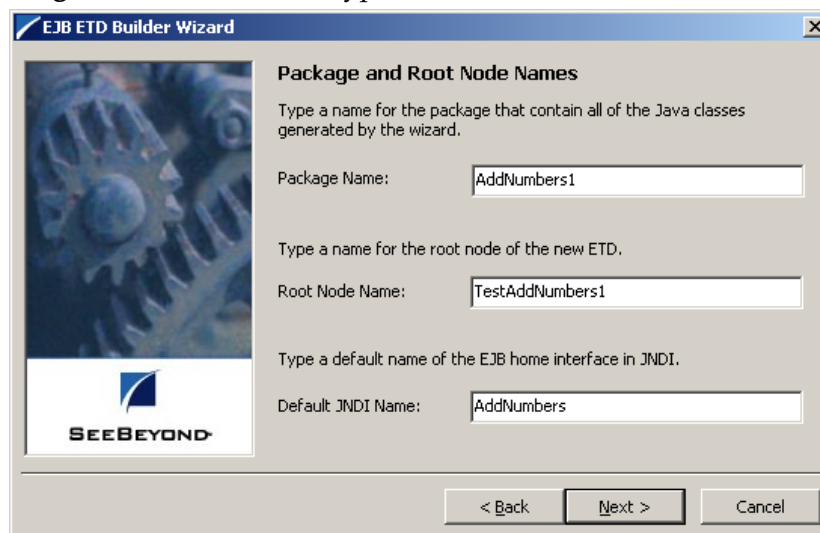
8 Enter the Default JNDI Name. This is the default name in JNDI and can be overridden in the Collaboration Rules. It is exposed as a node JNDI name in the ETD. The JNDI name can usually be found in the application server specific Deployment Descriptor of the EJB, for instance, **weblogic_ejb_jar.xml**.

9 The **Home and Remote Interfaces** page of the EJB ETD Wizard opens (see Figure 31). Enter the class file root, in the **Class File Root** field using the **File** and **Directories** buttons to browse to and locate the correct file. This can be either a jar file or a root directory which contains class files. If the root directory is used, the directory above the top-level java package should be entered into the Class File Root field.

Figure 31 Home and Remote Interfaces - EJB ETD Wizard
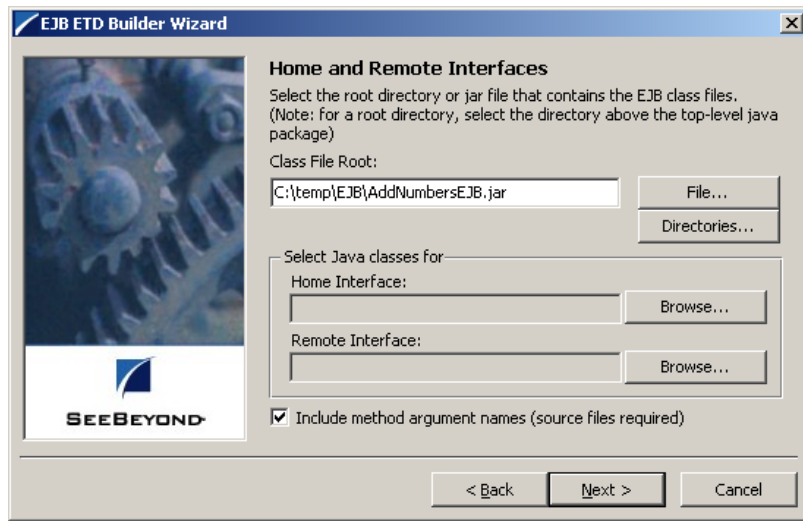
10 Enter the home interface by clicking **Browse** to the right of the **Home Interface** field. The Home Interface dialog box appears. Expand the root directory or jar file and select the home interface file (see Figure 32). Click **OK**

Figure 32 Home Interfaces - EJB ETD Wizard
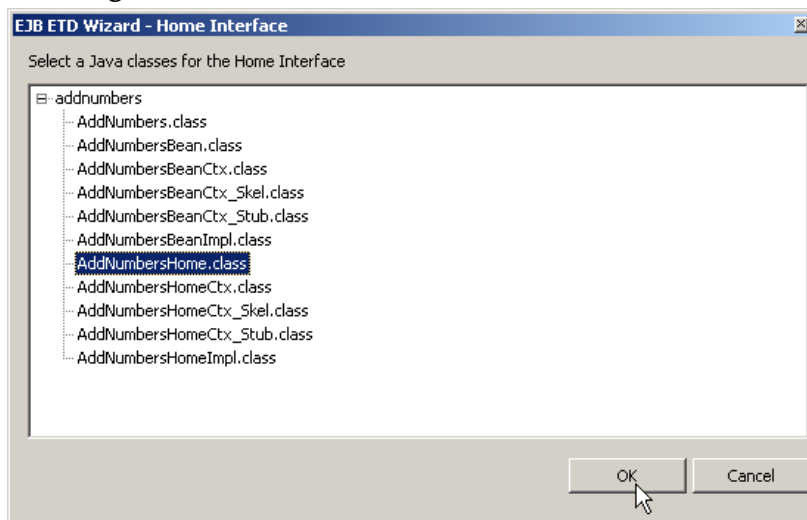
11 Enter the remote interface by clicking **Browse** to the right of the **Remote Interface** field. The Remote Interface dialog box appears. Expand the root directory or jar file and select the home interface file. Click **OK**

12 The **Include method argument names** checkbox is selected by default. Leave this checked unless source code is unavailable. This allows the exact parameter names to be displayed in the ETD (for example: "stockSymbols" and "shares"). If source code is unavailable and the checkbox is not selected, parameters are displayed as param1, param2, and so forth. Also, if the checkbox is not selected the **Method Argument Names** dialog box (the next page of the EJB ETD Wizard) will not be displayed and the **Recursive Expansion of Member Objects** page opens. Click **Next** to continue.

13 The **Method Argument Names** page of the EJB ETD Wizard opens. For the **Source File Root** field, use the **File** and **Directories** buttons to locate and select the root directory or jar file that contains the EJB source files. As in step 8, specify the directory above the top-level Java package (see Figure 33). If the proper directory or jar file is selected, the corresponding Java files are found and the home and remote interfaces are displayed in the EJB Java Source Files field. Click **Next** to continue.

Figure 33 Method Argument Names - EJB ETD Wizard
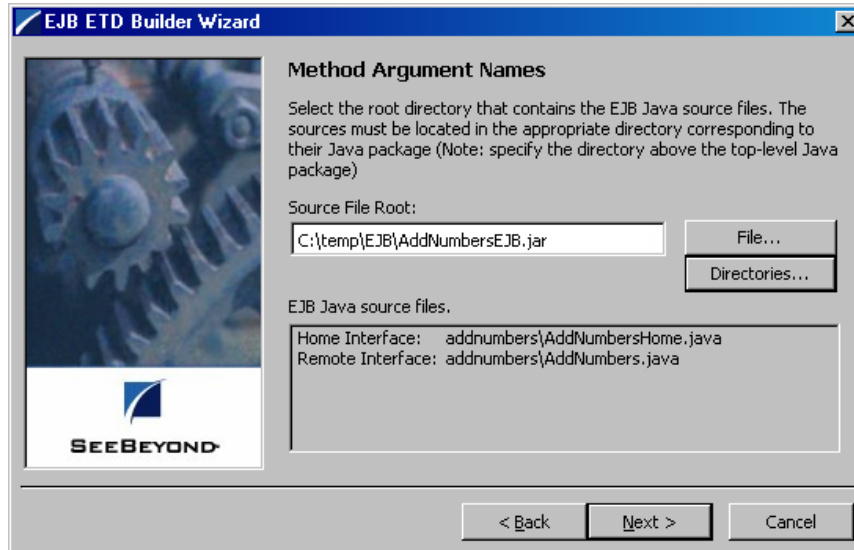
14 The **Recursive Expansion of Member Objects** page of the EJB ETD Wizard opens. Specify a parsing depth for the EJB ETD Builder. Select **Expand to:** and enter a specific depth. For example, entering **4** would expand nodes to expose the fields of classes referenced by the EJB to the forth level. Select **Expand All** to completely expand nodes to expose all referenced classes, or **Do Not Expand** for no node expansion. (See Figure 34)

Figure 34 Recursive Expansion of Member Objects - EJB ETD Wizard
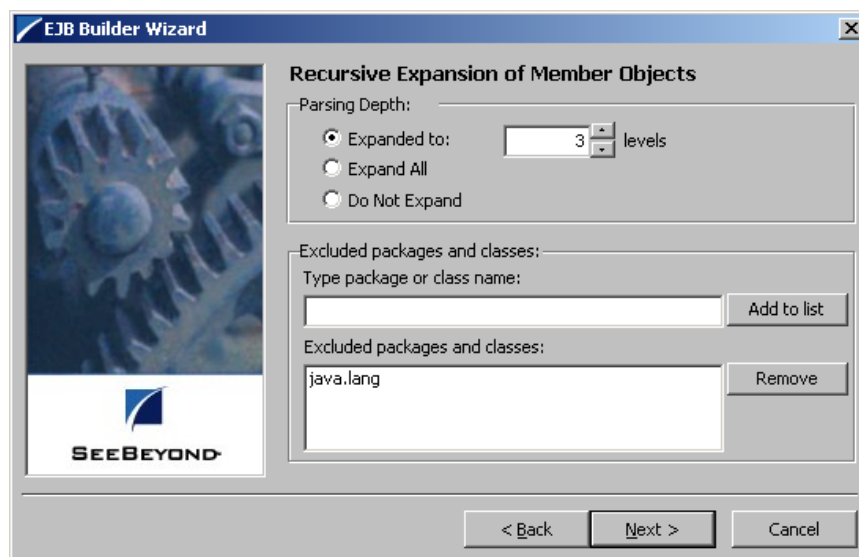
15 To exclude packages or classes from the generated ETD, such as a custom library referenced by the EJB, enter the package or class in the **Type package or class name:** field and click **Add to list**. The package or class is added to the **Excluded packages and classes** list. To remove a package or class from exclusion, select the item in the excluded list and click **Remove**. Click **Next** to continue

16 The **Classpath** page of the EJB ETD Builder Wizard opens. The Classpath dialog box allows the user to add any additional files to the classpath. The source root or jar file is added by default, but any additional classes referenced by the EJB can be added to the classpath by using **Add File** to locate and select a file or **Add Folder** and locating and selecting a folder. To remove an item from the classpath select the item and click **Remove** (see Figure 35). Click **Next** to continue.

Figure 35   Classpath - EJB ETD Wizard
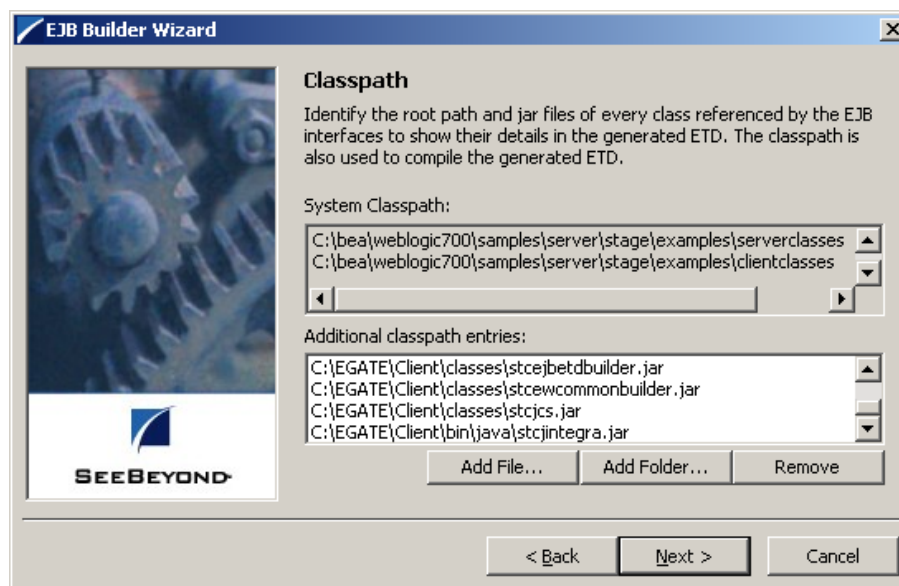
17 The **Completing the EJB Builder Wizard** page opens. Review all entries. Click **Back** to return to any fields that require changes. Click **Finish** to close the wizard and create the ETD.

18 The new EJB ETD opens in the ETD Editor (see Figure 36). The ETD created by the EJB ETD Builder has already been compiled. Save the ETD and promote to run time.

Figure 36 ETD Editor - EJB ETD
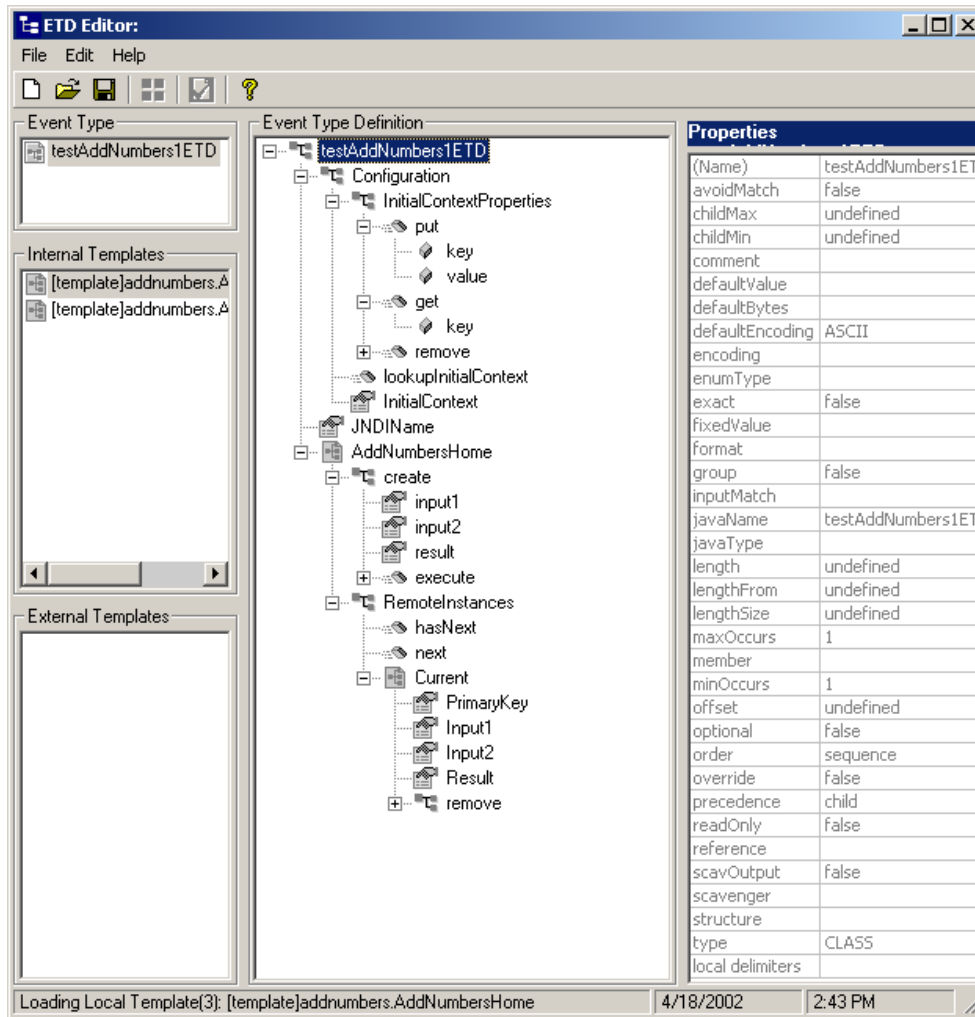
Step 2: Configure the e*Way Connection

e*Way Connection configuration parameters, using the weblogic.def file, facilitate communication between e*Gate and the JNDI directory service which connects e*Gate applications with objects in Session/Entity Beans that actually do the work.

