Introduction

IBM® WebSphere® Application Server (hereafter called Application Server) Version 5 has strong support for Java Message Service (JMS) and message driven beans. This article helps you understand your configuration options within Application Server for JMS applications. This includes using WebSphere Embedded Messaging and WebSphere MQ (formerly MQ Series). This article also focuses on setting up your development environment to support various JMS scenarios. It discusses various development and runtime topologies and when best to use them. In addition, it covers scripting and security in reference to JMS.
This article does not provide instructions on JMS and message driven bean concepts. It focuses on configuring WebSphere Application Server to support JMS applications with various JMS Provider options, including MQ for Java Developers (MQJD), WebSphere Embedded Messaging (WEM), and WebSphere MQ (WMQ). This article assumes you have some knowledge of JMS and message driven beans, WebSphere Application Server V5 administration, and WebSphere Studio Application Developer.

**Understanding WebSphere Application Server Packaging and Resource Scoping**

To better understand Application Server JMS configurations, we’ll start by providing background information about WebSphere application server packaging and clustering in version 5 that includes:

- WebSphere Application Server Express
- WebSphere Application Server
- WebSphere Application Server Network Deployment
- WebSphere Application Server Enterprise

**WebSphere Application Server Express** is a simplified, Web-centric package with low cost of entry. It supports servlets, JSPs, and Web services, but not EJBs. Consequently, it does not include support for message-driven beans or the Embedded JMS Server present in other editions of WebSphere Application Server V5. WebSphere Express customers can use the JMS programming API, but they need to provide their own JMS provider for this purpose.

**WebSphere Application Server** provides a fully functional enterprise Java programming model (including EJBs and JMS) that runs on a single Java Virtual Machine (JVM). Analogous to WebSphere Application Server Advanced Edition Single Server in V4, it does not support clustering or workload management. It only supports a single server configuration. However, users of this product will find it easy to upgrade to WebSphere Application Server ND and merge a standalone node into a distributed configuration for improved scalability and throughput over a single JVM.

**WebSphere Application Server Network Deployment (ND)** includes all the features in Application Server, as well as support for clustering, workload management, and distributed administration. With Network Deployment, multiple Application Server nodes are included in an Application Server cluster, which is termed a cell in this release.

**WebSphere Application Server Enterprise (EE)** includes all the features in WebSphere Application Server ND, plus many programming model extensions. These extensions include advanced features, such as workflow, scheduling, and business process choreography that are not part of the J2EE specification, but may be included in future versions. It includes container managed messaging, which encapsulates message sends and receives for programmers.

We will focus here on WebSphere Application Server and WebSphere Application Server ND. In Application Server (base), everything resides in a single process on a single machine. In other words, each machine holds a separate installation of the product that is unaware of installations on other machines. When you move up to Application Server ND, you can have several managed processes on a node. The managed processes are JVMs, typically application servers, which service requests. Each node also includes a lightweight node agent process, which manages the activity on that node.

A node cannot span multiple machines, but a machine can have multiple nodes, each with multiple managed servers. You can federate these nodes into an Application Server cell, which is a collection of multiple Application Server nodes that can you can workload balance and centrally administer. Within a cell, you can define one or more clusters that are aggregations of application
servers for the purposes of workload manager (WLM) and failover. In addition to the nodes, each cell also includes a deployment manager. The deployment manager is a central server that controls and communicates with all the node agents, and maintains the master cell configuration repository.

**Tooling Packaging**

WebSphere Studio Suite is built on top of the open source Eclipse platform. WebSphere Studio contains a set of plug-ins at different levels to support different needs and different clients. For each level of IDE, there is usually a supported runtime. Furthermore, each level higher in the chain contains all the plug-ins from the level below it. The suite includes:

- WebSphere Studio Site Developer
- WebSphere Studio Application Developer
- WebSphere Studio Application Developer Integration Edition
- WebSphere Studio Enterprise Developer

**WebSphere Studio Site Developer** (hereafter called Site Developer) is a development environment targeting Web developers. It supports the development of servlet, JSP, and XML based applications, but it does not support any additional J2EE Enterprise services, such as EJBs. The target deployment platform is usually WebSphere Application Server Express. There is little support for JMS development.

**WebSphere Studio Application Developer** (hereafter called WebSphere Studio) is a full J2EE and Web Services development environment that supports EJBs and JMS. The target deployment platform is WebSphere Application Server (both base and ND). WebSphere Studio can also deploy to other compliant J2EE servers and supports database tools (stored procedure building and wrapping, XML/XSLT development, Web Services development, and an array of development tools.) WebSphere Studio contains everything Site Developer does.

**WebSphere Studio Application Developer Integration Edition** extends beyond J2EE and Web services to address the integration needs of developers. This includes support for Enterprise services (expanding Web Services over other protocols, such as JMS or RMI/IIOP), support for J2EE extensions, such as container managed messaging or dynamic EJB QL), and development and deployment for business process and workflow. Integration Edition contains everything in WebSphere Studio. The target deployment platform for Integration Edition is WebSphere Enterprise Edition. You cannot deploy the programming model extensions in WebSphere Enterprise to the “base” WebSphere.

**WebSphere Studio Enterprise Developer** supports deployment to WebSphere Application Server on the z/OS platform. It also supports tools, such as EGL (replaces VisualAge® Generator). Enterprise Developer contains everything in Integration Edition.

There are other WebSphere Studio offerings, such as WebSphere Studio Device Developer for PDAs. There are toolkits that you can download to work on top of WebSphere Studio offerings mentioned above. For example, you can install the WebSphere Portal Toolkit for development to WebSphere Portal on WebSphere Studio, or you can install the WebLogic development environment plug-in on WebSphere Studio for deployment to WebLogic.

**JMS Options with WebSphere Studio**

WebSphere Studio V5 is the tool of choice for developing and unit testing J2EE 1.3 applications for WebSphere Application Server V5. When developing and testing JMS applications in WebSphere Studio V5, you have several JMS provider options:

- By default, WebSphere Studio is configured to run a special JMS Provider called MQ for Java Developers (MQJD). Unlike the normal Embedded Messaging Server, this MQJD
simulator is a pure Java JMS provider, so it avoids installation issues and the overhead of separate C processes running in the development environment. MQJD implements all of the JMS APIs, but it does not support security checks, persistence, or JMS communications across process boundaries. As a result, it does not work with application clients.

- You can set up the WebSphere Test Environment Application Server running within WebSphere Studio to run its own Embedded Messaging Server.
- You can configure WebSphere Studio V5 to connect to an Embedded Messaging Server that is installed with Application Server V5 on another computer.
- JMS code running in the WebSphere Studio V5 test server can connect to a full WMQ provider running on a separate server.
- Finally, you can configure the WebSphere Studio unit test server to connect to third-party JMS providers.

**JMS Development Topologies**

With development choices outlined above, we can define several development environment topologies to create JMS applications.

**Development Topology with MQJD Simple**

In this configuration, WebSphere Studio runs a lightweight, pure Java messaging engine within the same JVM as the unit test server. Because MQJD does not listen on a socket for requests from other processes, this configuration does not support JMS communications with other servers or processes. It can only receive messages sent by applications within the same unit test server process.

The development environment is self-contained. Figure 1 below shows a typical developers topology with MQJD.
Development Topology with Embedded Messaging

Because WebSphere Studio V5 comes with a fully functioning JMS Server, you can configure WebSphere Studio to support the Embedded Messaging Server instead of MQJD. We will show this later. You can then support a similar topology for development as shown in Figure 2. With this setup though, you have XA support as well as the necessary listeners that allow external application clients to communicate with destinations in the JMS Server.
This environment also gives you the added benefit of setting up different scenarios, such as developers sharing a Remote Embedded Messaging Server with various WebSphere Studio test environments connected through the Embedded Messaging client.
Development Topologies with WebSphere MQ

Finally, you can configure your development environment to support WebSphere MQ (formerly MQSeries). The same Embedded Messaging client can also connect to a Remote Queue Manager in client mode.
Furthermore, as of version 5.3 of WebSphere MQ, the Embedded Messaging Client inside Application Server supports XA transactions. Previously, this was only possible using bindings mode, which required that the Queue Manager and client reside on the same node.

