WebLogic Server Components

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 820–5892
June 2008

081219@21808

Contents

1 WebLogic Server Components 5

WebLogic Server Components Overview 5

WebLogic Server Components Task Overview 5

Java Naming and Directory Interface (JNDI) 6

The WebLogic Naming Service 6

Sample Code 7

Viewing the WebLogic JNDI Tree 8

Java Messaging Service (JMS) 8

Enterprise JavaBeans (EJBs) 10

Session Beans 11

Entity Beans 11

Message Driven Beans 11

XA Transactions 12

CHAPTER 1

WebLogic Server Components

The following sections provide instructions on how to work with WebLogic Server Components. If you have any questions or problems, see the Java CAPS web site at `http://goldstar.stc.com/support`.

This chapter covers the following topics:

- "WebLogic Server Components Overview" on page 5
- "WebLogic Server Components Task Overview" on page 5
- "Java Naming and Directory Interface (JNDI)" on page 6
- "Java Messaging Service (JMS)" on page 8
- "Enterprise JavaBeans (EJBs)" on page 10
- "XA Transactions" on page 12

WebLogic Server Components Overview

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

WebLogic Server Components Task Overview

The tasks included in this section allows you to perform the following,

- "Java Naming and Directory Interface (JNDI)" on page 6
- "Java Messaging Service (JMS)" on page 8
- "Enterprise JavaBeans (EJBs)" on page 10
- "XA Transactions" on page 12

Java Naming and Directory Interface (JNDI)

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

The WebLogic Naming Service

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

There are essentially two environments that have to be configured, **Context.PROVIDER_URL** and **Context.INITIAL_CONTEXT_FACTORY**.

For WebLogic, the Context.PROVIDER_URL environment is

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

where,

- <wlserverhost> is the hostname on which the WebLogic Server instance is running
- <port> is the port at which the Webserver instance is listening for connections

For example,

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

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

Sample Code

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

```
HashMap env = new HashMap();

env.put (Context.PROVIDER_URL, "t3://localhost:7001/");

env.put (Context.INITIAL_CONTEXT_FACTORY,

"weblogic.jndi.WLInitialContextFactory");

Context initContext = new InitialContext (env);

...
```

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

```
Topic topic

=(Topic)initContext.lookup("sbyn.inTopicToSunMicrosystemsTopic");

...
```

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

```
Queue queue = null;

try {

queue = new STCQueue("inQueueToSunMicrosystemsQueue");

initContext.bind ("sbyn.ToSunMicrosystemsQueue", queue);

}

catch (NameAlreadyBoundException ex)

{

try

{

if (queue != null)

initContext.rebind ("sbyn.ToSunMicrosystemsQueue", queue);
}

catch (Exception ex)

{

throw ex;

}

}
```

Viewing the WebLogic JNDI Tree

For information, see “To View the JNDI Tree Structure” in *Configuring WebLogic for Asynchronous Communications*.

Java Messaging Service (JMS)

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

Clients send messages to or receive messages from Topics or Queues (see Figure 1–1 and Figure 1–2). 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.

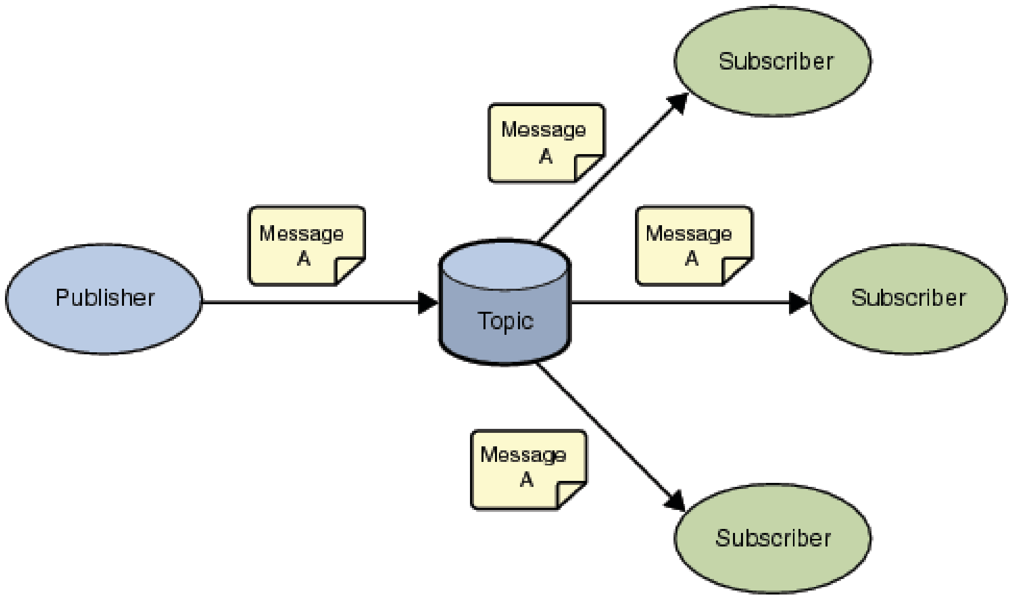

FIGURE 1–1 Topic - The Publish-Subscribe Model

Figure 1–1 shows multiple subscribers receiving the same messages when the publisher publishes the message to a Topic. This is the pubsub (publish-subscribe) model.