The connection configuration is used to locate and access the JNDI directory service that contains the home interface of the EJB to be accessed. The EJB ETD then uses these settings to create an Initial Context to JNDI and looks up the JNDI name in the ETD to find the home interface. One connection configuration can be used with multiple EJB ETDs if they require the same settings for JNDI.

For directions on configuring the EJB ETD e*Way Connection see **"EJB ETD e*Way Connection" on page 50**. For more information on e*Way Connections and parameters see the e*Gate Integrator User's Guide. For information about creating or modifying any component within the e*Gate Schema Designer, see the e*Gate Schema Designer's online Help system.

Step 3: Build Collaboration Rules to instantiate the Remote Interfaces

The e*Gate user builds Collaborations between the EJB ETD and other ETDs, using the tools available in the Collaboration Rules Editor to call methods and their parameters to build query calls that return remote interfaces to carry out the required business logic (see **Creating the AddNumbers Sample Collaboration Rules** on page 89).

Step 4: Bind the e*Way Connection to the Collaboration Rules

The e*Gate user enters the subscription and publication instance name, Event Type, source and destination (specifying the e*Way Connection as either source or destination) in the Collaboration to bind the e*Way connection to the Collaboration Rule so that, at run time, the Collaboration knows how to find the JNDI directory service.

4.5 Asynchronous Interaction, WebLogic EJB to e*Gate JMS

Step 1: Configure WebLogic to create JNDI entries for SeeBeyond JMS on WebLogic Server at startup

Configure WebLogic to create JNDI entries in the directory service for SeeBeyond JMS on WebLogic Server instance startup (see **Configuring the WebLogic Server Components** on page 65).

Step 2: Create a new Session Bean from the template

Create an EJB that can publish to SeeBeyond JMS. Basic sample Session Beans, STCPublisherSLSession and STCQueueRequestorSLSession, are provided that, when instantiated, publish to the Queue name listed in their parameters. Users can use these samples as a models to build their own Session Beans.

Step 3: Create a new Deployment Descriptor from the template

An EJB is a Java class that can be written following the protocols of the application server. A deployment tool (an XML file similar to a configuration file for an e*Way) 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.

Step 4: Packaging and Deployment

Take the created Session Bean and the Deployment Descriptor and use the WebLogic GUI to make the EJB available for external applications to call it and publish to the SeeBeyond JMS.

4.6 Asynchronous Interaction, e*Gate JMS to WebLogic Message Driven Bean

Step 1: Configure WebLogic to create JNDI entries for SeeBeyond JMS on WebLogic Server at startup

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. The user installs these classes in the startup area of the Console and specifies the name of the properties file. (see **Configuring the WebLogic Server Components** on page 65).

Step 2: Create a new message driven bean from the template.

The user builds the EJB, implements business logic. Implementation uses JNDI to lookup TopicConnectionFactory.

Step 3: Create a new Deployment Descriptor from the template.

An EJB is a Java class that can be 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) which defines the session type (stateless, stateful), registers the Home and Remote classes with JNDI, and defines the JNDI name. The other side is the Application Server vendor-specific parameters (weblogic-ejb-jar.xml) which defines Pooling parameters and Reference Resource parameters.

Step 4: Packaging and deployment.

Take the Bean class files, Deployment Descriptors and place these in a Jar file. The Jar files are uploaded using the WebLogic Console and the EJB is deployed, making the class available to other applications.

4.7 Event Type Definitions

The Event Type Definition supplied for use with the Asynchronous Interaction samples is referred to as Blob.xsc. It resides in the JMSAsyncProducersConsumers, etd\.

Figure 37   Event Type Definition - Blob.xsc
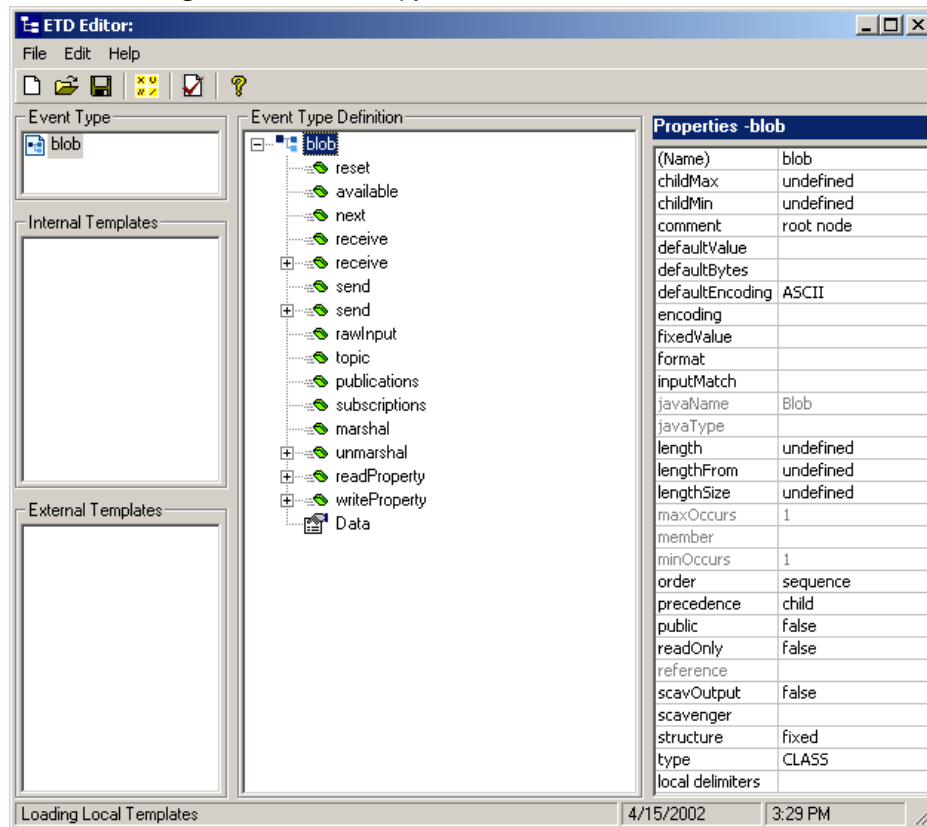
4.8 Creating the Sample Schemas

Sample schemas for the WebLogic e*Way synchronous (EJB ETD) and asynchronous (JMS) implementations are available in the ..\Samples\ewweblogic\ folder of the installation CD-ROM. Import the zip files into e*Gate to create the following schemas:

- **The AddNumbers Sample Schema (Synchronous, EJB ETD)** on page 85 demonstrates synchronous interaction (mode 1) in which an ETD is generated from a Session Bean's interface classes, that represents the methods of the EJB. The e*Way (feeder) triggers the EJB on WebLogic which runs the business logic (in this case, adds to numbers). The reply is then published to the JMS Queue, where it is picked up and published to a file.

- **The JMSAsynchProducersConsumers Sample Schema (Asynchronous, JMS)** on page 93 contains the following six samples that demonstrate the WebLogic e*Way's asynchronous interaction using the SeeBeyond JMS e*Way Connection. To install

the JMSAsynchProducersConsumers sample schema, import
JMSAsynchProducersConsumers.zip into the e*Gate Schema Designer.

- ◆ **The JMSQueueSend Sample** on page 94 demonstrates asynchronous
interaction (mode 3) from e*Gate to WebLogic via the SeeBeyond JMS Queue.
The e*Way picks up a message and publishes it to the SeeBeyond JMS Queue.
The message is then subscribed to by the WebLogic Message Driven Bean.
Sample input data, **JMSQueueSendQFIN.qfin**, is available in the WebLogic
samples directory on the Installation CD-ROM.

- ◆ **The JMSQueueRequestor Sample** on page 98 demonstrates asynchronous
interaction (modes 2 and 3) in which the e*Way receives a message from a
Queue and sends a reply back to the Session Bean which originated the
message.

- ◆ **The JMSXAQueueSend Sample** on page 102 demonstrates asynchronous
interaction (mode 3) similar to the JMSQueueSend Sample except with an XA
transaction. The e*Way picks up a message and publishes it to the SeeBeyond
JMS Queue. The message is then subscribed to by the WebLogic XA MDB.
Sample input data, **JMSXAQueueSendXAQFIN.xaqfin**, is available in the
WebLogic samples directory on the Installation CD-ROM.

- ◆ **The JMSTopicPublish Sample** on page 104 demonstrates asynchronous
interaction (mode 3) in which the e*Way picks up a message from a file and
publishes it to the SeeBeyond JMS Topic where the message is subscribed to by
the WebLogic MDB. Sample input data, **JMSTopicPublishTFIN.tfin**, is available
in the WebLogic samples directory on the Installation CD-ROM.

- ◆ **The JMSTopicSubscribe Sample** on page 108 demonstrates asynchronous
interaction (mode 2) in which the e*Way subscribes to a JMS Topic which is
published to by a WebLogic Session Bean.

- ◆ **The JMSXATopicSubscribe Sample** on page 112 demonstrates asynchronous
interaction (mode 3) similar to the JMSTopicSubscribe Sample except with an
XA transaction. The e*Way subscribes to an XA JMS Topic which is published to
by a WebLogic Session Bean.

4.8.1. Installing a Sample Schema

Import the schema at the startup of the e*Gate Schema Designer, or select "New
Schema" from the File menu of the e*Gate Enterprise. For either case, select "Create
from export:" and navigate to the zip file containing the necessary sample.

4.9 The AddNumbers Sample Schema (Synchronous, EJB ETD)

For the most part, these components are created when the sample schema is imported
into e*Gate. The following describes how those components were created and
configured. The AddNumbers sample demonstrates synchronous interaction, taking

the Session Bean, AddNumbersEJB.jar, and creating an ETD that represents the methods of the EJB. The sample takes two numbers, adds them, and returns the result.

The AddNumbersSchema sample is provided on the installation CD at ..\samples\ewweblogic\.

4.9.1. Running the AddNumbers Sample Schema

AddNumbersEJB is a Stateful Session EJB and must be deployed following the standard WebLogic deployment procedures using the AddNumbersEJB.jar file.

1 Import the schema, **AddNumbers.zip** and setup the input directory relative to your directory structure. The default is **C:\indata**

2 In the input directory create a file with the extension *.fin, and include any valid number (integer) in it and save the file.

3 Ensure that the Properties file for the Multi-Mode e*Way refers to the location of the **weblogic.jar file** and the **AddNumbersEJB.jar** file. Copy the **AddNumbersEJB.jar** file into the **bea\wlserver6.1\lib** directory for WebLogic Server 6.1 or into the **bea\weblogic700\server\lib** directory for WebLogic Server 7.0.

4 To recompile the Collaboration for this sample, make sure that the environmental classpath, as set through the Collaboration Editor, refers to the location of the weblogic.jar file.

5 Start the control broker and from the Monitor, start the Queue Manager first. Now start all the Modules and let the schema process the EJB.

6 If the schema ran successfully, an output0.dat file appears in the default output directory (C:\DATA) containing the sum of the input value plus 100.

Figure 38 AddNumbersSchema Sample Components
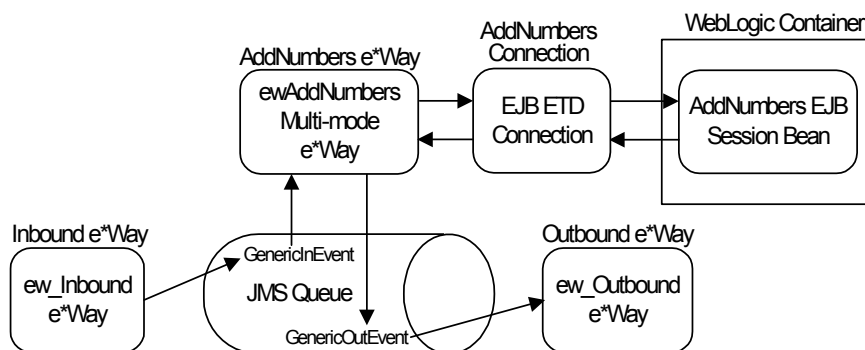
As seen in Figure 38, The Inbound e*Way reads the sample containing one number, and publishes to the JMS Queue. The AddNumbers e*Way subscribes to the JMS Queue, assigns the value from the sample to the input1 object, and triggers the EJB ETD Connection, sending a request to the AddNumbers EJB Session Bean on WebLogic Server. The EJB preforms the process, adding the values of input1 and input2 and returns the result. The AddNumbers e*Way gets the result and publishes it to the JMS Queue. The Outbound e*Way subscribes to the Queue and writes the result to file.

4.9.2. Configuring the AddNumbersSchema Sample

Once the sample has been successfully imported into e*Gate, the user must configure it to correspond to the system as necessary.