If you need to communicate with an older version of MQ, perhaps mainframe applications, and support XA, you can setup a development server with WebSphere MQ 5.3 for developers to share. This development server can then communicate to older versions of MQ using Sender and Receiver channels as shown in Figure 5.
JMS Runtime Topologies

There are several options for setting up your test and production runtime topologies. In this section, we discuss various runtime topologies and practices for setting up the system architecture for JMS applications.

WebSphere Embedded Messaging Topologies

WebSphere Embedded Messaging (WEM) is useful in a unit test environment, and for pilot or small scale production deployments, where JMS is only utilized inside the WebSphere Application Server domain. It is not recommended for large-scale production scenarios or application integration because it does not support remote queuing, QMgr channel configuration, message flows, message transformation, or encrypted client-server communications. You can use it for simple, pure asynchronous scenarios, such as logging, sending email, or firing off a batch job.
You can only use Embedded Messaging inside Application Server domains so this places a few requirements on the Application Server cell topology. To use Embedded Messaging in a cell, you must install the WEM server on a node that is configured as part of the cell. Application server instances in the same cell that want to use WEM must have the WEM client installed on the same node as the application server. You can only have one instance of the WEM server for a specific queue on a node, but you can have multiple instances of the WEM server, each with a distinct queue in a cell. Thus, you can host JMS applications on different nodes. As long as those nodes have the WEM client installed, they can make JMS calls to a WEM server on another node in the Application Server cell.

The set of topologies below are not intended to be exhaustive, but they do depict typical patterns. They represent a progression from simple topologies to more complex ones for higher QoS, scalability, and failover requirements.

**WebSphere Application Server Base with Embedded Messaging**

This topology is for the WebSphere Application Server (“base”), in which all components run on a single application server JVM. Figure 6 depicts the Embedded Messaging server running in Application Server. In this case, Application Serve runs as one JVM, and this instance of Application Server contains, among other components, the deployed J2EE applications, the JNDI namespace, and the Embedded Messaging server. The Embedded Messaging server provides the necessary runtime support to host JMS 1.0.2-based applications. It houses the publish-and-subscribe broker component and a security listener. Most importantly, it serves as a gateway to the underlying native messaging provider (based on WMQ) to enable the Application Serve administration tools to create and manage the underlying WMQ objects, such as the queue manager and queues.

In this topology, there is one queue manager per Embedded Messaging server. This queue manager is created and started automatically at server startup and requires no other configuration.

JMS objects such as connection factories, topics, and queues are defined using the Application Server Administration console (through WebSphere JMS Provider dialogs), or the wsadmin JMX-based command line interface. The objects are registered in the JNDI namespace of the WebSphere Application Server server, so they are looked up by applications. The physical queues are also defined using the administration console or wsadmin.

For monitoring and checking the status of resources, such as queue depths, there are a couple of options. The preferred option is to use the JMX Mbeans provided with Application Serve for monitoring: JMSBasicFunction, JMSAdministration, and EmbeddedJMSAdministration.

You can access these Mbeans through wsadmin or programmatically. Secondly, you can use the traditional WMQ administration utilities, such as runmqsc or MQExplorer, to look at queues and other resources. If you do use these utilities, it is essential that you do not make any configuration changes to the Application Server queue manager and queues. These resources are under the control of Application Server. Making changes to these resources using the MQ utilities results in a non-functioning configuration.
In a distributed environment, WebSphere applications are deployed across several nodes (which comprise a cell) to increase scalability and failover. Each node within a cell hosts one or more Application Server JVMs, as shown in Figure 7. In Network Deployment, the Embedded Messaging server runs in its own JVM and is monitored by certain Application Server administrative components that run on the Application Server node. The functionality of each Embedded Messaging server is the same as in the WebSphere Application Server “base” product.

In Figure 7, the Embedded Messaging server is configured on Node B. Node A is configured with only the Embedded Messaging client. This client allows a JMS-based application to communicate with an Embedded Messaging server, whether on the same node or on a remote node. To enable this topology, JMS administered objects that are shared by multiple application server instances are defined with the appropriate scope (server, node, or cell) so that the applications running on other instances can lookup these objects in JNDI. In our example, cell scoping allows a JMS application on Node A to obtain a reference to the corresponding JMS connection factories, and the JMS destinations that are defined in the JNDI namespace on Node B and physically hosted by the Embedded Messaging server runtime on Node B. *(You can scope resources placed in the WebSphere naming server at various levels: either the server (managed process) level, the node level (managed processes on the same node within a cluster), or the cell level (on all the nodes throughout the cluster). This affects the visibility of the resources to other applications.)*

It is not possible to have more than one Embedded Messaging server on a node. However, it is possible to have multiple Embedded Messaging servers in a cell. If the cell supports more than one application, you may want to have multiple Embedded Messaging servers in a cell. For
example, a cell that supports both order processing and payroll applications could have two JMS servers, one for each application, respectively, if the load necessitates it.

In this topology, you can have several Application Server MDB instances running in different application server clones, and these MDB instances are tied to the same queue. This allows the inbound messages to be balanced across different application servers, thus allowing the EJB work to be distributed and scaled. You cannot add more Embedded Messaging servers to scale the queue manager work. If you want failover, you must use a hardware clustering solution, such as High Availability Cluster Multiprocessing (HACMP). HACMP is a hardware failover technology for the IBM® RS/6000 AIX platform. Similar technologies are available for other operating systems, such as Veritas® Cluster Manager, Sun® Cluster, MC/Serviceguard and Microsoft® Cluster Server.

Figure 7. WebSphere Application Server ND with Embedded Messaging

WebSphere MQ Topologies

If you need the additional qualities of service that are not present in the Embedded Messaging server, such as remote queuing, QMgr channel configuration, message flows, encrypted client-server communications, and greater performance tuning flexibility, then upgrade to WebSphereMQ. Options also exist in WebSphere MQ for clustering that do not exist in the Embedded Messaging server. The key addition that makes these topologies possible is that WebSphere Application Server V5 ships with the Embedded Messaging client, which is same as the WMQ 5.3.0.1 client. This supports XA connections to a remote queue manager on the distributed platforms.
WebSphere Application Server ND with a single WMQ queue manager either on same or different box

This is the simplest topology involving WMQ (see Figure 8). This topology uses a single WMQ queue manager, which is either on the same machine or a physically separate machine from the application server. A single queue manager, on the right piece of hardware with the right disks, can support extremely high-volumes of persistent messages, so this topology is quite powerful. WMQ can be on the same machine or reside on a different one.

Physical queues are created and manipulated using the WMQ administration utilities. JMS connection factories and destinations are created using wsadmin or through the Application Server administrative console under the WebSphere MQ JMS Provider heading. The exception to this is if you need to set up connection factories to use SSL so that client-server communications with WMQ are encrypted. In this case, you need to use the WMQ jmsadmin utility. You can monitor queues using WMQ utilities and some administration capabilities are available through the JMSAdministration Mbean (JMX managed bean).

Figure 8. WebSphere Application Server ND with one WMQ queue manager
WebSphere Application Server ND with WMQ on a Hardware Cluster

If you want to be completely protected with minimal delays in the event of a queue manager failure, set up two queue managers using hardware clustering technology. You can configure the two queue managers in either active/active or active/standby mode. It is recommended to have your queue managers on hardware that is separate from the Application Server instances because hardware configuration for hardware clustering is non-trivial and costly.

