

# BEA WebLogic Integration



Integrating Talarian's SmartSockets for JMS product with the  
BEA WebLogic Server version 6.x

# Contents



<b>Contents</b>	<b>1</b>
<b>Introduction</b>	<b>2</b>
Background on Application Servers	2
Background on JMS	2
Business Reasons for Integration	3
Background on Message Driven Beans	3
<b>Product Integration</b>	<b>5</b>
What does WebLogic Integration Mean?	5
Bridging WebLogic JMS and SmartSockets for JMS	5
Configuration of SmartSockets for JMS and WLS for Coexistence	5
Using SmartSockets for JMS instead of WebLogic JMS	8
Running a SmartSockets MDB Inside WLS	8
<b>Conclusion</b>	<b>13</b>
<b>Appendix A: Sending from Within an MDB</b>	<b>14</b>
<b>Appendix B: Code Samples</b>	<b>16</b>

# Introduction

This white paper provides an overview of how BEA Weblogic Server and Talarian SmartSockets for JMS products can interoperate, the business advantages for doing so, and technical details on this integration. This white paper is targeted at managers and developers who wish to understand why they would want to integrate these two products, as well as consultants and developers who want step-by-step instructions on how to do so.

BEA's Weblogic Server (WLS) has become one of the most widely used application servers in the industry. WLS implements the J2EE standard, providing a means for multi-tier applications to be deployed in a distributed environment. Java Messaging Service (JMS) is a required component of J2EE, which provides a standard programming interface for publish/subscribe and message queuing products. JMS is the "SQL for messaging".

Talarian's SmartSockets is the best of breed publish/subscribe product, providing the highest performance, global scalability, robustness and proven deployment of any publish/subscribe product in the industry. SmartSockets for JMS provides a fully compliant JMS interface on top of Talarian's proven SmartSockets and SmartMQ publish/subscribe and queuing products. SmartSockets for JMS interoperates with Talarian's other products, allowing developers to both take advantage of the JMS standard, while also retaining the extended functionality and cross language support of SmartSockets.

BEA's WLS includes a basic JMS implementation, which is sufficient for many basic applications. For enterprise deployments where performance, global scalability, security, firewall tunneling, real-time fault tolerance, manageability, load balancing or cross language support are essential, many customers want to have these two best of breed products interoperate. Since BEA and Talarian are both compliant with the J2EE specifications, this is straightforward. More importantly, this integration has been tested at multiple customer sites.

## Background on Application Servers

An application server operates as a middle-tier between the client and back-end services such as databases. Enterprise Java Beans (EJBs) can be loaded into the application server in much the same way that a stored procedure can be loaded in to a database, or a server extension module in to a SmartSockets RT server. Low-level issues such as multithreading, transaction management, state management and resource pooling are handled by the application server while business logic and application specific code are executed within EJBs.

A multi-tier J2EE application consists of several components that operate as one software unit. For example, an application might consist of a Java client component, an EJB component and a back-end database component. These three components function as a single application and would be viewed as such from the viewpoint of a SmartSockets messaging server cloud.

## Background on JMS

Java Messaging Service (JMS) provides the first industry standard API for message oriented middleware. Over the past decade, thousands of the world's top corporations have deployed message-based distributed applications for many of their most revenue-critical applications. A class of communication software called message-oriented middleware (MOM) has supported these applications. There are two basic types of messaging software: message queuing and publish/subscribe. Both provide the same asynchronous, 'fire-and-forget' model of communication. Message queuing provides reliable message delivery for transaction processing systems, and is used primarily for non-real time delivery of messages to a single receiver. It is

typically used where messages have to be delivered to a single destination, as in financial transactions. Publish/subscribe software, by contrast, provides delivery to more than one receiver at a time. It is particularly useful when many receivers need the same data, where high performance is required, or the data has to be delivered in real time.

MOM products have in the past been primarily deployed using proprietary interfaces, rather than standardized web protocols. This has caused problems for customers with vendor lock-in, and has impeded developers from being able to transfer their skills from one messaging product to another. JMS helps solve these issues, by providing a standard API that covers a wide range of message queuing and publish/subscribe products. It is a first generation API, and so developers and managers still need to evaluate vendor's products for support of essential extensions such as cross-language support, security, and manageability. JMS does not specify implementation details, and so there can be wide disparity among products in terms of global scalability, performance, fault-tolerance, security, advanced features, support, and price.