Copy and Deploy the Sample EJB

Copy the sample EJB, **AddNumbersEJB.jar** to an available temporary directory (for example C:\temp\EJB\AddNumbersEJB.jar). Open the WebLogic Console, go to Deployments\EJB and select **Install a new EJB**. From the **Install or Update an Application** page, use the **Browse** button to locate and select the Stateful Session Bean, **AddNumbersEJB.jar** on your system. Click on Upload to upload the .jar.

Note: The e*Gate JMS server must be started before the deployment of EJBs using SeeBeyond JMS to prevent the risk of message loss.

Configure STCWLStartup.properties

STCWLStartup.properties must be configured to match the localhost SeeBeyond JMS. Go to <WL HOME>\lib and open STCWLStartup.properties to edit the JMS host and port number to match that of the SeeBeyond JMS server.

Create and Configure the e*Ways

The AddNumbers sample schema contains three e*Ways, two of which are pass-through (ew_Inbound and ew_Outbound) and one multi-mode (ewAddNumbers).

Configuring the Pass-Through e*Ways.

The pass-through e*Ways, **ew_Inbound** and **ew_Outbound** use the executable file "**stcewfile**", set in the e*Way's properties (See Figure 39). For each of the e*Ways, go to the **Start Up** tab of the properties file, and select **Start automatically**.

Configuration files for the e*Ways can be saved as default except for the following: For **ew_Outbound** the General Settings must be set to AllowIncoming: **NO**, AllowOutgoing: **YES**, and for the Outbound (send) settings, set OutputFileName to **Result%d.dat**. When configuration is complete, save the Configuration files and promote to run time.

Figure 39   e*Way Properties - Pass-through
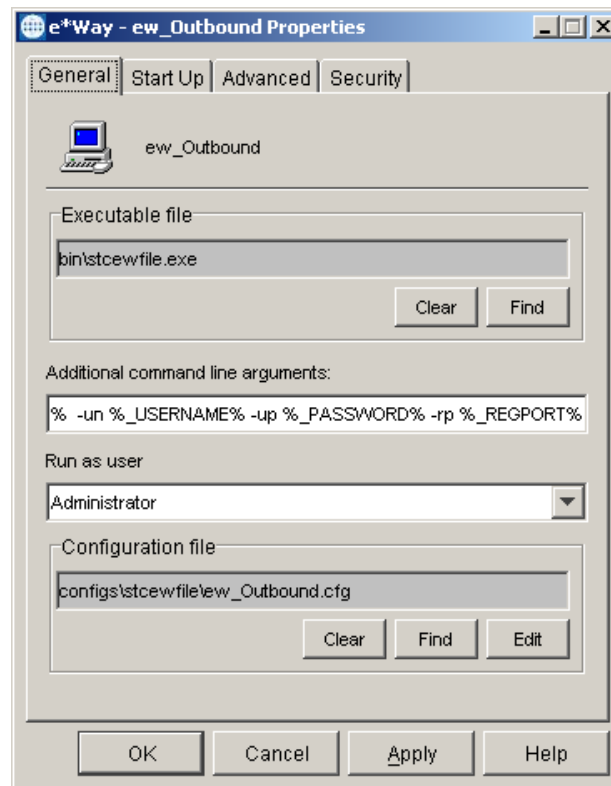
Configuring the Multi-Mode e*Way

The Multi-mode e*Way, **ewAddNumbers**, uses the executable file "**stceway**", set in the e*Way's properties. Also, go to the **Start Up** tab, and select **Start automatically**.

The Configuration file for the Multi-mode e*Way can be saved as default except for the following setting:

The JVM Settings **CLASSPATH Prepend** parameter must include (append) **AddNumbers.jar** and **weblogic.jar**.

When configuration is complete, **Save** the file and select **Promote to Run Time**.

For more information on the Multi-Mode e*Way configuration settings see the *e*Gate Integrator User's Guide*.

Create the ETD

To create the ETD using the EJB ETD Builder follow the directions in **Step 1: Build the ETD from the interface classes** on page 76.

Configure the Queue Manager

Open the IQ Manager Properties and select SeeBeyond JMS for the IQ Manager Type. Click New for the Configuration file, save the default file and promote to run time.

Create the e*Way Connections

One e*Way Connection is created for the AddNumbers sample.

The **AddNumbersConnection**, e*Way Connection Type is **EJB ETD**. Click **New** under the e*Way Connection Configuration File field. The **e*Way Template Selection** dialog box opens. Select the e*Way template for **WebLogic**. The Configuration file for the AddNumbersConnection e*Way Connection can be saved as default except for the JNDI InitialContext Setting/java.naming.provider.url, for which the user must specify the WebLogic .url, (for example: t3://localhost:7003). Save the file and promote to run time.

For more information on the EJB ETD e*Way Connection configuration parameters see **Configuring the ETD e*Way Connection** on page 51.

Creating the AddNumbers Sample Collaboration Rules

The cr_JavaPassThru Collaboration Rules Properties appear as follows when complete (see Figure 40). The Figure displays both the General and the Collaboration Mapping tabs.

Figure 40 Collaboration Rules Properties - Java_collabrule
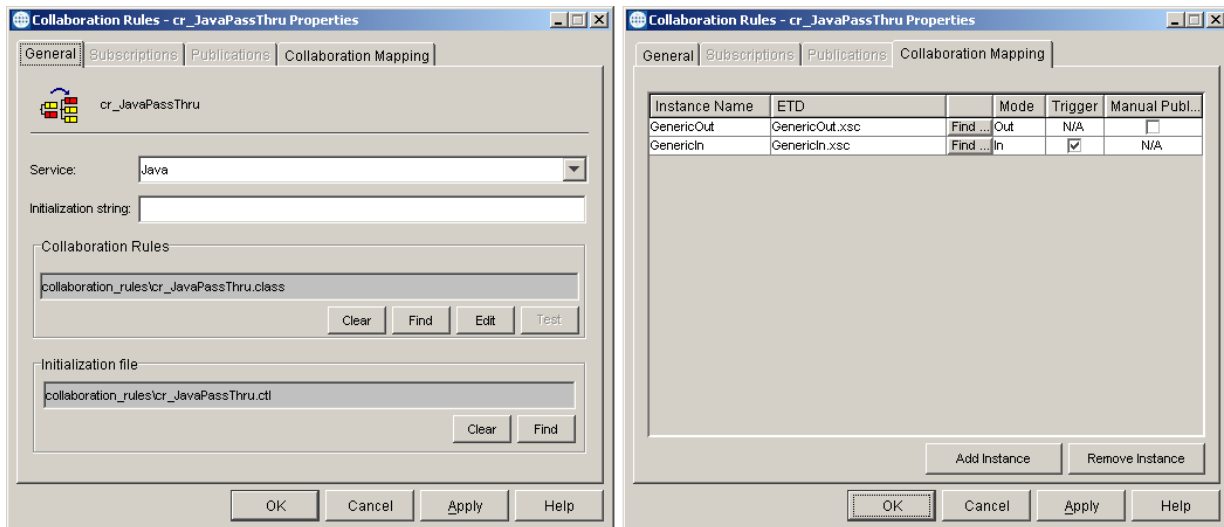
The cr_**AddNumbers** Collaboration Rules Properties dialog box appear as follows (see Figure 41):

Figure 41 Collaboration Rules Properties - cr_AddNumbers
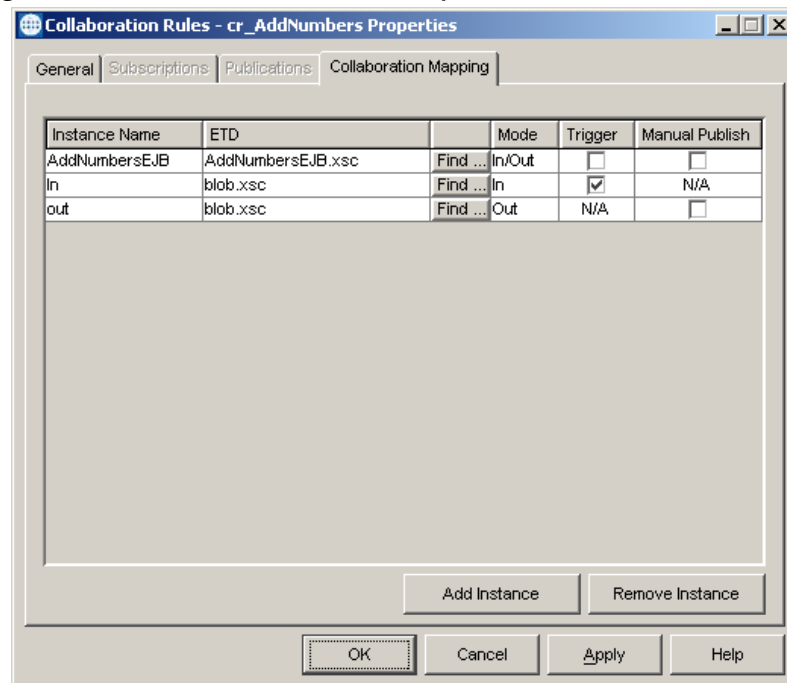
From the General tab of the cr_AddNumbers Collaboration Rules Properties dialog box, click **Edit** or **New** under the Collaboration Rules field. The Collaboration Rules Editor opens.

Creating the Business Rules Using the Collaboration Rules Editor

Each **rule** is created by clicking the rule button on the Business Rules toolbar. For more information on using the Java Collaboration Rules Editor, see the *e*Gate Integrator User's Guide.*

The cr_**AddNumbers** Collaboration Rules (see Figure 42) are created as follows:

1   First, on the Menu bar, select **Tools**, **Options**, and **append the classpath**.

2   The first **rule**, under retBoolean in the Business Rules window, is created by "dragging and dropping" **blobField** under AddNumbersEJB, In [BlobRoot], on the Source Events command node to **input1** under AddNumbersEJB, AddNumbersHome, create, on the Destination Events command node.

3   To create the second **rule** drag **input2** under AddNumbersEJB, AddNumbersHome, create, on the Destination Events command node to the Rule Properties, Rule window and set the parameter for setinput2 as 1000.

4   For the third **rule**, drag the **execute** method under AddNumbersEJB, AddNumbersHome, create on the Destination Events command node to the Rule Properties, Rule window.

5   The **if** expression is created by clicking the **if** button on the Business Rules toolbar, then dragging the **hasNext** method under AddNumbersEJB, AddNumbersHome, RemoteInstances in the Source Events command node to the Rule Properties, Rule window.

Figure 42   .Collaboration Rules Editor - AddNumbers
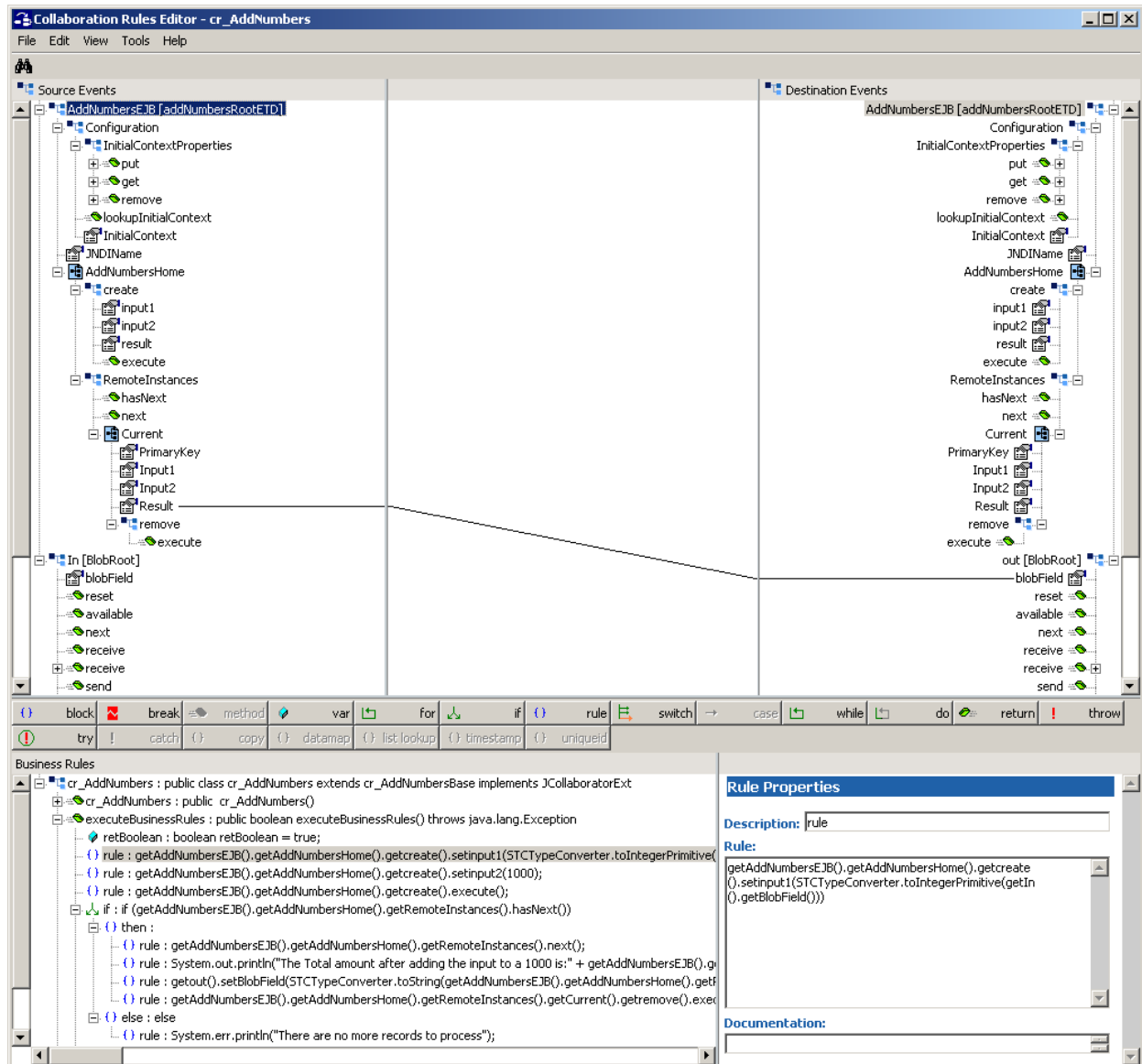
6 For the next **rule**, highlight (select) the **then** expression under **if**, and click on the **rule** button. Drag the **hasNext** method under AddNumbersEJB, AddNumbersHome, RemoteInstances on the Source Events command node to the Rule Properties, Rule window.