If your WMQ cluster has two queue managers (see Figure 9), and you are in active/standby mode, define one QueueConnectionFactory corresponding to the active queue manager. In this example, QCF_QM1 and QCF_QM2 correspond to the active (primary) QMGR_1. If QMGR_1 fails, QMGR_2 takes over and the clients do not notice the failure.

Figure 9. WebSphere Application Server ND with highly available queue manager cluster
WebSphere Application Server ND with WMQ Cluster

Clustering of messaging servers is possible through queue manager clustering, which extends the scalability of WebSphere MQ. This topology combines Application Server clustering with WMQ clustering for the most optimal workload management of inbound requests. The addition of WMQ clustering at setup in the following topology allows for workload distribution across queue managers. The Application Server server instances in turn pull inbound load from the queue managers and queues.

In Figure 10, QM1, QM2, and QM3 are in a WMQ cluster. Q3 (on QM3) is setup as a clustered queue (no underlying physical queue), which the applications can then “put” messages on. Q1 is defined on both QM1 and QM2 as a local queue with attribute of cluster. Application Server instance JVM_A1 has MDBs whose listener ports point to QCF_QM1 and Q1. Application Server instance JVM_B1 has MDBs whose listener ports correspond to QCF_QM2 and Q1.

Applications connect to the queue manager on node C and put to the clustered (logical) queue Q1 on QM3. Since this is a clustered queue, any messages arriving at this queue are distributed in a round-robin fashion to the physical Q1 instances on QM1 and QM2. This focuses all the costly MQI channel work to a single queue manager, and allows QM1 and QM2 to focus on message queuing. As throughput demands increase, you can add additional queue managers to meet scaling requirements. The traffic from the incoming queue managers need to be split across the Application Server nodes accordingly.

Figure 10. WebSphere Application Server ND with WMQ cluster

The risk is that if, say, QM2 goes down, there could be a small number of messages on Q1 of QM2 that are orphaned and would not get into the appserver until the QM2 comes back up.
Also, the JVM on node B would be without work until QM2 restarts. The following derivative of WMQ3 improves upon this situation (see Figure 11).

**Figure 11. Improved derivative of WebSphere Application Server ND with WMQ cluster**

In Figure 11, each queue manager has two JVMs servicing requests from it, one on each node. This accommodates two different scenarios:

- If one of the QM1 or QM2 queue managers fails, each appserver node is kept busy with requests from the other queue manager.
- If the JVM on node B fails, QM2 is still able to have its inbound work queues serviced by one of the JVMs on Node A.

Thus, this topology reduces the likelihood of failure or orphan cases.

**Failover Issues of Runtime Topologies**

The above runtime topologies cover load balancing and failover of *inbound* requests into the application server. However, they do not cover failover and load balancing of *outbound* requests from the application server. Remember that each MDB can listen only to a single queue manager. On the listener port for an MDB, you specify one QCF that corresponds to only one queue manager.

For that matter, for any J2EE component, the JMS model requires an application to use a specific ConnectionFactory, which maps to one and only one queue manager. Application Server V5 has no mechanism to automatically failover to another queue manager within the WebSphere MQ cluster. In our WMQ3 example, let’s suppose QM2 goes down, and that is the one that an MDB is...
supposed to reply to. In this case, the MDB instances running in Application Server JVM_B1 cannot send JMS reply messages or any outgoing traffic.

As a result, the only way, at present, for an application to achieve true workload balancing and failover for outbound messages is to make your destination queue manager highly available using WMQ facilities, or you must build failover into your application code. One simple way to do this is to configure each session bean or MDB sending a reply message with a primary and backup ConnectionFactory. You should not hardcode this in the code, but include it in a configuration file or deployment descriptor. The application tries to send to one ConnectionFactory, traps exceptions, and then tries backups in case of exception.

If you really need to, you can roll a more comprehensive workload management solution. One such workaround is add true load balancing and failover capability from the application server to MQ for outbound JMS traffic. Install a WebSphere Application Server on each WMQ box that you want to load balance. These boxes then require hardware clustering to protect persistent messaging. You can then create some JMS session beans (let’s call them sender/receiver beans) that exist only to take care of JMS activities, and do not have any business logic. You would deploy these sender/receiver beans on the application servers that are on the WMQ boxes. Then, if your original MDB or session beans that are running on a separate box invoke the put/get methods of the sender/receiver beans (rather than making direct JMS calls), you can extend Application Server load balancing and failover to WMQ.

This solution is fully transactional, but slower than talking to WMQ directly. Of course, this is not perfect – it results in the cost of additional Application Server licenses, creating the risk of an additional tier of Application Server servers going down, significant deviation from traditional JMS coding, and so on. It may be easier to roll your own workload management layer, but it is preferred to use MQ and hardware clustering facilities to make your WMQ infrastructure highly available.

Setting up your WebSphere Studio and WebSphere Application Server for JMS

To support various configurations, it is important to understand how to setup your environment. In this section, we show how to configure JMS support for various configurations using WebSphere Studio. Because WebSphere Studio contains a base version of Application Server V5, you can use these settings on the server. This section does not show you how to deploy once you have setup your JMS provider. To test the configuration you can use the following articles as reference. Each article has a sample application you can install.

- Creating and Testing Message Driven Beans using WebSphere Studio Application Developer 5.0
- Developing and Testing Message-driven Bean Applications with the MQ Simulator for Java Developers in WebSphere Studio Application Developer 5.0

Note: This is only with the WebSphere 5 test environment. WebSphere 4 has no built-in JMS support and requires an external JMS provider. To set up JMS support for WebSphere 4 test environment inside WebSphere Studio 5, follow the directions in one of these articles:

- WebSphere JMS/JTA support for MQSeries Overview
- Develop an Asynchronous Logging Framework using log4j with JMS and WebSphere MQ
WebSphere Studio MQJD

As stated, WebSphere Studio comes installed a lightweight in-process only JMS server. When you run the install program for WebSphere Studio, you see several install options. After running **Install IBM WebSphere Studio Application Developer**, MQJD gets installed with it. You will notice there is a separate “**Install embedded messaging client and server option**.” To use MQJD, you do *not* need to install this. Figure 12 illustrates the install screen.

**Figure 12. Installing WebSphere Studio with MQJD**

![WebSphere Studio Installation Launcher]

Once you install WebSphere Studio, you have JMS support. However, it is important to understand that you can do limited unit testing. There are no network hooks or channels built in. You can only write and read messages from within the same JVM. For many unit test scenarios, this is valid.

If you are using MQJD, most of the configuration settings for JMS applications are all within WebSphere Studio, including Embedded Messaging. You can open the server configuration editor in the Server Configuration view in the Server Perspective as shown in Figure 13.
Figure 13. WebSphere Studio Server Perspective

The WebSphere Studio Server Editor contains a **JMS** tab as shown in Figure 14.

Figure 14. JMS tab of WebSphere Studio Server Editor

Here you can define your Connection Factories and your destinations. You can also apply the settings of the JMS Server shown in Figure 15. Because you are dealing with one JVM, it is not necessary to have different scope settings. Finally, you can also set the initial state of the JMS Server to Start or Stop.
Figure 15. WebSphere Studio JMS Server settings

Let's set the JMS Server state to **START** to start the MQJD Server whenever we start our Application Server test server. You can also adjust your message driven bean settings on the **EJB** tab of the server configuration editor as shown in Figure 16.
As you recall from the message driven bean articles previously mentioned, WebSphere uses a listener port to bind the message driven beans to the actual JNDI names of the JMS objects. By going into the EJB tab, you can set the listener port to configured JMS resources on the JNDI tree.

With MQJD, you can do all the settings within WebSphere Studio. However, you can also do this from the WebSphere Admin Console. Although this is not necessary, we will illustrate to make the connection between WebSphere Studio and WebSphere. Furthermore, it helps us understand how to configure JMS with WebSphere Application Server itself. We need to do this later when using WebSphere MQ.