## Business Reasons for Integration

The market for JMS implementations currently consists of two tiers of products—pure JMS and enterprise JMS products. A number of vendors, including BEA, Progress and Fiorano, have created a native JMS implementation, with both the client libraries and server infrastructure written in Java. These pure JMS implementations provide all of the basic functionality of the JMS specification, on top of a single server or simple replicated server architecture. Their primary advantages are price, simplicity, and in the case of BEA, integrated bundling with the WebLogic Server. Talarian and IBM are examples of enterprise JMS vendors, providing a JMS interface to their traditional MOM infrastructure. IBM's MQSeries is usually acknowledged as the best of breed queuing product, and Talarian's SmartSockets is usually acknowledged as the best of breed publish/subscribe product. Enterprise JMS products typically have their client libraries written in Java, while their server infrastructure is highly optimized and deployment hardened C/C++ code. They offer extended functionality, higher levels of performance and scalability, and cross language support. They are typically more expensive, and their advanced capabilities are usually still dependent on proprietary extensions, though this should be reduced as the JMS specification evolves.

Talarian's SmartSockets for JMS product supports both queuing and publish/subscribe, and provides a range of advanced functionality such as security, load balancing, firewall tunneling, detailed management and monitoring instrumentation, on the fly configuration, real-time fault tolerance, and multicast capabilities. It provides the highest performance in the industry, with customers such as the American Stock Exchange and Philadelphia Stock Exchange requiring 72,000 messages per second. It also provides the only globally scalable solution, with customers like Micron Technologies running their semiconductor manufacturing operations on it, involving 20,000 worldwide computers over 10 countries, running 24x7x365. It is a very available and robust solution, running not only extremely demanding applications like the New York Stock Exchange, but also being shipped to hundreds of thousands of computers through 40 OEM customers like Micromuse, BMC, Aspect, ABB, DoubleClick, Novell, and Computer Associates. It supports C, C++, ActiveX, VisualBasic, COM, Java, and JMS client libraries. It can interoperate seamlessly with other products such as IBM MQSeries, JDBC databases, Talarian's SmartCache real-time cache, and many others.

## Background on Message Driven Beans

As mentioned above, Enterprise Java Bean's (EJB's) are the managed objects that execute presentation, business or application logic, inside of a J2EE application server such as BEA WebLogic Server. An EJB can make procedure calls to any Java library, but its event notifications and interrupt mechanisms are managed by the J2EE server.

Prior to the JMS standard, EJB's came in two flavors: session beans and entity beans. A session bean is created inside the application server when a client connects and is destroyed when the client disconnects. An entity bean interacts primarily with a database. Both of these types of beans are request-reply in nature and are not capable of receiving JMS messages in an event-driven manner. Along with the addition of

JMS, the J2EE standard has further been expanded to include a new kind of EJB called a message-driven bean (MDB). An MDB is essentially a session bean with the addition of a JMS message listener for receiving messages asynchronously.

Message driven beans are important for the many applications that need to use both client-server and message based computing. Message based computing allows asynchronous operation, the ability for both clients and servers to generate messages, fire and forget delivery, one to many delivery, and loose coupling of applications. It is particularly superior in real-time applications, environments where many applications are being integrated together, or in cases where more than one receiver needs to receive a message.

# Product Integration

## What does WebLogic Integration Mean?

Customers who want to integrate BEA's Weblogic Server with Talarian's SmartSockets for JMS typically want to exclusively use the Talarian JMS implementation as an enterprise-class replacement for the JMS implementation shipped with WLS.

Existing BEA customers may also want to create a "bridge" between WLS's JMS and another JMS provider such as Talarian. This document explains how to support both of these options.

## Bridging WebLogic JMS and SmartSockets for JMS

Some BEA Customers, especially those who are currently using WLS JMS in production, might want to continue using WLS for some applications and begin using SmartSockets for JMS for other applications. In theory, this would not be necessary because a JMS application should be able to talk to any JMS provider. However, real world factors and/or deadlines may call for an interim solution that necessitates a bridge between WLS JMS and SmartSockets for JMS.

This can be accomplished by developing a standard Java Application that creates both WLS JMS and SmartSockets for JMS connection factories. Connections, sessions, queues, producers, and consumers corresponding to each provider can be created in order to send and receive messages via both JMS providers. The following section is a tutorial for configuring WLS JMS and SmartSockets JMS to coexist in this manner.