7 The next **rule** is created by typing the following in the Rule Properties, Rule window:

```
System.out.println("The Total amount after adding the input to a 1000 is:" +)
```

the drag **Result** under AddNumbersEJB, AddNumbersHome, RemoteInstances, Current, on the Source Events command node into the parenthesis following the +.

8 The next **rule** is created by dragging **Result** under AddNumbersEJB, AddNumbersHome, RemoteInstances, Current, on the Source Events command node to the Rule Properties, Rule window.

9 The next **rule** is created by dragging and dropping **Result**, under AddNumbersEJB, AddNumbersHome, RemoteInstances, Current, on the Source Events command node to the **blobField** under AddNumbersEJB, Out [BlobRoot], on the Destination Events command node.

10 The next **rule** is created by dragging and dropping the **execute** method, under AddNumbersEJB, AddNumbersHome, RemoteInstances, Current, remove, on the Source Events command node to the Rule Properties, Rule window.

11 The last **rule** is created by selecting **else**, clicking the **rule** button, and typing the following in the Rule Properties, Rule window:

```
System.err.println("There are no more records to process")
```

12 When the business logic is complete, save and compile.

Creating the Collaborations

The Collaborations for the ew_Inbound and ew_Outbound e*Ways named collab_in and collab_out appear as follows when complete (see Figure 43):

Figure 43 Collaboration Properties - ew_Inbound and ew_Outbound
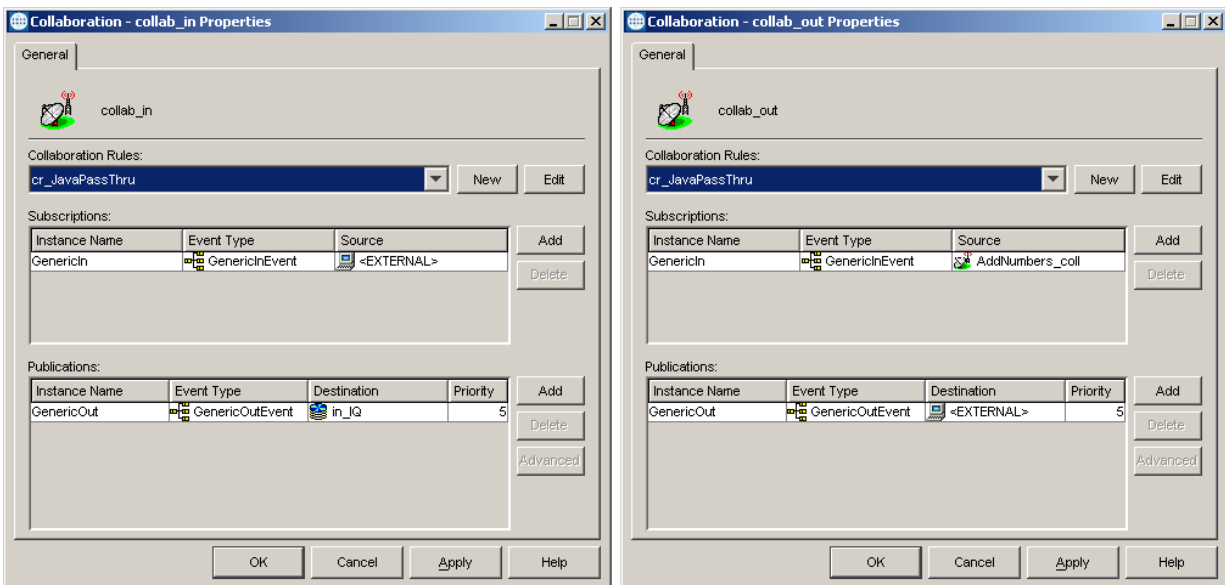
The AddNumbers Collaboration named collab_AddNumbers appears as follow when complete (see Figure 44):

Figure 44   Collaboration Properties - collab_AddNumbers
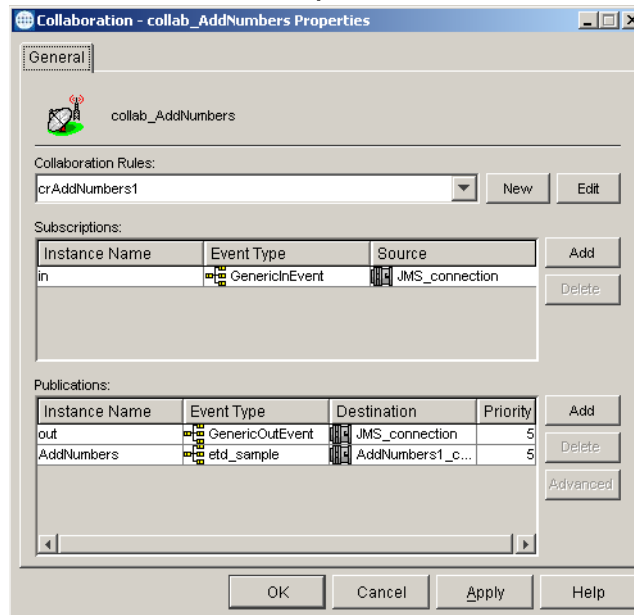
4.10 The JMSAsynchProducersConsumers Sample Schema (Asynchronous, JMS)

The JMSAsynchProducersConsumers sample schema contains six e*Ways configured to utilize the SeeBeyond JMS e*Way Connection to deliver and receive message to and from the Enterprise JavaBeans running inside the WebLogic container. The schema also configures the IQ Manager as a SeeBeyond JMS IQ Manager. Sample EJBs, included in **stcejbweblogic.jar**, are deployed using the configured SeeBeyond JMS IQ Manager. There are essentially two modes of operations: e*Ways sending or publishing messages to a Queue or Topic, and e*Ways which receive or subscribe to a Queue or a Topic.

4.10.1. Running the JMSAsynchProducersConsumers Schema

When running the JMSAsynchProducersConsumers Schema containing the six asynchronous JMS samples do the following:

1   For directions on importing the sample see **Installing a Sample Schema** on page 85. The default STCWLStartup.properties file as shipped for the e*Way do not need to be modified for the samples to work. **ejb.jar** and **weblogic.jar** (with ejb.jar preceding weblogic.jar in order) must be added to the system classpath for the ETD to be generated successfully.

2   Make sure that the sample schema is running first prior to deploying the EJBs. This ensures that the SeeBeyond IQ Manager (SeeBeyond JMS Server) is available so that the WebLogic container can create the connections on behalf of the MDBs during deployment.

3 Do NOT feed messages into the feeder e*Ways UNTIL the sample EJBs are deployed. This guarantees that there are subscriber or receiver MDBs running before messages are sent to Topics or Queues.

4 Start the WebLogic "examples" server in a console using the startup script.

5 Deploy the sample EJBs (**stcejbweblogic.jar**).

6 For message flow from **e*Gate to WebLogic**, feed messages to the feeder e*Ways. Messages are seen on the WebLogic console. For message flow from **WebLogic to e*Gate**, use the EJB sample clients to feed messages to the EJBs. Messages from the eater e*Ways are written to files.

Note: *For the STCQueueRequestorSLSessionBean sample, messages are displayed on the sample remote client console. See* **The JMSQueueRequestor Sample** *on page 98 for details.*

4.10.2. The JMSQueueSend Sample

In this sample, the JMSQueueSend e*Way (stcewfile.exe) acts as a feeder of messages to the Queue.Sample1 Queue. The JMSQueueSend e*Way looks for files with extension ".qfin" as input files (the input directory configured is c:\InputData). The colJMSQueueSend Collaboration subscribes to external (for an event from a file) and publishes to the conJMSQueueProducer JMS e*Way Connection. The conJMSQueueProducer JMS e*Way Connection is configured to use the internal SeeBeyond JMS IQ Manager as the JMS "server." The colJMSQueueSend Collaboration uses the crJMSQueueSend Collaboration Rule which copies data from the source event to the output event. The STCReceiverMDBean MDB receives messages from the Queue.Sample1 Queue and display the message it receives to the WebLogic console.

Figure 45 JMSQueueSend Sample Components
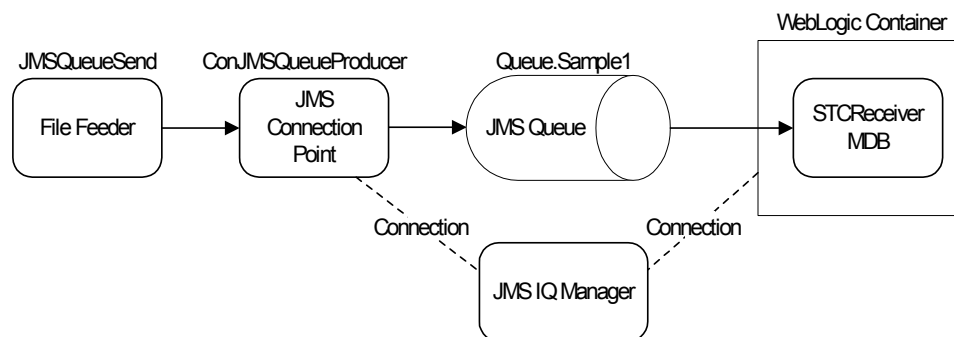
As seen in Figure 45, the File Feeder reads a file containing the input message event. A feeder Collaboration subscribes from external and publishes the input message, as a Queue.Sample1 event, to the JMS e*Way Connection. The JMS e*Way Connection is configured to use a JMS Queue and acts as a QueueSender. Both the JMS e*Way Connection 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 17.) The STCRecieverMDBean MDB receives the method that is passed from the container, and displays the message in standard out (the WebLogic console).

Configuring the JMSQueueSend Sample

Once the sample has been successfully imported into e*Gate, the user must configure it to correspond to the system as necessary. The following items should be examined.

- Each of the configuration files associated with the e*Way must be configured as needed, saved, and promoted to run time. Specifically, the following parameters must be addressed as shown in Table 2:

Table 2   e*Way Configuration Parameters - JMSQueueSend

e*Way Configuration Parameters	
General Settings - Set as directed, otherwise leave as default.	
AllowIncoming	YES
AllowOutgoing	NO
PerformanceTesting	NO
Outbound (send) settings - Set as directed, otherwise leave as default.	
OutputDirectory	C:\DATA
OutputFileName	output%d.dat
MultipleRecordsPerFile	YES
MaxRecordsPerFile	10000
AddEOL	Yes
Poller (inbound) settings - Set as directed, otherwise leave as default.	
PollDirectory	C:\INDATA
InputFileMask	*.qfin
PollMilliseconds	1000
RemoveEOL	YES
MultipleRecordsPerFile	NO
MaxBytesPerLine	4096
BytesPerLineIsFixed	NO
File Records Per eGate Event	1
Performance Testing - Set as directed, otherwise leave as default.	
Performance Testing	100
InboundDuplicates	1

- The conJMSQueueProducer e*Way Connection parameters associated with the JMSQueueSend sample appear as shown in Table 3:

Table 3   e*Way Connection Parameters - JMSQueueSend

e*Way Connection Parameters	
General Settings - Set as directed, otherwise leave as default.	
Connection Type	Queue
Transaction Type	Internal
SDelivery Mode	Persistent

Table 3   e*Way Connection Parameters - JMSQueueSend

e*Way Connection Parameters	
Maximum Number of Bytes to read	10000000
Default Outgoing Message Type	Text
Message Selector	
Factory Class Name	com.stc.common.collabService.SBYNJMSFactory
Message Service - Set as directed, otherwise leave as default.	
Server Name	localhost_iqmgr
Host Name	localhost
Port Number	24053
Maximum Message Cache Size	100

For more information on e*Way Connection Configuration Parameters for JMS see **Configuring the JMS e*Way Connection parameters** on page 61

The JMSQueueSend Collaboration Rules Script

The crJMSQueueSend Collaboration Rules Script appears as follows (see Figure 46):

Figure 46   Collaboration Rules Script - crJMSQueueSend
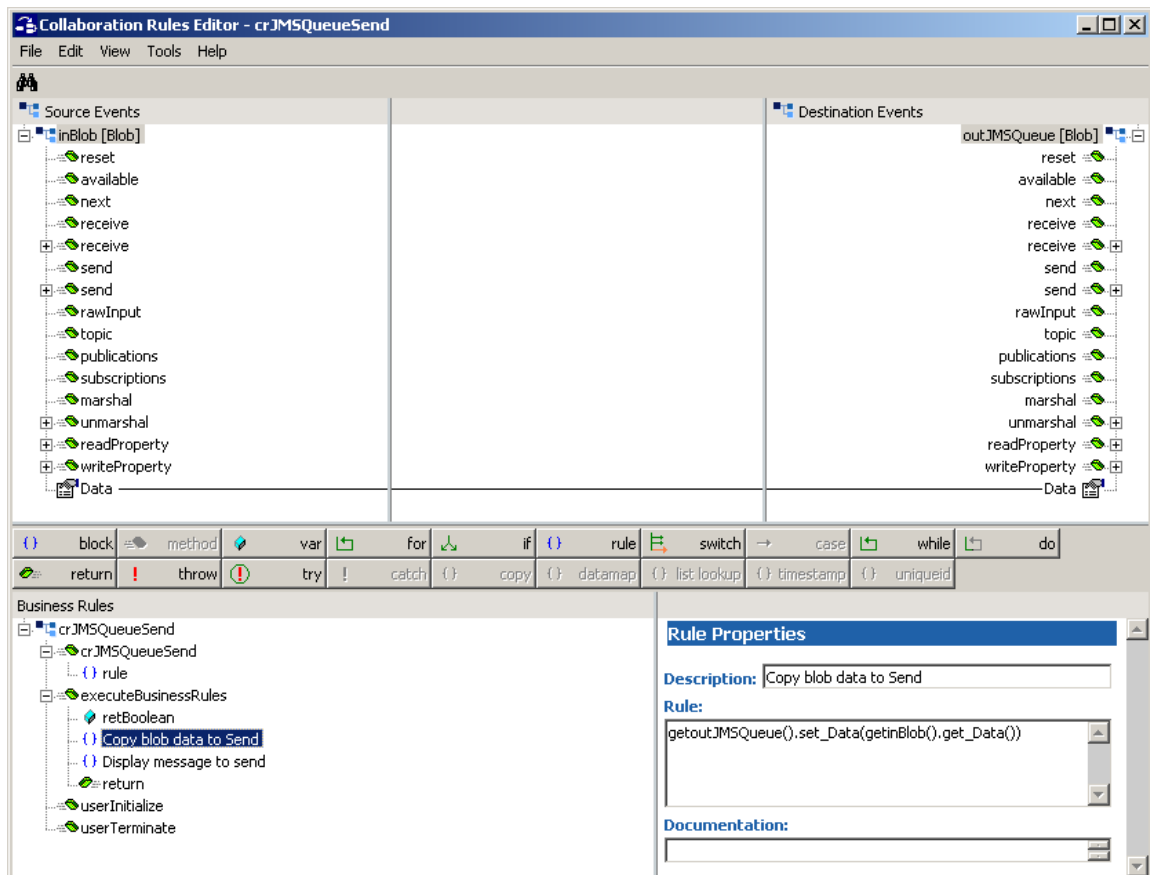
Each new rule is created by clicking the **rule** button on the Business Rules toolbar. For additional information on using the Java Collaboration Rules Editor, see the *e*Gate Integrator User's Guide*. The crJMSQueueSend business rules are created as follows:

1 "**Copy blob data to Send**" is created by dragging **Data** located under Source Events command node and dropping it on **Data** located under the Destination Events.