You can enable the administrative console for the WebSphere Server inside WebSphere Studio on the Configuration tab of the Server Configuration Editor. You see an option for enabling the administrative console as shown in Figure 18.
The WebSphere Administrative Console is a Struts-based Web application. Setting this option deploys the administrative console. Start the server to use the administrative console because it is a Web application. You can do this in the Server perspective by right-clicking a Server in the Servers View and selecting Start as shown in Figure 19.
Once you start the server, you need to wait for the following message:

**Server server1 open for e-business**

Scroll up to the console and look for the MQJD messages. This is a good indication of which provider we are using. If you look at the console, you notice a few messages as shown in Figure 20.

**Figure 20. Test server console startup messages**

- **CJ0240I**: WebSphere configuration information successfully pushed to a
- **GS0602I**: Security service initialization completed successfully
- **AS0001I**: WebSphere Embedded Messaging Client only has been installed
- **AS0002I**: Authentication protocol: CSIV2/IBM
- **AS0003I**: Authentication mechanism: SWAM
- **AS0004I**: Principal name: localhost/db2admin
- **AS0005I**: SecurityCurrent registered.
- **AS0006I**: Security connection interceptor initialized.
- **AS0007I**: Client request interceptor registered.
- **AS0008I**: Server request interceptor registered.
- **AS0009I**: IOR interceptor registered.
- **CJ0136I**: Custom Registry.com.ibm.ws.security.registry.nt.NTLocalDoma
- **MC0013I**: SOAP connector available at port 8880
- **CJ0243I**: Security service started successfully
- **CJ0210I**: Security enabled true
- **GS0656I**: Starting the MQJD JMS Provider
- **GS0650I**: MQJD JMS Provider open for business
The first one is a “WebSphere Embedded Messaging Client only has been installed” message. Note that you can, in your application, be a client to another embedded server with MQJD. You can look up Connection Factories from an embedded server running in another server.

The next two messages indicate that MQJD is starting and is open for e-business because we set the initial state to START earlier.

**Configuring JMS through the WebSphere Admin Console**

In WebSphere Studio, you do not need to go into the Admin Console for WebSphere JMS (MQJD or Embedded Messaging). Instead, use WebSphere Application Server installed in WebSphere Studio to configure WebSphere JMS through the admin console. If you are using a standalone version of WebSphere, the same settings still apply, except you are using the installed Embedded Messaging. The next section shows how to configure WebSphere Studio to use Embedded Messaging instead of MQJD.

You can access the administrative console by doing the following:

1. From the server view, you can right click your server and select **Run administrative console** as shown in Figure 21.
2. You can also bring up the administrative console by opening a browser and going to the following URL: [http://localhost:9090/admin](http://localhost:9090/admin).

3. The administrative console shows a login form. If security is not configured on the server, you can enter the ID that you are using on your developer’s machine.
4. The admin console's main page appears. To create the Resources for MQJD using the admin console rather than WebSphere Studio, go to the Resources section of the menu and select **WebSphere JMS Provider**.

5. After selecting WebSphere JMS, you are in the WebSphere JMS Provider page as shown in Figure 24. This page allows you to configure JMS resources. You create the JMS Connection factories, queues, and topics. You also see the various scope levels available for resources. You can define resources in WebSphere (JMS, JDBC, J2C, and so on) at three levels as mentioned before:
• Cell level makes the resource available throughout a cluster. A cell defines a single WebSphere Administrative domain and allows servers residing on different network nodes to share resource definitions. Inside WebSphere Studio, we are not usually concerned with this level.
• Node level makes the resource available on a single network node. This means the multiple processes on the same machine can share definitions.
• Server means the definition is only available inside a single process or JVM.

6. The links on the bottom allow you to create the various resources. You can create both queue and topic factories and destinations. Let's look at the queue as a sample.
Figure 24. Websphere JMS Provider page

**WebSphere JMS Provider**

A JMS provider enables asynchronous messaging based on the Java Messaging Service connections for specific JMS queue or topic destinations. WebSphere JMS provider administers the internal WebSphere JMS provider.

7. Click the **WebSphere Queue Connection** factory link. The WebSphere Queue Connection Factories page appears as shown in Figure 25.
8. Press the New button to bring up the New Connection factory page. Enter the following information:

- Unique name for the factory
- JNDI
- Description
- Category
- Node (In this case, this defines the physical location of the factory, rather than the scope in which the definition is available for clients to look up.)
- Authentication Alias
- Whether the Connection Factory supports XA

Figure 26 shows the screen.
A queue connection factory is used to create connections to the associated JMS provider of JMS queue objects to manage queue connection factories for the internal WebSphere JMS provider.

9. Similarly, you can click on the **New** button on the **WebSphere Queue Destinations** page to create a queue as shown in Figure 27.
WebSphere Queue Destinations

Queue destinations provided for point-to-point messaging by the internal WebSphere objects to manage queue destinations for the internal WebSphere JMS providers.

10. After clicking New, the New Queue Page appears as shown in Figure 28. Similarly, you can do this with topics.
Finally, you can also set the start state and create MDB listeners from the admin console. From the menu tree, expand **servers** and select **Application Servers** as shown in Figure 29.
12. From the Application Servers screen, click server1 link as shown in Figure 30.

13. Look for the Server Components link as highlighted in Figure 31.
server1

An application server is a server which provides services required to run enterprise applications.

14. In the Server Component screen, select the JMS Servers link as shown in Figure 32.
15. You can now set the JMS server setting through the admin console as shown in Figure 33. You can add queues for this JMS Server and the start state. Note in WebSphere’s base install, there is a single JVM and, therefore, the JMS server is a server component. With WebSphere Network Deployment, the JMS Server is its own JVM per node. We will show this later.
You can also create the Listener Port in the admin console. Back on the server1 screen, select the **Message Listener Service** link as shown in Figure 34.
An application server is a server which provides services to other software components.

<table>
<thead>
<tr>
<th>Runtime</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Properties</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td></td>
</tr>
<tr>
<td>Initial State</td>
<td></td>
</tr>
<tr>
<td>Application class loader policy</td>
<td></td>
</tr>
<tr>
<td>Application class loading mode</td>
<td></td>
</tr>
<tr>
<td>Apply</td>
<td>OK</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Additional Properties</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction Service</td>
<td>Specify settings and options for transactions.</td>
</tr>
<tr>
<td>Web Container</td>
<td>Specify thread pool settings and transport settings.</td>
</tr>
<tr>
<td>EJB Container</td>
<td>Specify cache and connection pooling settings.</td>
</tr>
<tr>
<td>Dynamic Cache Service</td>
<td>Specify settings for dynamic cache services.</td>
</tr>
<tr>
<td>Logging and Tracing</td>
<td>Specify Logging and Tracing configurations.</td>
</tr>
<tr>
<td><strong>Message Listener Service</strong></td>
<td>Configuration for ListenerPorts that listen for incoming messages.</td>
</tr>
<tr>
<td>ORB Service</td>
<td>Specify settings for the ORB service.</td>
</tr>
<tr>
<td>Custom Properties</td>
<td>Additional custom properties.</td>
</tr>
</tbody>
</table>

17. From the Message Listener screen, select the **Listener Ports** screen.
18. Create a new Listener Port by selecting the **New** button.

19. Finally, just like in WebSphere Studio, you can map the listener to its corresponding Connection Factory and Destination as shown in Figure 37.
Mapping the message driven bean to a listener port is a deployment time issue and usually done with WebSphere Studio or the assembly toolkit.

**WebSphere Studio and WebSphere Application Server with Embedded Messaging**

MQJD is a feature only available in WebSphere Studio. WebSphere Application Server comes with a fully embedded JMS Server based on WebSphere MQ. The configuration of WebSphere Embedded JMS is similar to what we saw for MQJD from the admin console. You can configure WebSphere Studio to use WebSphere JMS.