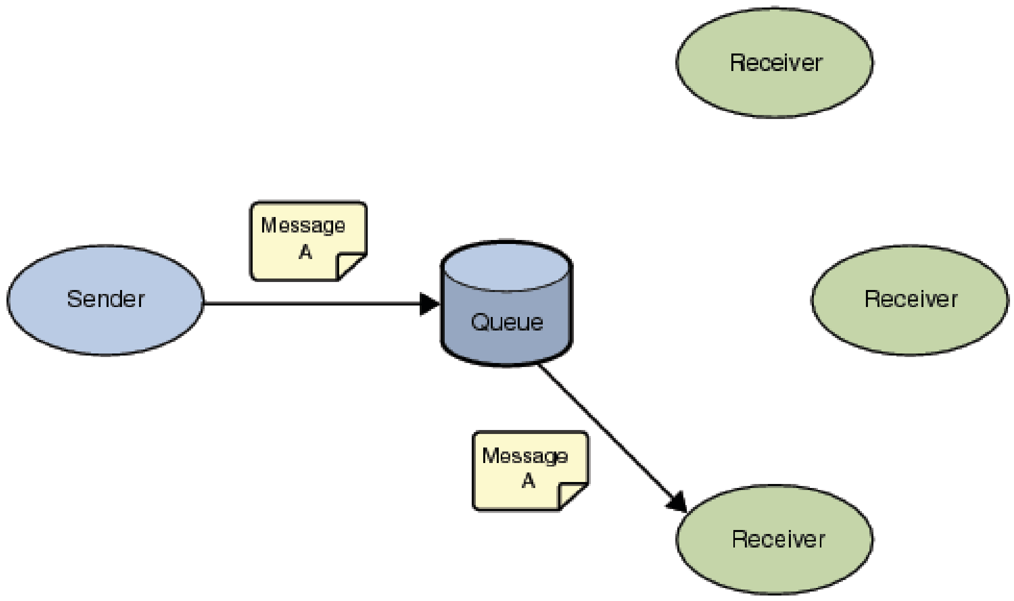

FIGURE 1–2 Queue - The Point-to-Point Model

The Point-to-Point model (Figure 1–2), on the other hand, allows for only one receiver to get the message when a sender sends a message to a Queue.

Enterprise JavaBeans (EJBs)

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

There are three types of Enterprise JavaBeans:

- Session Beans
- Entity Beans
- Message Driven Beans

Note – Do not use the WebLogic adapter to directly invoke an EJB 3.0 bean. Instead, wrap the EJB 3.0 in an EJB 2.x wrapper and then invoke the EJB 2.x bean. This workaround is a consequence of the EJB 3.0 specification itself (namely, that EJB 3.0 does not guarantee support across vendors).

Session Beans

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

Entity Beans

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

Message Driven Beans

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

An MDB implements two interfaces:

- **javax.ejb.MessageBean**
- **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.

XA Transactions

XA is a two-phase commit protocol that is natively supported by many databases and transaction monitors. It ensures data integrity by coordinating single transactions accessing multiple relational databases. XA guarantees that transactional updates are committed in all of the participating databases, or are fully rolled back out of all of the databases, reverting to the state prior to the start of the transaction.

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

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

Both Sun Microsystems JMS and BEA WebLogic Server implement the X/Open XA interface specifications. Because both systems support XA, the EJBs running inside the WebLogic container can subscribe or publish messages to Sun Microsystems JMS in XA mode. When running in XA mode, the EJBs subscribing or publishing to Sun Microsystems JMS can also participate in a global transaction involving other EJBs. For the example, EJBs running in XA mode, Container Managed Transactions (CMTs) are used. In other words, we define the transactional attributes of the EJBs through their deployment descriptors and allow the container to transparently handle the XA transactions on behalf of the EJBs. The WebLogic

Transaction Manager coordinates the XA transactions. The Sun Microsystems JMS XA Resource is enlisted to a transaction so that the WebLogic Transaction Manager is aware of the Sun Microsystems JMS XA Resource involved in the XA transaction. The WebLogic container interacts closely with the Transaction Manager in CMT such that transactions are almost transparent to an EJB developer.