2 "**Display message to send**" is created by dragging **Data** located under Source Events command node into the Rule Properties, Rules window and entering code before and after to create the following code:

```
System.out.println("\nSending Message:\n*****Start of Message*****\n" + getinBlob().get_Data()
+ "\n*****End of Message*****\n")
```

JMSQueueSend Collaboration Rule Mapping

The Collaboration Mapping associated with the crJMSQueueSend Collaboration Rule appears as follows (see Figure 47):

Figure 47 crJMSQueueSend - Collaboration Mapping
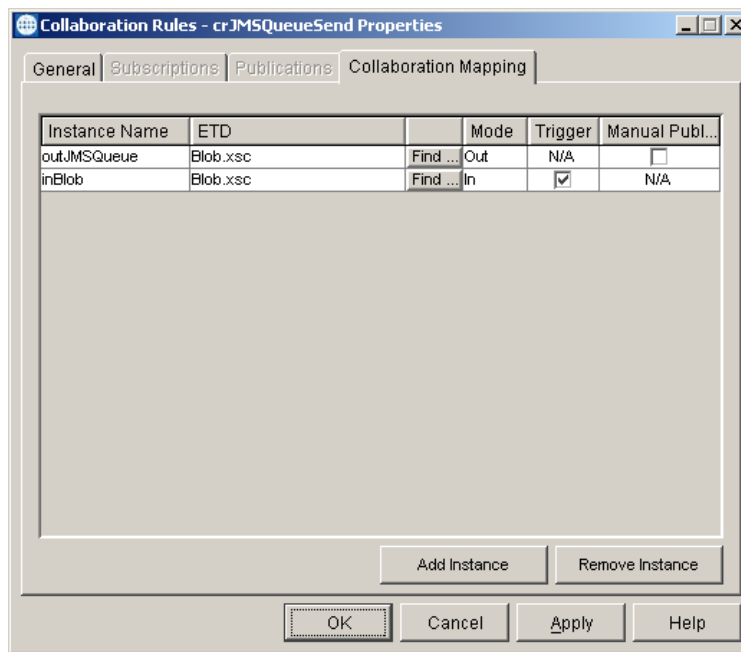
JMSQueueSend Collaboration Properties

The colJMSQueueSend Collaboration Properties for the JMSQueueSend sample appears as follows (see Figure 48):

Figure 48   colJMSQueueSend - Collaboration Properties
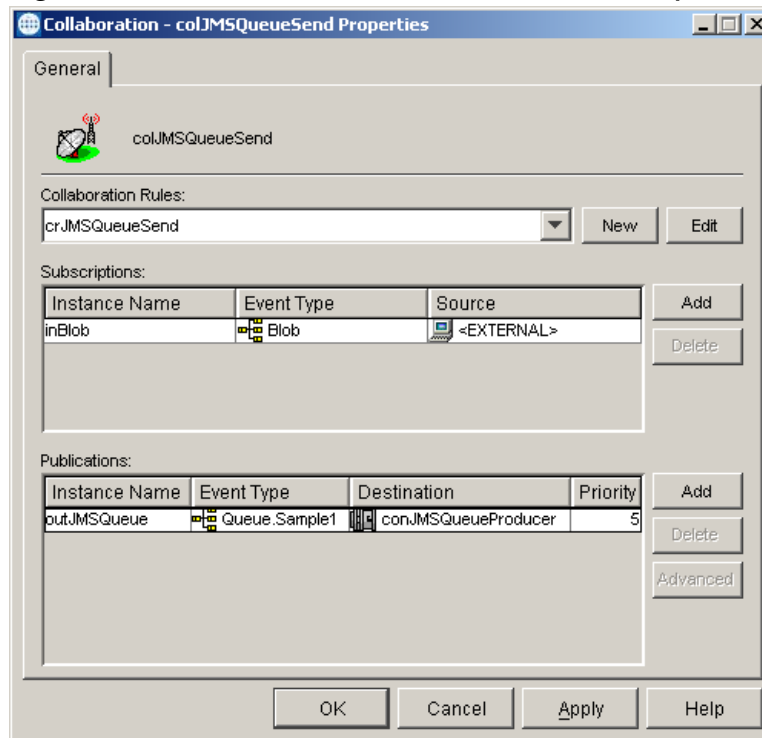

4.10.3. The JMSQueueRequestor Sample

In this sample, the JMSQueueRequestor e*Way (stceway.exe) acts as a receiver of
messages to the Queue.Sample2 Queue. The colJMSQueueRequestor Collaboration
subscribes to the conJMSQueueRequestor JMS e*Way Connection on the
Queue.Sample2 Queue and manually publishes back to the conJMSQueueRequestor
JMS e*Way Connection. The conJMSQueueRequestor JMS e*Way Connection is
configured to use the internal SeeBeyond JMS IQ Manager as the JMS "server." The
colJMSQueueSend uses the crJMSQueueRequestor Collaboration Rule which simply
constructs a reply string, by prepending the String "e*Gate got message:" to the
message it received from the Queue and manually publishing 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 e*Gate.
Essentially, this demonstrates a request/reply usage of the QueueRequestor JMS object
by the STCQueueRequestorSLSessionBean.

Figure 49 JMSQueueRequestor Sample Components
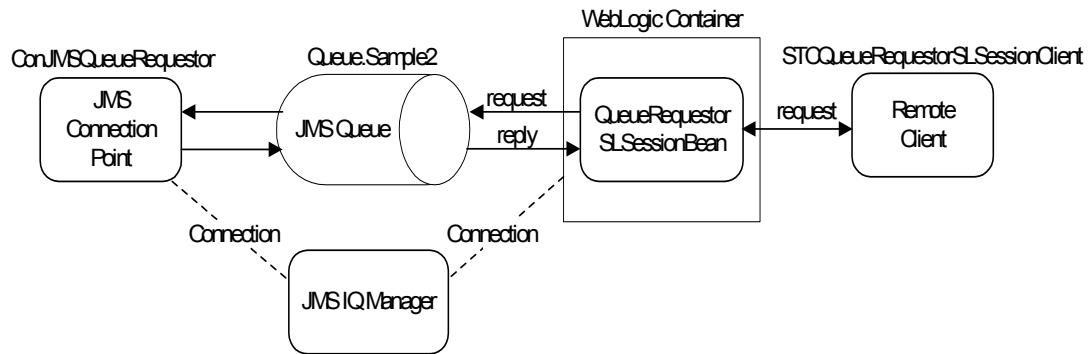
As seen in Figure 49, The stand-alone remote client,
com.stc.eways.ejb.sessionbean.queuerequestor.STCQueueRequestorSLSessionClient, is
used to invoke the request() method of the STCQueueRequestorSLSessionBean and
wait for a reply from the Session Bean. As parameters, the client takes the provider
URL 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 (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:7003 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.queuerequestor.STCQueueRequestorSLSessionClient t3://
localhost:7003 SeeBeyond.STCQueueRequestorSLSessionBean c:\temp\testfile.txt file
```

Configuring the JMSQueueRequestor Sample

Once the sample has been successfully imported into e*Gate, the user must configure it
to correspond to the information as necessary. Each of the configuration files associated
with the e*Way must be configured as needed, saved, and promoted to runtime.

- The conJMSQueueRequestor e*Way Connection Configuration settings are the
  same as those in **Table 3 on page 95**.

- The JMSQueueRequestor Multi-mode e*Way uses the default configuration
  parameters.

For more information on e*Way Connection Configuration Parameters for JMS see
Configuring the JMS e*Way Connection parameters on page 61.

JMSQueueRequestor Collaboration Rule

The crJMSQueueRequestor Collaboration Rule appears as follows (see Figure 50). For
this example, "Display Code" under View on the menubar has been enabled so that the
Java code is displayed in the Business Rules window.

Figure 50   Collaboration Rules - crJMSQueueRequestor
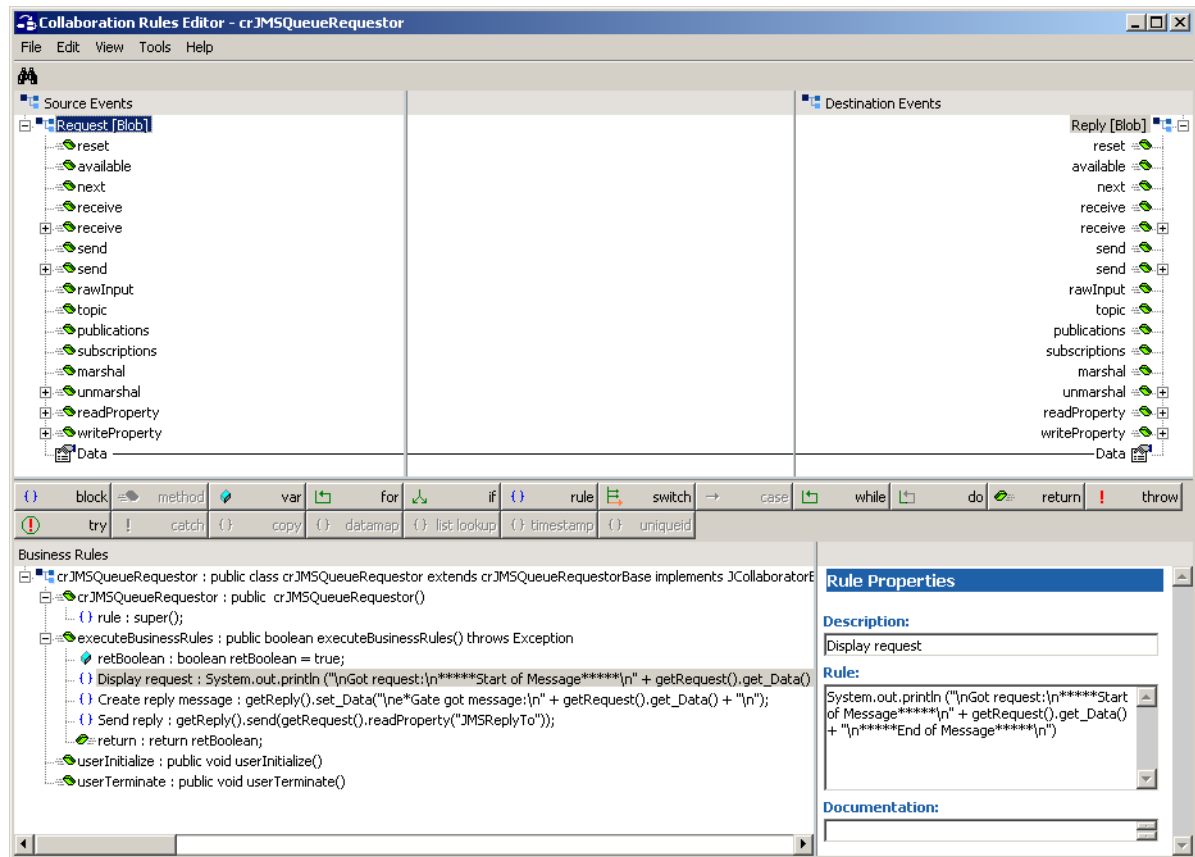
Each new rule is created by clicking the **rule** button on the Business Rules toolbar. For additional information on using the Java Collaboration Rules Editor, see the *e*Gate Integrator User's Guide*. The crJMSQueueRequestor business rules are created as follows:

1 "**Display request**" is created by dragging Data located under Source Events command node and dropping it on Data located under the Destination Events.

2 "**Create reply message**" is created dragging **Data** located under Source Events command node into the Rule Properties, Rules window and entering code before and after to create the following code:

```
System.out.println ("\nGot request:\n*****Start of Message*****\n" + getRequest().get_Data() +
"\n*****End of Message*****\n")
```

3 "**Send reply**" is created by Dragging send under Reply located under the Destination Events command node into the Rule Properties, Rules window. Drag **propName** located under Reply, readProperty under the Destination Events command node into the properties for send (the last set of parenthesis) in the Rules window. Enter JMSReplyTo as the parameter for the readProperty() propName.

JMSQueueRequestor Collaboration Rule Mapping

The Collaboration Mapping associated with the crJMSQueueRequestor Collaboration Rule appears as follows (see Figure 51):

Figure 51 crJMSQueueRequestor - Collaboration Mapping
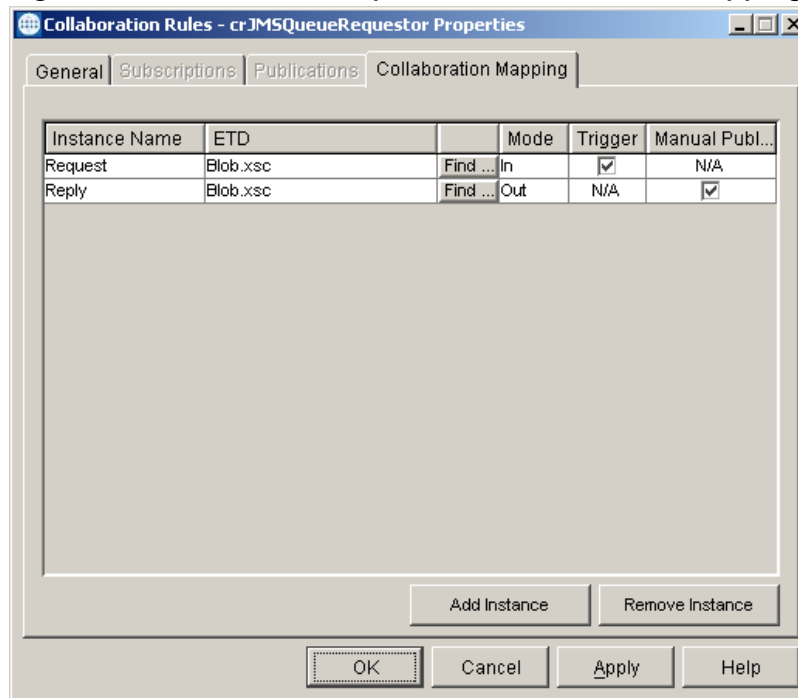
JMSQueueRequestor Collaboration Properties

The colJMSQueueRequestor Collaboration for the JMSQueueSend sample appears as follows (see Figure 52):

Figure 52 colJMSQueueRequestor - Collaboration Properties
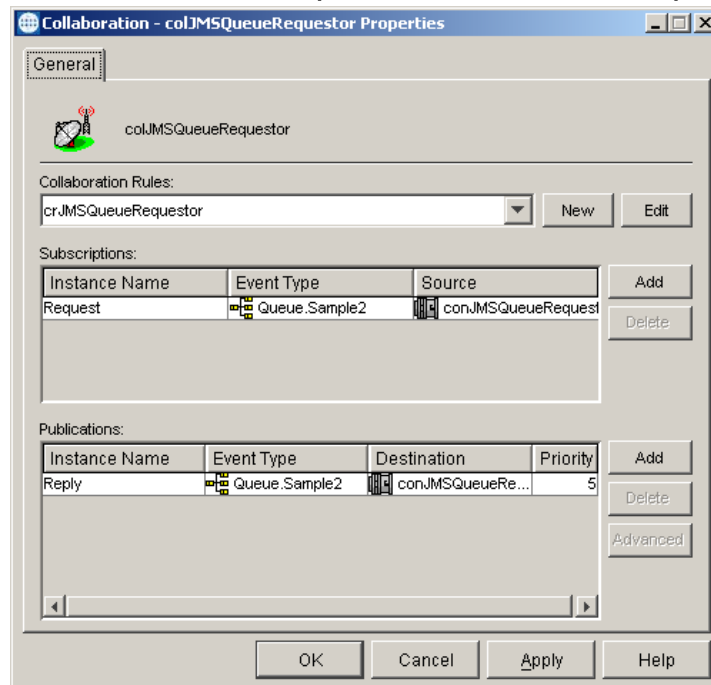
4.10.4. The JMSXAQueueSend Sample

In this sample, the JMSXAQueueSend e*Way (stcewfile.exe) acts as a feeder of messages to the Queue.Sample3 Queue. The JMSXAQueueSend e*Way looks for files with the extension .xaqfin as input files (the input directory configured is c:\InputData). The colJMSXAQueueSend Collaboration subscribes to external (for an event from a file) and publishes to the conJMSXAQueueProducer JMS e*Way Connection. The conJMSXAQueueProducer JMS e*Way Connection is configured to use the internal SeeBeyond JMS IQ Manager as the JMS server. The colJMSXAQueueSend Collaboration uses the crJMSQueueSend Collaboration Rule which copies data from the source event to the output event.

Figure 53   JMSXAQueueSend Sample Components
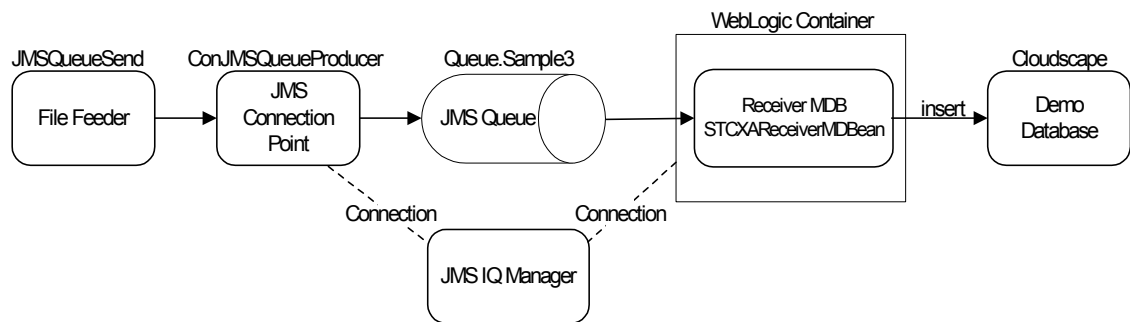
As seen in Figure 53, the File Feeder reads a file containing the input message event. A feeder Collaboration subscribes from external and publishes the input message to the JMS e*Way Connection as a Queue.Sample3 event. The JMS e*Way Connection is configured to use a JMS Queue and therefore acts as a QueueSender. Both the JMS e*Way Connection 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 17.) 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 Cloudscape sample demoXAPool to receive messages from SeeBeyond JMS and write database records into the sample Cloudscape database table. Checking the database to see that the record is there does not necessarily confirm that a two phase commit has occurred.

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, to see XA at work, see **Verifying XA At Work** on page 41. 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 43. For details on the format of the input message for the feeder e*Way see **SeeBeyond Sample XA Message Driven Beans** on page 36.

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

The result of the test is that e*Gate 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 e*Gate.

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

Configuring the JMSXAQueueSend Sample

Once the sample has been successfully imported into e*Gate, the user must configure it to correspond to the information as necessary. The following items should be examined

- The JMSXAQueueSend **e*Way Connection Configuration settings** are the same as those in **Table 3 on page 95**.

- Configuration parameters for the **JMSXAQueueProducer** e*Way Connection used with the JMSXAQueueSend sample are the same as those in **Table 2 on page 95** with the exception of the following:

Table 4 e*Way Configuration Parameters - JMSXAQueueProducer

e*Way Configuration Parameters	
General Settings - See Table 2 on page 95.	
Outbound (send) settings - See Table 2 on page 95.	
Poller (inbound) settings - Set parameters as directed, otherwise see Table 2 on page 95.	
InputFileMask	*.xaqfin
Performance Testing - See Table 2 on page 95.	

- For more information on e*Way Connection Configuration Parameters for JMS see **Configuring the JMS e*Way Connection parameters** on page 61.

The JMSXAQueueSend Collaboration Rule

The JMSXAQueueSend sample uses the JMSQueueSend Collaboration Rule (see **Collaboration Rules Script - crJMSQueueSend** on page 96).

JMSXAQueueSend Collaboration Rule Mapping

The JMSXAQueueSend sample uses the crJMSQueueSend Collaboration Rule Mapping (see **crJMSQueueSend - Collaboration Mapping** on page 97).

JMSXAQueueSend Collaboration Properties

The colJMSXAQueueSend Collaboration for the JMSXAQueueSend sample appears as follows (see Figure 54):

Figure 54 colJMSXAQueueSend - Collaboration Properties
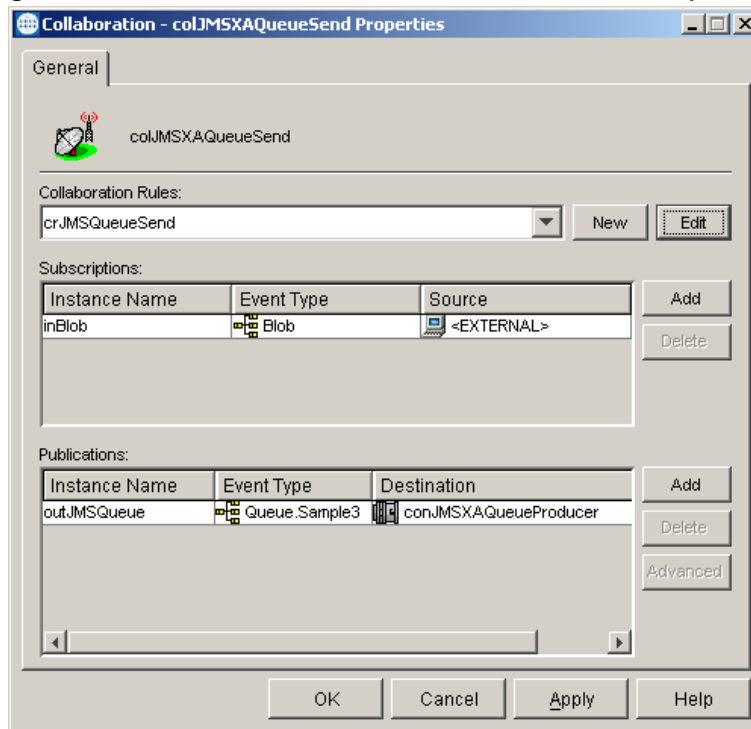
4.10.5. The JMSTopicPublish Sample

In this sample, the JMSTopicPublish e*Way (stcewfile.exe) acts as a feeder of messages to the Topic.Sample1 Topic. The JMSTopicPublish e*Way looks for files with the extension .tfin as input files (the configured input directory is c:\InputData). The colJMSTopicPublish Collaboration subscribes to external (for an event from a file) and publishes to the conJMSTopicProducer JMS e*Way Connection. The conJMSTopicProducer JMS e*Way Connection is configured to use the internal SeeBeyond JMS IQ Manager as the JMS server. The colJMSTopicPublish Collaboration uses the crJMSTopicPublish Collaboration Rule which simply copies data from the source event to the output event. The STCSubscriberMDBean MDB receives messages from the Topic.Sample1 Topic and displays the message it receives to the WebLogic console.

Figure 55 JMSTopicPublish Sample Components
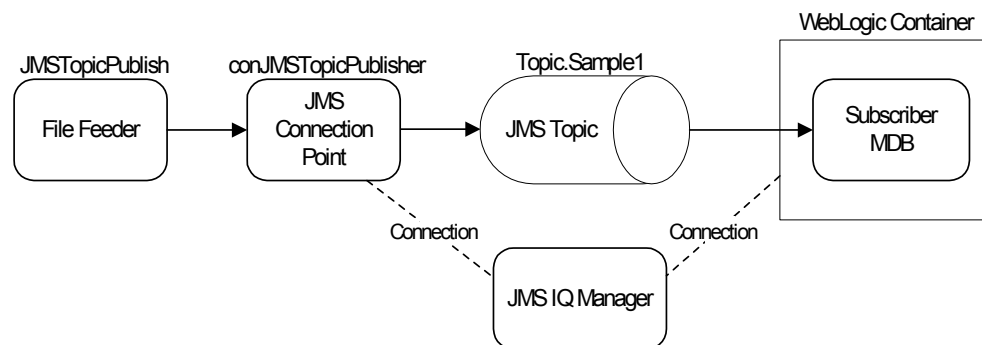
As seen in Figure 55, the File Feeder reads a file containing the input message event. A feeder Collaboration subscribes from external and publishes the input message, as a Topic.Sample1 event, to the JMS e*Way Connection. The JMS e*Way Connection is configured to use a JMS Topic, acting as a TopicPublisher. Both the JMS e*Way Connection 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 17. The STCSubscriberMDBean MDB receives the message, passed to it by the container, and displays the message in standard out (the WebLogic console).

Configuring the JMSTopicPublish Sample

Once the sample has been successfully imported into e*Gate, the user must configure it to correspond to the information as necessary. The following items should be examined

▪ Configuration parameters for the **conJMSTopicProducer** e*Way Connection used with the JMSTopicPublish sample are the same as those in **Table 2 on page 95** with the exception of the following:

Table 5 e*Way Configuration Parameters - JMSTopicPublish

e*Way Configuration Parameters	
General Settings - See Table 2 on page 95.	
Outbound (send) settings - See Table 2 on page 95.	
Poller (inbound) settings - Set as directed, otherwise see Table 2 on page 95.	
InputFileMask	*.tfin
Performance Testing - See Table 2 on page 95.	

▪ The conJMSTopicProducer e*Way Connection Parameters associated with the JMSTopicPublish sample appear as shown in Table 6:

Table 6 e*Way Connection Parameters - conJMSTopicProducer

e*Way Connection Parameters	
General Settings - Set as directed, otherwise leave as default.	
Connection Type	Topic
Transaction Type	Internal
SDelivery Mode	Persistent
Maximum Number of Bytes to read	5000
Default Outgoing Message Type	Text
Message Selector	
Factory Class Name	com.stc.common.collabService.SBYNJMSFactory
Message Service - Set as directed, otherwise leave as default.	
Server Name	localhost_iqmgr
Host Name	localhost
Port Number	24053
Maximum Message Cache Size	100

For more information on e*Way Connection Configuration Parameters for JMS see **Configuring the JMS e*Way Connection parameters** on page 61

The crJMSTopicPublish Collaboration Rule

The crJMSTopicPublish Collaboration Rule appears as follows (see Figure 56):

Figure 56 Collaboration Rules - crJMSTopicPublish
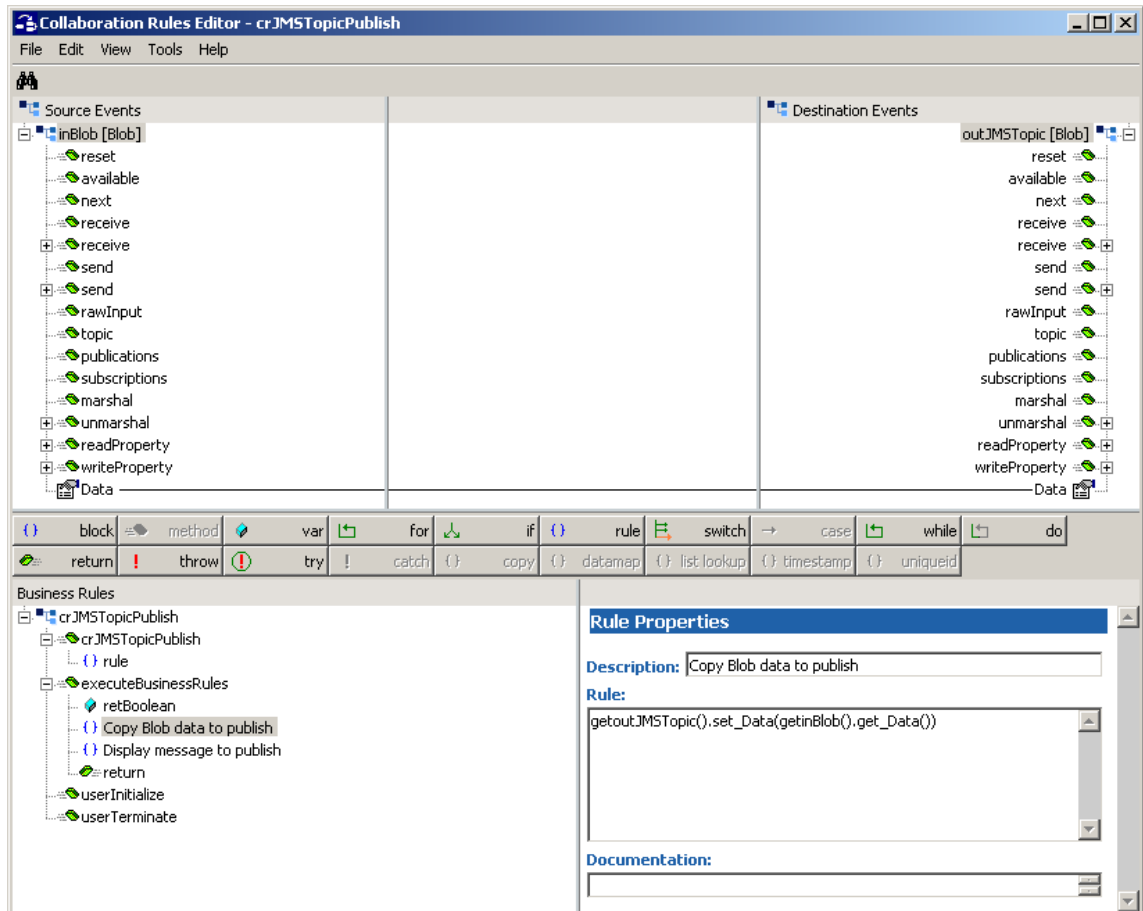
Each new rule is created by clicking the **rule** button on the Business Rules toolbar. For additional information on using the Java Collaboration Rules Editor, see the *e*Gate Integrator User's Guide*. The crJMSTopic Publish business rules are created as follows:

1 "**Copy blob data to Publish**" is created by dragging Data located under Source Events command node and dropping it on Data located under the Destination Events.

2 "**Display message to Publish**" is created by dragging **Data** located under Source Events command node into the Rule Properties, Rules window and entering code before and after to create the following code:

```
System.out.println("\nMessage to Publish:\n*****Start of Message*****\n" +
getinBlob().get_Data() + "\n*****End of Message*****\n")
```

JMSTopicPublish Collaboration Rule Mapping

The Collaboration Mapping associated with the crJMSTopicPublish Collaboration Rule appears as follows (see Figure 57):

Figure 57 crJMSTopicPublish - Collaboration Map
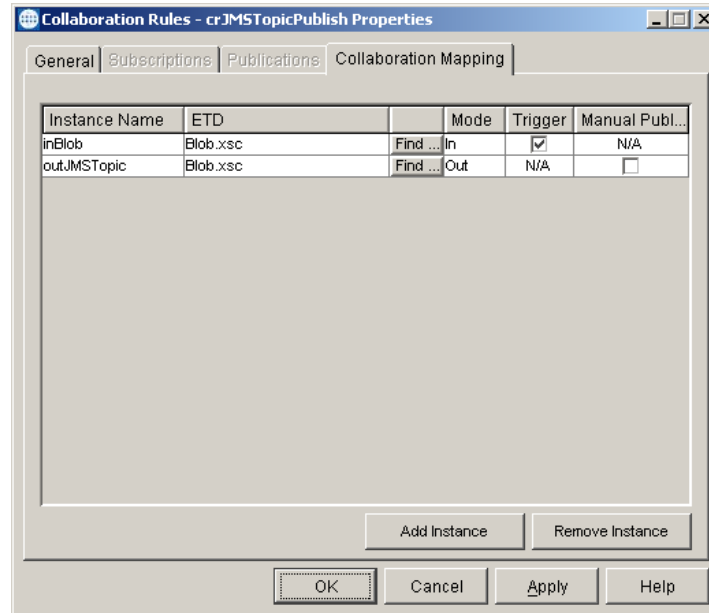
JMSTopicPublish Collaboration Properties

The colJMSTopicPublish Collaboration for the JMSQueueSend sample appears as follows (see Figure 58):

Figure 58 colJMSTopicPublish - Collaboration Properties
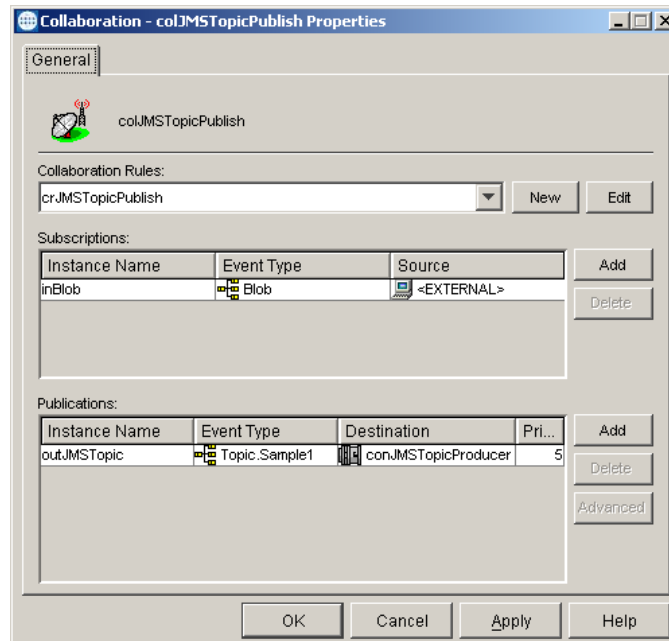
4.10.6. The JMSTopicSubscribe Sample

In this sample, the JMSTopicSubscriber e*Way (stcewfile.exe) acts as a eater of messages coming from the Topic.Sample2 Topic. The colJMSTopicSubscribe Collaboration subscribes to the conJMSTopicConsumer JMS e*Way Connection on the Topic.Sample2 Topic. The conJMSTopicConsumer JMS e*Way Connection is configured to use the internal SeeBeyond JMS IQ Manager as the JMS server. The colJMSTopicSubscribe uses the crJMSTopicSubscribe Collaboration Rule, which displays the message received to standard output, and publishes the message to the external (writes the message received to a file).

In this case, the STCPublisherSLSessionBean Session Bean acts as publisher to the Topic.Sample2 Topic. Essentially, this demonstrates publishing messages asynchronously from an EJB running in WebLogic to a SeeBeyond JMS Topic.

Figure 59 JMSTopicSubscribe Sample Components
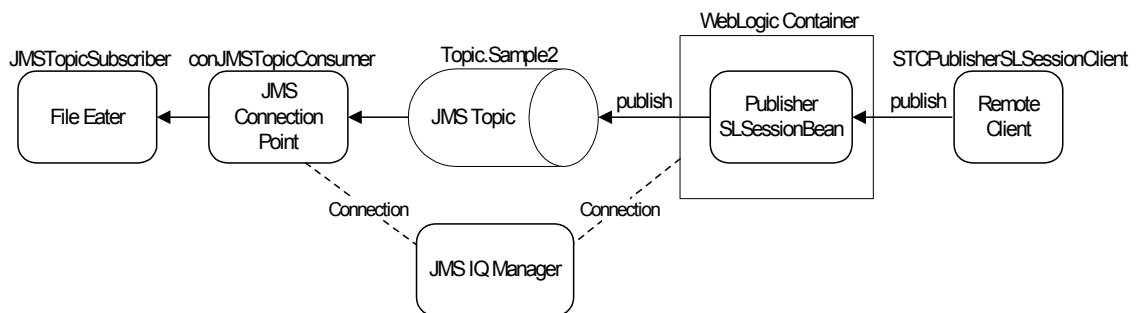
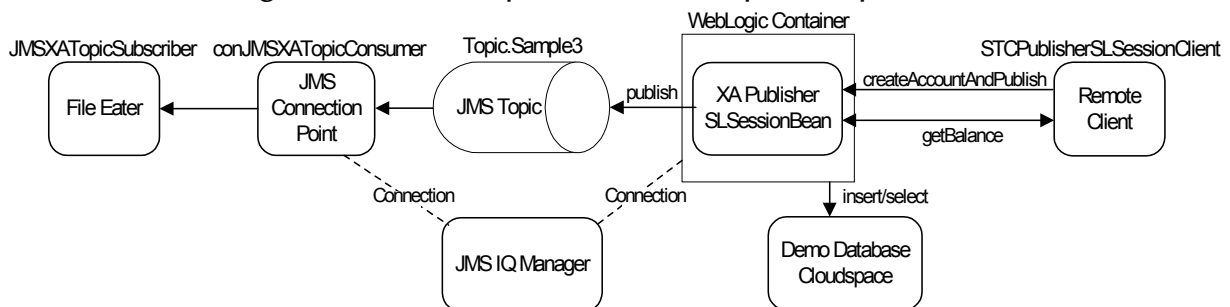The stand-alone remote client, com.stc.eways.ejb.sessionbean.publisher.STCPublisherSLSessionClient, can be used to invoke the publish() method of the STCPublisherSLSessionBean to send a message to e*Gate asynchronously. The parameters taken by the client are: the provider URL of the WebLogic JNDI where the Session Bean is bound, the JNDI name of the Session Bean (SeeBeyond.STCPublisherSLSessionBean), a text message or a file name, and the option specifying whether the third parameter is a file (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:7003
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:7003
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 preceding weblogic.jar in order), and stcejbweblogic.jar.*

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

Configuring the JMSTopicSubscribe Sample

Once the sample has been successfully imported into e*Gate, the user must configure it to correspond to the information as necessary. The following items should be examined

▪ Parameters for the JMSTopicSubscribe e*Way configuration used with the JMSTopicSubscribe sample appear as shown in Table 7:

Table 7 e*Way Configuration Parameters - ewJMSTopicSubscribe

e*Way Configuration Parameters	
General Settings - Set as directed, otherwise leave as default.	
AllowIncoming	NO
AllowOutgoing	YES
PerformanceTesting	NO
Outbound (send) settings - Set as directed, otherwise leave as default.	
OutputDirectory	C:\DATA
OutputFileName	topicrecv%d.dat
MultipleRecordsPerFile	NO
MaxRecordsPerFile	10000
AddEOL	Yes
Poller (inbound) settings - Set as directed, otherwise leave as default.	
PollDirectory	C:\INDATA
InputFileMask	*.fin
PollMilliseconds	1000
RemoveEOL	YES
MultipleRecordsPerFile	YES
MaxBytesPerLine	4096
BytesPerLineIsFixed	NO
File Records Per eGate Event	1
Performance Testing - Set as directed, otherwise leave as default.	
Performance Testing	100
InboundDuplicates	1

▪ Configuration parameters for the **conJMSTopicConsumer** e*Way Connection used with the JMSTopicSubscribe sample appear as shown in Table 8:

Table 8 e*Way Connection Parameters - conJMSTopicConsumer

e*Way Connection Parameters	
General Settings - Set as directed, otherwise leave as default.	
Connection Type	Topic
Transaction Type	Internal
SDelivery Mode	Persistent
Maximum Number of Bytes to read	10000000

Table 8   e*Way Connection Parameters - conJMSTopicConsumer

e*Way Connection Parameters	
Default Outgoing Message Type	Bytes
Message Selector	
Factory Class Name	com.stc.common.collabService.SBYNJMSFactory
Message Service - Set as directed, otherwise leave as default.	
Server Name	localhost_iqmgr
Host Name	localhost
Port Number	24053
Maximum Message Cache Size	100

For more information on e*Way Connection Configuration Parameters for JMS see **Configuring the JMS e*Way Connection parameters** on page 61

The JMSTopicSubscribe Collaboration Rule

The crJMSTopicSubscribe Collaboration Rule appears as follows (see Figure 60):

Figure 60   Collaboration Rules - crJMSTopicSubscribe
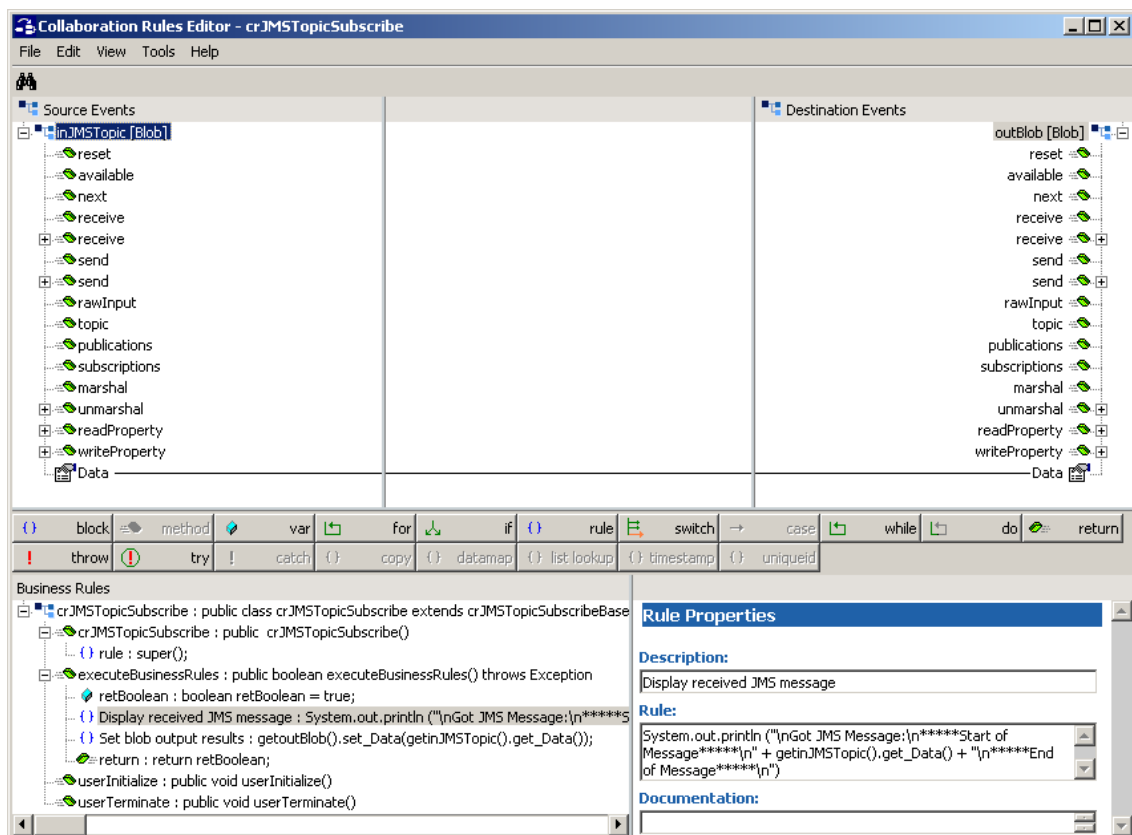
Each new rule is created by clicking the **rule** button on the Business Rules toolbar. For additional information on using the Java Collaboration Rules Editor, see the *e*Gate Integrator User's Guide*.

The crJMSTopicSubscribe business rules are created as follows:

1 "**Display received JMS message**" is created by dragging **Data** located under Source Events command node into the Rule Properties, Rules window and entering code before and after to create the following code:

```
System.out.println ("\nGot JMS Message:\n*****Start of
Message*****\n" + getinJMSTopic().get_Data() + "\n*****End of
Message*****\n")
```

2 "**Set blob output results**" is created by dragging Data located under Source Events command node and dropping it on Data located under the Destination Events.

JMSTopicSubscribe Collaboration Rule Mapping

The Collaboration Mapping associated with the crJMSTopicSubscribe Collaboration Rule appears as follows (see Figure 61):

Figure 61   crJMSTopicSubscribe - Collaboration Map
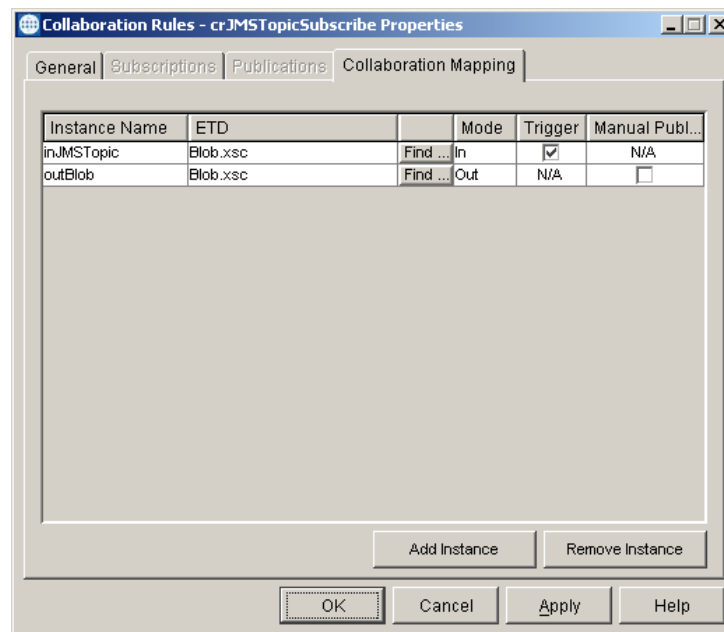
JMSTopicSubscribe Collaboration Properties

The colJMSTopicSubscribe Collaboration for the JMSTopicSubscribe sample appears as follows (see Figure 62):

Figure 62   colJMSTopicSubscribe - Collaboration Properties
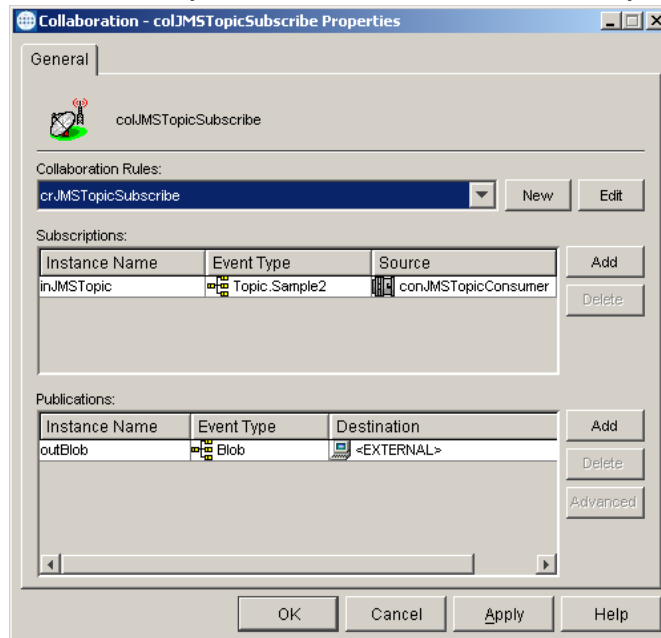
4.10.7. The JMSXATopicSubscribe Sample

In this sample, the JMSXATopicSubscriber e*Way (stcewfile.exe) acts as an eater of messages coming from the Topic.Sample3 Topic. The colJMSXATopicSubscribe Collaboration subscribes to the conJMSXATopicConsumer JMS e*Way Connection on the Topic.Sample3 Topic. The conJMSXATopicConsumer JMS e*Way Connection is configured to use the internal SeeBeyond JMS IQ Manager as the JMS server. The colJMSXATopicSubscribe uses the crJMSTopicSubscribe Collaboration rule which displays the message received to standard output, and publishes the message to the external (writes the message received to a file).

In this case, the STCXAPublisherSLSessionBean Session Bean acts as publisher to the Topic.Sample3 Topic. Essentially, this demonstrates publishing messages asynchronously from an EJB running in WebLogic to a SeeBeyond JMS Topic transactionally.

Figure 63   JMSXATopicSubscribe Sample Components

The stand-alone remote client,
com.stc.eways.ejb.sessionbean.xapublisher.STCPublisherSLSessionClient, can be used

to invoke the **createAccountAndPublish()** method of the STCXAPublisherSLSessionBean. This method takes two parameters: an account ID of type java.lang.String and 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 has successfully been 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, and 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:7003
SeeBeyond.STCXAPublisherSLSessionBean JohnDoe 8888.99
```

Note: *Before running this client, make sure that the system classpath includes ejb.jar, weblogic.jar (with ejb.jar preceding 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 to confirm that the record has indeed been inserted successfully. Note that getBalance does NOT confirm that a two phase commit has occurred. To see that both the database and SeeBeyond JMS XA Resources have been used, look at the weblogic.log and SeeBeyond JMS IQ Manager log. In addition, upon successfully publishing to the Topic, the file eater e*Way 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 43. For details on the format of the input message for the feeder e*Way see **SeeBeyond Sample XA Message Driven Beans** on page 36.

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

Configuring the JMSXATopicSubscribe Sample

Once the sample has been successfully imported into e*Gate, the user must configure it to correspond to the information as necessary. The following items should be examined

▪ Parameters for **ewJMSXATopicSubscribe** e*Way configuration used with the JMSXATopicSubscribe are the same as those in **Table 7 on page 109** with the exception of the following parameters:

Table 9   e*Way Configuration Parameters - ewJMSXATopicSubscribe

e*Way Configuration Parameters	
General Settings - See Table 7 on page 109.	
Outbound (send) settings - Set as directed, otherwise see Table 7 on page 109.	
OutputFileName	topicrecv%d.dat

Table 9   e*Way Configuration Parameters - ewJMSXATopicSubscribe

e*Way Configuration Parameters
Poller (inbound) settings - See Table 7 on page 109.
Performance Testing - See Table 7 on page 109.

- Configuration parameters for the **conJMSXATopicConsumer** e*Way Connection used with the JMSXATopicSubscribe sample Are the same as those that appear in **Table 7 on page 109**.

For more information on e*Way Connection Configuration Parameters for JMS see **Configuring the JMS e*Way Connection parameters** on page 61.

The JMSXATopicSubscribe Collaboration Rule

The JMSXATopicSubscribe Sample uses the crJMSTopicSubscribe Collaboration Rule (see **Table 60 on page 110**).

JMSXATopicSubscribe Collaboration Rule Mapping

The JMSXATopicSubscribe sample uses the crJMSTopicSubscribe Collaboration Mapping (see **Table 61 on page 111**).

JMSXATopicSubscribe Collaboration Properties

The colJMSTopicSubscribe Collaboration for the JMSXATopicSubscribe sample appears as follows (see Figure 64):

Figure 64   colJMSXATopicSubscribe - Collaboration Properties
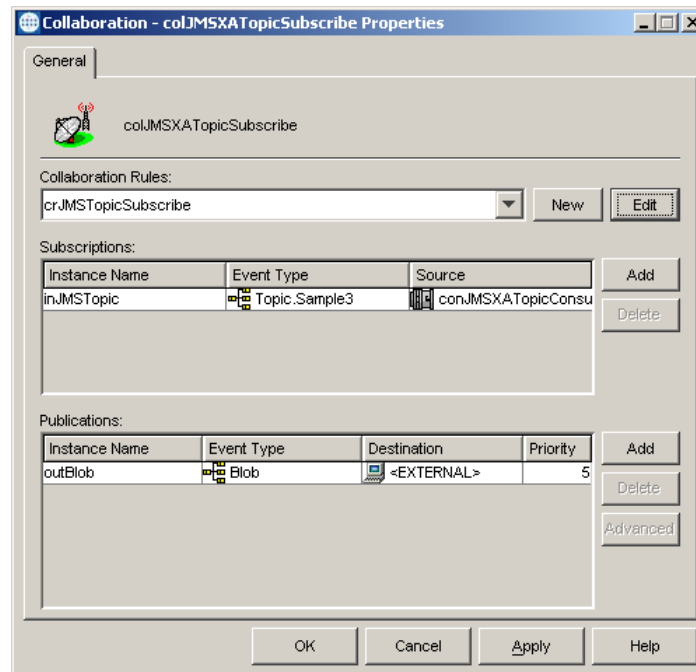
4.11 **Executing the Schema**

To execute a schema, do the following:

1 Go to the command line prompt, and enter the following:

```
stccb -rh hostname -rs schemaname -un username -up user password
-ln hostname_cb
```

Substitute *hostname*, **schemaname**, *username*, and *user password* as appropriate.

2 Start the Schema Manager. Specify the server that contains the Control Broker you started in Step 1 above.

3 Select the schema.

4 Verify that the Control Broker is connected. To do this, select and right-click the Control Broker in the Schema Manager, and select **Status**. (The message in the Control tab of the console will indicate command *succeeded* and status as *up*.)

5 Select the IQ Manager, *hostname*_igmgr, then right-click and select **Start**. (This will already be started if **Start automatically** is selected in the IQ Manager properties.)

6 Select each of the e*Ways, right-click select **Start**. (These will already be started if **Start automatically** is selected in the e*Way's properties.)

7 To view the output, copy the output file (specified in the Outbound e*Way configuration file). Save to a convenient location and open.

Java Methods

The WebLogic e*Way's available Java methods fall into the following class.

5.1 The EJBConfiguration Class

The EJBConfiguration class provides implementation for the functionality exposed in the EJB ETD e*Way configuration node, such as settings configured in the e*Way Connection.

```
java.lang.Object
    com.stc.ejbetd.EJBConfiguration
```

public final class **EJBConfiguration**

Extends java.lang.Object.

Methods of the EJBConfiguration Class

These methods are described in detail on the following pages:

getInitialContext

Description

Gets the JNDI Initial Context. Looks up the initial context 'on demand' if there is no initial context set yet.

Syntax

```
public javax.naming.InitialContext getInitialContext()
```

Parameters

None

Return Values

javax.naming.InitialContext
The JNDI initial context.

Throws

javax.naming.NamingException

getInitialContextProperties

Description

Gets the JNDI Initial Context Properties. These are used in to look up the initial context with the configured JNDI provider. The default properties are read in from the connection configuration. They can be overridden/added to in the Collaboration.

Syntax

```
public java.util.Hashtable getInitialContextProperties()
```

Parameters

None.

Return Values

java.util.Hashtable

Throws

None.

lookupInitialContext

Description

Perform a lookup of the JNDI initial context, using the properties set in the Connection Configuration or overridden/set in the InitialContextProperties node. Assigns the resulting initial context to the initial context field. This method is executed automatically 'on demand' upon the first EJB Home interface method call. If the InitialContextProperties are changed sub- sequently, lookupInitialContext has to be called manually to make use of the new configuration.

Syntax

```
public void lookupInitialContext()
```

Parameters

None.

Return Values

None.

Throws

javax.naming.NamingException

setInitialContext

Description

Set the Initial Context to a user obtained instance of it.

Syntax

```
public void setInitialContext(javax.naming.InitialContext
anInitialContext)
```

Parameters

Name	Type	Description
anInitialContext	`javax.naming.Init ialContext`	the user obtained instance of an initialContext

Return Values

None.

Throws

None.

setInitialContextProperties

Description

Set the Initial Context Properties used with lookupInitialContext.

Syntax

```
public void setInitialContextProperties(java.util.Hashtable
initialContextProperties)
```

Parameters

Name	Type	Description
initialContextProperties	`java.util.Hashtable`	The new properties to use with the next call of lookupInitialContext.

Return Values

None.

Throws

None.

Overriding the JNDI Name

The JNDIName node, for which the default is specified in the Wizard, is generated into the ETD. This setting can be overridden in the Collaboration. If no default is set by the user, it must be set in the Collaboration before accessing any of the methods in the EJB Home interface.

The default JNDI name generated by the wizard can be overridden by setting the JNDIName node in the Collaboration Rule. Similarly, the settings to contact the JNDI provider are read in from the connection configuration and made available in the Configuration/InitialContextProperties node so that further details can be added to this Hashtable programatically (for example, …getInitialContextProperties().put("myproperty", "myvalue")). The **lookupinitialcontext()** method only has to be called explicitly if a valid initialcontext is already in place and the user wants to switch to another JNDI provider dynamically in the middle of a Collaboration Rule. Otherwise, upon first access to the home interface or InitialContext node, the initial context is automatically created with all the details provided in the InitialContextProperties at that time. The InitialContext node is a context to the JNDI provider and can be used to access other entries or functionality in JNDI.

Index