As you remember, when installing WebSphere Studio, the initial screen installs the Embedded Messaging and client. **Note:** By installing this, you are installing the embedded JMS Server. Furthermore, for WebSphere Studio, it changes the WebSphere Test Environment from MQJD to Embedded Messaging as shown in Figure 28. The Integration Edition screen may look different.
Within WebSphere Studio, WebSphere JMS refers to MQJD before running the Embedded Messaging install, and WebSphere JMS refers to Embedded Messaging after the install.

This change is transparent. It does this by updating a file in the runtime directory of the WebSphere Application Server test environment. The name of the file is `implfactory.properties`. The file in the `<install dir>\runtimes\base_v5\properties` directory points to the embedded connection factory. For the Enterprise Edition test environment, the file is also updated in `<install dir>\runtimes\ee_v5\properties`.

If you search for the JMS Provider in the file, you see the following section. The MQJD Provider is commented out and the Embedded JMS Provider is installed instead.

```shell
#Embedded JMS Provider
com.ibm.ws.messaging.JMSProvider=com.ibm.ws.messaging.JMSEmbeddedProviderImpl

#MQJD JMS Provider
#com.ibm.ws.messaging.JMSProvider=com.ibm.ws.messaging.JMSMQJDProviderImpl
```

You can switch your WebSphere Studio runtime back and forth between Embedded and MQJD by switching this property. WebSphere Application Server by itself does not contain MQJD so
Embedded Messaging is always WebSphere JMS. It is an option during the WebSphere Application Server install.

Because WebSphere JMS uses native binaries, you need to setup your OS Path to find three directories as shown below:

```
SET PATH=F:\Program Files\IBM\WebSphere MQ\bin;%PATH%
SET PATH=F:\Program Files\IBM\WebSphere MQ\java\bin;%PATH%
SET PATH=F:\Program Files\IBM\WebSphere MQ\WEMPS\bin;%PATH%
```

The install should also update the following three WebSphere substitution variables:

```
MQ_INSTALL_ROOT=Root of embedded messaging
MQJMS_LIB_ROOT=$<MQ_INSTALL_ROOT>/java/lib
MQ_PUBSUB_ROOT=$<MQ_INSTALL_ROOT>/WEMPS
```

The first variable is where Embedded Messaging is installed. The second is the location of the jar files needed for the JMS provider. The last one is needed if you plan to use Publish and Subscribe model. If you plan to use WebSphere JMS or WebSphere MQ JMS, these variables are necessary.

You can set these variables in both WebSphere Studio and WebSphere. In WebSphere Studio, you can use the Server Configuration Editor to set the variables as shown in Figure 39.
Similarly, you can update Substitution variables in the WebSphere Admin console by going to the **Manage WebSphere Variables** from the Environment section of the navigation screen as shown in Figure 40.
The Substitution variables screen allows you to add, edit, or delete variables.
Once this is done, Embedded Messaging is set up. From there, you can define Connection Factories, Destinations, and MDB Listener Ports like we did before in WebSphere Studio and WebSphere.

Embedded Messaging is a scaled down version of WebSphere MQ. Although you do not need to define queue managers, queues, channels, and other administrative objects as you do in WebSphere MQ, it still functions similarly. However, JMS is the only available API.

Embedded Messaging actually creates a Queue Manager and Server Connection Channels underneath the hood. This is hidden from the WebSphere administrator. When starting your WebSphere Server either stand-alone or within WebSphere Studio, you should no longer see references to MQJD. Instead, you see messages as shown in Figure 42.
WebSphere Studio 5.1 Update

WebSphere Studio 5.1 makes it easier to switch the default messaging provider between MQJD and Embedded Messaging by removing the need to change the underlying properties file. Under the JMS tab of the Server Configuration Editor, there is a new JMS Provider combo box that allows you to switch modes as shown in Figure 43.

Using the WebSphere Embedded Messaging Client to connect to WebSphere MQ

By installing Embedded Messaging, you also install the needed binaries to connect to WebSphere MQ in client mode. From a setup point of view, doing the setup for Embedded Messaging is all you need to do. However, you can also use a standard MQ Client install. Standard MQ client setup does not include Publish and Subscribe support and would need a separate MA0C patch installed or MQ Event Broker.

The difference is how you configure the JMS objects. You cannot do this with the editors within WebSphere Studio. However, if you can configure WebSphere Studio through the normal WebSphere Admin console, so you can configure WebSphere Studio and WebSphere to configure administrative objects for WebSphere MQ.
If you bring up the Administrative console as before and expand the navigation tree, you see both WebSphere MQ JMS and WebSphere JMS. WebSphere MQ JMS connects with WebSphere MQ. The navigation tree is shown in Figure 44.

Figure 44. Navigate to WebSphere MQ JMS Provider

- Servers
- Applications
- Resources
  - J2EE Providers
  - Generic JMS Providers
  - WebSphere JMS Provider
    - WebSphere MQ JMS Provider
  - Mail Providers
  - Resource Environment Providers
  - URL Providers
  - Resource Adapters
- Security
- Environment
- System Administration
- Troubleshooting

Clicking on the **WebSphere MQ JMS Provider** link brings you to the main WebSphere MQ JMS screen. It is similar to WebSphere JMS. The four links on the bottom allow you to create the administrative objects. Figure 45 shows the WebSphere MQ configuration screen.
Figure 45. WebSphere MQ JMS Provider configuration

Clicking on those links brings you to the configuration of the configured WebSphere MQ objects, and allows you to create new ones as shown in Figure 46.
WebSphere MQ JMS Provider >

WebSphere MQ Queue Connection Factories

A queue connection factory is used to create connections to the associated JMS provider of JMS queue or topic. Use WebSphere MQ Queue Connection Factory administrative objects to manage queue connection factories.

Clicking on New creates a Connection Factory for WebSphere MQ as a provider. Looking at Figure 47, you can enter several options. You can enter the Name and JNDI Name as before. You can enter WebSphere MQ specific configurations. In client mode, you need to enter the Queue Manager Name, the host of the Queue Manager, and the port the Server Connection Channel that is configured on the Queue Manager Host. The transport type needs to be changed to CLIENT. Bindings mode requires that the Queue Manager is on the same node as WebSphere. You can read about other specific WebSphere MQ options.
You can also create a Queue Definition and associate WebSphere MQ specific options as shown in Figure 48. You still need to create queues in WebSphere MQ separately. The Name text box makes the association.
One option that is interesting is the ability to set the target client. By setting this to JMS, your JMS program is limited to only understanding messages written or read by other JMS programs. By setting the target client to MQ, JMS programs inside WebSphere can understand messages written to WebSphere MQ by the other APIs supported by MQ. It is common for this option to be
set to MQ in scenarios where WebSphere Application Server communicates with WebSphere MQ on the mainframe. The JMS option is highlighted in Figure 49.

Figure 49. Setting target client

<table>
<thead>
<tr>
<th>Target Client</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>JMS</td>
<td></td>
</tr>
<tr>
<td>MS</td>
<td></td>
</tr>
<tr>
<td>MQ</td>
<td></td>
</tr>
</tbody>
</table>

If WebSphere MQ is configured with MA0C or WebSphere MQ Event Broker, you can configure the extra settings for the Publish and Subscribe broker when creating a Topic Connection Factory as shown in Figure 50.
You can also configure topic options at the topic level as shown in Figure 51. For example, configuring durability of topic definitions in pub/sub applications is done here. Durability is the ability for the messaging broker to save messages for durable subscribers while they are not connected, such as failures. Durable subscribers have to explicitly disconnect from the TOPIC.

<table>
<thead>
<tr>
<th>Transport Type</th>
<th>BINDINGS ▼</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td></td>
</tr>
<tr>
<td>Queue Manager</td>
<td></td>
</tr>
<tr>
<td>Broker Control Queue</td>
<td></td>
</tr>
<tr>
<td>Broker Queue Manager</td>
<td></td>
</tr>
<tr>
<td>Broker Publication Queue</td>
<td></td>
</tr>
<tr>
<td>Broker Subscription Queue</td>
<td></td>
</tr>
<tr>
<td>Broker CC Subscription Queue</td>
<td></td>
</tr>
<tr>
<td>Broker Version</td>
<td>Advanced ▼</td>
</tr>
<tr>
<td>Model Queue Definition</td>
<td></td>
</tr>
<tr>
<td>Clone Support</td>
<td>☐ Enable clone support</td>
</tr>
<tr>
<td>Client ID</td>
<td></td>
</tr>
<tr>
<td>CCSID</td>
<td></td>
</tr>
<tr>
<td>XA Enabled</td>
<td>☑ Enable XA</td>
</tr>
</tbody>
</table>
Setting up WebSphere with WebSphere MQ Bindings Mode

When using WebSphere MQ in bindings mode, you do not need to install Embedded Messaging because WebSphere Application Server and WebSphere MQ are on the same node. Because WebSphere Embedded Messaging is a lightweight WebSphere MQ, there is no need to have it installed when WebSphere MQ is installed.

Configuring JMS objects is no different than client mode, except you are using bindings mode. It is configured at the Connection Factory level as we showed in client mode. Figure 52 re-illustrates the setting.

It is not necessary to set the host, port, or channel name in bindings mode.

To have WebSphere MQ as the default provider on the same node, the same three WebSphere Substitution variables still need to be set. The install should also update the following three WebSphere substitution variables:

MQ_INSTALL_ROOT=Root of WebSphere MQ
MQJMS_LIB_ROOT=$<MQ_INSTALL_ROOT>/java/lib
MQ_PUBSUB_ROOT=$<MQ_INSTALL_ROOT>/WEMPS or root of Publish and Subscribe install.

Lastly, WebSphere MQ by default does not install with a Publish and Subscribe engine, so you need to install it on top of WebSphere MQ. For standard JMS Publish and Subscribe support as provided by Embedded Messaging, it is necessary to install the standard MA0C MQ pack. However, for a robust Publish and Subscribe engine, you can use WebSphere MQ Event Broker.

WebSphere ND and JMS

So far, we have been talking about WebSphere Application Server, the base product – a single server that stands by itself and is not part of a larger group of servers. When you deploy WebSphere Application Server into a production environment, you will most likely be using the Network Deployment (ND) version of the product because this supports clustering, workload management, and distributed administration. It allows you to federate multiple single servers into
a cell of server, which often spans servers on multiple nodes. When you setup a network
deployment cell, there are several changes from the base topology that pertain to JMS that you
need to know (see Figure 7).

The embedded JMS server in the base product is actually a component of each individual server.
When you federate an individual (previously standalone) application server into a Network
Deployment cell, the federation process creates a single JMS server on each node that all the
application servers on that node or in the cell can use to service JMS requests. This is analogous
to a state joining a nation or federation of states. Previously, it had its own state-run postal
service. Now, when joining the union, it can share a postal server with the other states (servers)
in the same geographical region (node).

If you have plans to move a server in a test or production environment from single server to ND,
you want to define your JMS resources with cell scope even on a single server. This minimizes
maintenance later when you move to ND. The federation process can handle the proper
migration of definitions at the cell level.

Figure 53 shows the view of the separate JMS server in the Administrative console. Contrast this
with the view of the JMS server as a component of the base application server in Figures 32 and
33.

Figure 53. WebSphere Application Server ND admin console showing separate JMS Server

As discussed earlier in the messaging topologies section, multiple nodes in a cell can either share
one JMS server on one of the nodes, or they can use multiple JMS servers (on more than one
node) if the cell services multiple applications. When you are setting up a Queue Connection
Factory or Topic Connection Factory in a cell environment, where the application may reside on
one node and the JMS server is on another node, ensure the objects are configured properly for
JNDI namespace visibility.

One option is to setup a Connection Factory with Node scope as shown in Figure 54 (on the node
on which the application that utilizes the Connection Factory runs), and point it to the JMS Server
on the remote node. The preferable alternative is to setup the Connection Factory with cell scope,
so that the Connection Factory is visible to all servers in the cell because it is placed in the cell-
wide portion of the JNDI namespace. This is less topology dependent than the former approach.
Scripting for Configuration

With WebSphere Application Server 4, if you were using MQSeries, you had to define the JMS objects using the MQ jmsadmin command line utility to bind them into the Application Server JNDI namespace. In most cases with Version 5 (the exception being connection factories with SSL attributes that need to be set), you no longer have to use jmsadmin. When you define a JMS Connection Factory or destination using the Administrative console or the command-line scripting facilities, the admin infrastructure takes care of binding the object into the Application Server JNDI namespace. There is no extra jmsadmin definition needed.

After you go through the process of creating connection factories and destinations a few times using the administrative console, you will want to automate it. Wsadmin is a command-line scripting tool which ships with WebSphere Application Serve 5. You can use it to execute interactive scripting sessions or batch scripts to create administrative objects and modify their attributes, and also monitor and administer objects running in the cell.

We will review the steps to script the creation Connection Factories and destinations for the embedded JMS provider. The same principles apply to WMQ or a third-party provider. We will also show how to perform a few simple administrative functions using wsadmin. The scripts apply to both base and Network Deployment installations. Note that wsadmin is case-sensitive.

To create a Queue Connection Factory using the WebSphere (embedded) JMS provider:

1. With Application Server started, under the “bin” directory of your Application Server install root, launch the wsadmin .bat or .sh script. This brings up an interactive wsadmin prompt.

2. First, you’ll need to set variables to point the name of your Application Server cell, node, and the desired JMS provider, and then create the provider. On each of the lines below, the first line shows you the wsadmin command, the second line shows you an example of what the output would look like.
wsadmin>set mycell [$AdminControl getCell] myserver
wsadmin>set mynode [$AdminControl getNode] myserver
wsadmin>set myjmsp "WebSphere JMS Provider"
wsadmin>set embeddedjmsp [$AdminConfig getid /Cell:$mycell/Node:$mynode/JMSProvider:$myjmsp/]
"WebSphere JMS Provider(cells/myserver/nodes/myserver:resources.xml#builtin_jmsp provider)"

3. Next, find out what the required attributes are for the WASQueueConnectionFactory:

$AdminConfig required WASQueueConnectionFactory
Attribute Type
Name String
jndiName String

4. The next three statements build the attribute list:

wsadmin>set name [list name QCF700]
name QCF700
wsadmin>set jndi [list jndiName jms/QCF700]
jndiName jms/QCF700
wsadmin>set mqcfattrs [list $name $jndi]
{name QCF700} {jndiName jms/QCF700}

5. Create the QCF:

wsadmin>$AdminConfig create WASQueueConnectionFactory $embeddedjmsp $mqcfattrs
QCF700(cells/myserver/nodes/myserver:resources.xml#WASQueueConnectionFactory_5)
wsadmin>$AdminConfig save

From the output of the statement that created the QCF, you can see your changes persisted in the resources.xml config file after you executed the save command.

If you want to create a QCF for the WMQ Provider, there are two changes to the above script:

1. First, change the provider name to Websphere MQ JMS Provider:

wsadmin>set myjmsp "WebSphere MQ JMS Provider"
wsadmin>set mqjmsp [$AdminConfig getid /Cell:$mycell/Node:$mynode/JMSProvider:$myjmsp/]
"WebSphere MQ JMS Provider(cells/myserver/nodes/myserver:resources.xml#builtin_boosted_mqprovider)"

2. Next, create an MQQueueConnectionFactory instead of a WASQueueConnectionFactory:

wsadmin>$AdminConfig create MQQueueConnectionFactory $mqjmsp $mqcfattrs
If you are using a third-party provider, set jmsprovider to the name you created for the third-party provider in the console. Or, create the third-party provider using wsadmin. Then, create a GenericJMSConnectionFactory object.

The next step is to create a Queue destination. Use the WebSphere (embedded) JMS provider. You can extrapolate the principles to the other two types of providers.

1. As with the QCF creation, setup the cell, node, and JMS provider variables:

```
wsadmin>set mycell [$AdminControl getCell]
myserver
wsadmin>set mynode [$AdminControl getNode]
myserver
wsadmin>set myjmsp "WebSphere JMS Provider"
wsadmin>set embeddedjmsp [$AdminConfig getid /Cell:$mycell/Node:$mynode/JMSProvider:$myjmsp/]
"WebSphere JMS Provider(cells/myserver/nodes/myserver:resources.xml#builtin_jmsp provider)
```

2. Find out the required attributes for the WASQueue object, and then populate the attribute list:

```
wsadmin>$AdminConfig required WASQueue
Attribute Type
name String
jndiName String
wsadmin>set name [list name EmbeddedJMSQ1]
name EmbeddedJMSQ1
wsadmin>set jndi [list jndiName jms/EmbeddedJMSQ1]
jndiName jms/EmbeddedJMSQ1
wsadmin>set wqAttrs [list $name $jndi]
{name EmbeddedJMSQ1} {jndiName jms/EmbeddedJMSQ1}
```

3. Finally, create the WASQueue and save the definition:

```
wsadmin>$AdminConfig create WASQueue $embeddedjmsp $wqAttrs
EmbeddedJMSQ1(cells/myserver/nodes/myserver:resources.xml#WASQueue_1)
wsadmin>$AdminConfig save
```

Next, add the queue to the Queue Names list of the Internal JMS Provider. In other words, the command-line equivalent of the step depicted in Figure 33. The script below is for Network Deployment and differs slightly for the WebSphere Application Server “base”:

```
set mynode [$AdminControl getNode]
set int_jms_serv [$AdminConfig getid /Node:$mynode/Server:jmsserver/JMSServer:JMSServer/]
$AdminConfig modify $int_jms_serv {{queueNames EmbeddedJMSQ1}}
$AdminConfig show $int_jms_serv queueNames
```
If you are using the Websphere MQ JMS Provider, create an object of type MQQueue under the WebSphere MQ JMS Provider. If using a generic JMS Provider, create an object of type GenericJMSDestination.

**Scripting for Management Tasks**

There are three JMX Mbeans in WebSphere Application Server 5 that allow you to administer JMS providers. You can access them through wsadmin or programmatically.

The JMSBasicFunction MBean works against embedded JMS provider and WMQ, and should work against third party providers if properly registered. It allows you to send and receive messages to and from and queue, and publish and subscribe messages to and from topics. Let's see how it works:

1. Start the WAS server and launch wsadmin.
2. Assign an alias to the JMSBasicFunction MBean:

   ```
   set jmsFunction [$AdminControl queryNames*:*,mbeanIdentifier=JMSBasicFunction]
   ```

3. View the possible operations, using the jmsFunction alias created in the prior step:

   ```
   $Help operations $jmsFunction
   ```

4. Send a message to a queue. You'll need to have already created the Queue and QueueConnectionFactory and know their names.

   ```
   $AdminControl invoke $jmsFunction send {{jms/MYQ} {jms/MYQCF} "Howdy"} {java.lang.String java.lang.String java.lang.String}
   ```

5. Receive a message from a queue.

   ```
   $AdminControl invoke $jmsFunction receiveFromQueue {{jms/MYQ} {jms/MYQCF}}
   ```

The JMS Administration MBean works against all JMS providers, and allows you to check if a queue is empty, browse a queue, and clear a queue. Let's try it out:

1. Start the server and launch wsadmin.
2. Assign an alias to the JMSAdministration MBean:

   ```
   set jmsAdmin [$AdminControl queryNames*:*,mbeanIdentifier=JMSAdministration]
   ```

3. View the possible operations.

   ```
   $Help operations $jmsAdmin
   ```

4. Browse a queue. You'll need to know the Queue and QCF names.

   ```
   $AdminControl invoke $jmsAdmin browse {{jms/MYQ} {jms/MYQCF}}
   ```
5. Clear a queue.

   $AdminControl invoke $jmsAdmin clear {{jms/MYQ} {jms/MYQCF}}

The EmbeddedJMSAdministration MBean, as the name suggests, provides administrative
operations that are specific to the embedded JMS provider. It allows you to obtain the depth of a
queue, administer the backout queue, and communicate with the broker to get
a list of durable subscriptions for the broker.

1. Start WebSphere Application Server and launch wsadmin.

2. Assign an alias to EmbeddedJMSAdministration MBean.

   set embeddedAdmin [$AdminControl queryNames
   *:* ,mbeanIdentifier=EmbeddedJMSAdministration]

3. View possible operations.

   $Help operations $embeddedAdmin

4. Browse the backout queue. SYSTEM.DEAD.LETTER.QUEUE is the backout queue for the
   embedded provider.

   $AdminControl invoke $embeddedAdmin browseBoQ

5. Clear the backout queue:

   $AdminControl invoke $embeddedAdmin clearBoQ

**JMS Security: Authentication Aliases**

Security for messaging resources is part of the WebSphere Application Server global security
framework. Access to a JMS resource is secured once you enable the Application Server global
security. JMS ConnectionFactory resources are accessed underneath the covers by Application
Server through the J2C framework. This has several implications for JMS resources.

First, when you create a JMS Connection Factory using the console or wsadmin, it becomes
visible in the JNDI namespace. An application can lookup the resource using a global JNDI
lookup, or the resource can be bound into the local namespace of an application using resource-
references). If you enable the Application Server global security, the ConnectionFactory has an
associated J2C authentication alias, which provides userid and password credentials that are
typically used for authentication to the underlying queue manager or broker.

In this section, let’s discuss the different types of authentication aliases, and how to set them up.
You need to consider this carefully because the type of authentication alias you use has
important implications on security. Because the resources are in the JNDI namespace, you need
to carefully guard access to them.

In the J2C resource security framework, there are two possible authentication modes:

- **Application** (also known as **component** in Application Server): If your application
  accesses a JMS resource through a resource reference in the deployment descriptor,
  and specifies a res-auth element of the application, then application authentication mode
  is used. This is also the authentication mode used implicitly if the resource’s global JNDI
  name is looked up in the code. With application authentication, if the application that tries
to access the Connection Factory specifies a userid and password in the code, those values are used to authentication the connection creation request. If the application does not specify a userid and password, and the ConnectionFactory has been configured with a component-managed authentication alias, the credentials in the configured alias is used. If there is no component-managed authentication alias defined, a security runtime exception results, and no credentials are passed on to JMS. This may or may not work, depending on the underlying provider. For example, WebSphere MQ may use process authentication in this case.

- **Container**: If your application’s deployment descriptor specifies a res-auth element of Container, container-managed authentication is used. With container-managed authentication, the deployer or administrator always handles the authentication configuration. The application code does not provide authentication information. If the ConnectionFactory has been configured with a container-managed authentication alias, the credentials in that alias are used. Otherwise, a runtime security exception occurs.

### Deciding on type of alias

Now, how do you decide which type of alias to use? If a resource uses a component alias, the resource is through JNDI by any application that has access to the JNDI namespace. Prior to WebSphere Application Server 5.0.3 (or 5.0.2 with APAR PQ76478), any client or server, even from outside the cell, is able to gain access to these resources. With the fix, only applications within the cell can gain access. This is certainly an improvement, but still leaves the JMS resources open to internal forms of attack. Obviously, this constitutes a security threat in a shared application environment.

If a resource uses a container-managed authentication alias, the application can access the resource only when a local resource-reference is specified in the deployment descriptor for that application. During application deployment time, the administrator has to bind this resource-reference to a specific set of credentials. Thus, container authentication is used only through an explicit action on the part of the administrator to bind the local resource reference to the authentication alias. On the other hand, the administrator cannot control or prevent authentication. If component aliases are defined, anyone with application access to JNDI can use it. Therefore, container aliases are far more secure than component aliases.

We have recommended container authentication, but there is a catch. In WebSphere Application Server 5.0.0 and 5.0.1, message-driven beans only support the use of component aliases. Therefore, if you are using JMS with 5.0.0 or 5.0.1, we strongly recommend that you obtain PTF (fixpack) 2, which includes the fixes necessary for MDBs to use container authentication aliases.

Now let’s look at how to configure container authentication aliases in both WebSphere Studio and WebSphere Application Server.

1. From the Server perspective in WebSphere Studio, double-click on your server to get to the configuration panel. Select the **Security** tab.
2. Enable global security. Select the **Enable security** checkbox as shown in Figure 55. Local OS Authentication refers to the operating system user registry, which is the default simple security registry available to you in WebSphere Studio.
3. Under Local OS Authentication, specify a user ID and password from the Windows user registry, which is a member of the Administrators group, and has the ‘Act as part of the operating system’ and ‘Log on as a service’ user rights. For simplicity, we will not Enforce Java 2 security at this time.
4. Under JAAS Authentication entries, click the **Add** button to specify a set of credentials. In the resulting Add JAAS Authentication Entry dialog box, enter
a name for the Alias (for example, mq_user), specify a user ID and password from the Windows registry that can access the messaging provider, and enter a description.

5. Click OK. The Authentication entry is in the table in the Figure 56.
6. Ctrl-S to save your changes.

Figure 55. Security tab of WebSphere Studio server configuration editor

<table>
<thead>
<tr>
<th>Security Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope: localhost=localhost/server1</td>
</tr>
</tbody>
</table>

Cell Settings

- Enable and setup security.
  - Enable security (Not supported on Windows 98 and Windows ME)
    - Local OS Authentication:
      - Server ID: myadminid
      - Server password: ****
      - Confirmed password: ****

- Enforce Java 2 security

JAAS Authentication Entries:

<table>
<thead>
<tr>
<th>Alias</th>
<th>User ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mq_user</td>
<td>mquser</td>
<td>User who can access mq</td>
</tr>
</tbody>
</table>

7. Now that the container alias has been created, the next step is to associate it with your ConnectionFactory.
8. Under the JMS tab of the test server settings panel, create a ConnectionFactory.
9. Under WASQueueConnectionFactory settings or MQConnectionFactorySettings (depending on your JMS provider), click Add. On the resulting Add WASQueueConnectionFactory dialog, enter the usual parameters and then set the container-managed authentication alias field to the name of the alias that you created earlier.
10. The final step is in your application. If your code accesses the ConnectionFactory directly, create a resource reference for the ConnectionFactory in your deployment descriptor. Specify an authentication type of container as shown in Figure 57. If the ConnectionFactory is accessed implicitly by the MDB’s listener port, it uses the container-managed authentication alias that you have configured, as long as you have the MDB security fixes from PTF2 as discussed earlier.
Overview of process in WebSphere Application Server console

Now that you’ve seen how to set up container aliases in WebSphere Studio, let’s go over the process in the Application Server console – it is similar.

1. Enable global security, under **Security -> Global Security**. You have a choice of the local operating system registry, LDAP, or a custom user registry. We will not go into further detail, refer to the WebSphere Application Server Infocenter to learn more on enabling security.
2. Navigate to **Security, JAAS Configurations, J2C AuthenticationData**.
3. Create a new J2C Authentication Data Entry as shown in Figure 58, specifying the user ID and password that can access the messaging provider.
4. Next, when you create the ConnectionFactory, associate it with the container alias as shown in Figure 59.

5. Finally, when you install the application, if it uses any ConnectionFactories other than in MDBs, bind those resource references to the newly created QCF. It is this tight administrator control that makes container aliases very secure.
JMS Security: Differences between Embedded Messaging and WMQ

So far we've talked about how to specify the userid and password for authentication. Now you need to understand how the authentication is performed. This differs based on your choice of messaging provider:

- If you are using Embedded Messaging, the userid and password are verified against the WebSphere Application Server security registry. Note you cannot use user IDs longer than 12 characters for authentication with the embedded JMS provider.
- If you are using full WMQ, in bindings mode there is no built-in userid/password authentication. The user ID and password that you specify on the authentication alias is ignored. This is because in bindings mode, WMQ traditionally expects the operating system to have authenticated the user. WMQ checks to see if the userid of the calling process (what Application Server runs as) can access WMQ. It should be in the appropriate mqm group.
- If you are using WMQ in client/server mode, there is no built in authentication either because WMQ relies upon process level authentication by default. However, the userid and password specified on the authentication alias are passed to WMQ. Normally, they are ignored, but you can write a WMQ security exit to access this information on the connection and verify it.

To authenticate with WMQ, you have three options:

- Rely on the existing WMQ process level authentication.
- With the advent of version 5.3, WMQ now supports SSL authentication. Therefore, you can configure its SSL support, which provides coarse-grained process-to-process level authentication along with encryption. The SSL certificates, presented by Application Server to WMQ, are used to perform this authentication.
• If you need fine-grained authentication that can distinguish between different applications in a shared environment, write a custom WMQ security exit for authentication. Security exits can provide granular per ConnectionFactory authentication.

Once authenticated (if applicable), authorization checks are performed to make sure the user can access the queue or topic requested. For embedded JMS, authorization to resources is controlled by the `config/integral-jms-authorizations.xml` file. For WMQ, you can use the traditional `setmqaut` command to control access to WMQ resources. In both cases, access to queues is based on queue authentication, and not on any J2EE identity.

Superficially, it may appear that Embedded Messaging has some advantages over WMQ in the security department. However, these are just that – superficial. Embedded Messaging does not support SSL, so the user ID and password may flow in the clear in multi-node JMS topologies. It also lacks other production quality features previously described that are unrelated to security.

**Third Party JMS Providers**

It is possible to configure WebSphere Application Server V5 to work with third-party generic JMS providers, such as SonicMQ, Tibco, and so on. To do so, you need to register the third-party provider to Application Server in the administrative console, under `Resources, Generic JMS Provider`. Once you have registered the provider, you can create the Connection Factories and Destinations. This procedure for registering a Generic JMS Provider and creating the necessary resources is out of scope for this article. Refer to [WebSphere Distributed Transaction Support for a Generic JMS Provider](#) on setting up third party JMS Providers.

**Conclusion**

This article provided a comprehensive guide to how WebSphere Application Server and WebSphere Studio support JMS using MQJD, WebSphere Embedded Messaging, and WebSphere MQ. You learned several topologies both in development and runtime scenarios. You also learned how to configure your WebSphere Studio for development under various JMS Providers and how to configure the WebSphere runtime, both base and ND. Finally, you learned about security considerations and how scripting can make JMS configurations easier.

**Acknowledgements**

The authors would like to thank Tom Alcott and Keys Botzum for their detailed review and comments; Matt Courtois, Susan Paice and Shahryar Sedghi for their contributions to the deployment topologies; and Bob Wehrle for his assistance with the wsadmin scripts.

**Related information**

[Creating and Testing Message Driven Beans using WebSphere Studio Application Developer 5.0](#)

[JMS Support in WebSphere Application Server V5 - Part 1: WebSphere Embedded Messaging](#)

[JMS Support in WebSphere Application Server V5 - Part 2: Deploy a P2P Application to WebSphere Embedded Messaging](#)

[MS Support in WebSphere Application Server V5 - Part 3: Message-Driven Beans in WebSphere Application Server V5](#)

[WebSphere Distributed Transaction Support for a Generic JMS Provider](#)

[JMS Troubleshooting](#)
About the authors

- **Roland Barcia** is a Consulting IT Specialist for IBM Software Services for WebSphere.
- **Sree Ratnasinghe** is an Advisory Software Engineer for IBM Software Services for WebSphere.
- **Benedict Fernandes** is a Consulting IT Specialist with IBM Software Services for WebSphere.