### Configuration of SmartSockets for JMS and WLS for Coexistence

The WebLogic integration examples are packaged into the zip file `WebLogicIntegrationExamples.zip` for distribution. Please unzip these files into the desired location on your machine. The distribution contains two folders: `bridge`, `mdb`, and `mdbsend`. The `bridge` directory contains the files needed for this example. They are as follows:

- `JMSObject.class` – a compiled class defined in `t.java`
- `t.class` – compiled main class of `t.java`
- `t.java` – source code for the JMS bridge example

In order to run this example, you first need to download and install WebLogic Application Server (WLS) 6.x. For this tutorial, WLS 6.1 is recommended. Please refer to the following URL:

[http://commerce.beasys.com/downloads/weblogic\\_server.jsp](http://commerce.beasys.com/downloads/weblogic_server.jsp)

You can install WLS in `C:\bea\wlserver6.x` to be consistent with this tutorial or in another location of your choosing.

Next, you need to download and install Talarian SmartSockets for JMS 1.1. Please go to <http://www.talarian.com> and select "Download, SmartSockets for JMS 1.1 – for Windows NT" (save the self-extracting exe file to your hard drive, and install in `C:\Program Files\Talarian\`). Shortly after downloading SmartSockets for JMS, a Talarian representative will contact you and provide

you with temporary licenses for both SmartSockets 6.x and SmartMQ 2.1, both of which are required to run SmartSockets for JMS.

You may want to use a JDK other than the JDK that comes with WLS 6.x (JDK 1.3). You can download the latest JDK from <http://java.sun.com>.

Now is a good time to start the Talarian MQserver, which is the server shipped with SmartMQ 2.1, necessary for point-to-point (message queuing) communication between JMS clients. To start MQserver, from the Start Menu, select Programs ? SmartSockets for JMS ? SmartMQ ? MQserver. You will need MQserver up and running in order to run the example.

You may also want to start the Talarian RTserver, which is the server shipped with SmartSockets 6.x, necessary for keeping track of JMS topics and routing JMS messages based on those topics. To start RTserver, from the Start Menu, select Programs ? SmartSockets for JMS ? SmartSockets ? RTserver to start the RTserver. Although the first example involves queuing only and doesn't use RTserver, it is a good idea to get in the habit of starting RTserver. Also, if you happen to be using SS for JMS 1.0 (actually called Workbench for JMS) instead of SS for JMS 1.1, then you need to have RTserver running even if you're not using topics.

You will also need to make some JMS queues for sending and receiving messages. You can do this using the SmartSockets for JMS administration tool, which has been provided in order to create and maintain topics, queues, topic connection factories, and queue connection factories for your JMS applications. To start the SmartSockets for JMS Admin tool, from the Start Menu, select Programs ? SmartSockets for JMS ? JMS ? SSJMS Admin. Under Point-to-Point, create a QueueConnectionFactory called "myQCF" and a Queue called "myQueue", giving it a SmartMQ name of "mqs-default-local-queue". This means that the JMS Queue name of myQueue will map to the SmartMQ name of mqs-default-local-queue.

You should also change the provider URL defined in the `jndi.properties` file that comes with SS for JMS. The `jndi.properties` file is located in `C:\Program Files\Talarian\jms11\lib`. Assuming that you have installed JMS in the default install location, the entry should read:

```
java.naming.provider.url=file://localhost/C:/PROGRA~1/Talarian/jms11/bin/ndings
```

WLS cannot find the SS for JMS JNDI if the `“//localhost/”` portion of the provider URL isn't there. If it isn't there, insert it and save the file. In later releases of SS for JMS, it will be there by default.

Next you should open up a command prompt to run the test application (`t.java`). This command prompt window will need to have the environment of both WLS 6.x and SmartSockets for JMS. There are at least two ways to do this: start with a WebLogic environment and set the SS for JMS environment or start with an SS for JMS environment and set the Weblogic environment.

To get the SS for JMS environment settings, it is convenient to simply open up an SSJMS command prompt. From the Start Menu, select Programs ? SmartSockets for JMS ? JMS ? SSJMS Command Prompt.

To get the WebLogic environment settings, it is convenient to use `setEnv.cmd`, located in `C:\bea\wlserver6.1\config\mydomain`. However, beware that `setEnv.cmd` destructively sets the `CLASSPATH`, overwriting its previous value. You can modify your `setEnv.cmd` to non-destructively set the `CLASSPATH`, by appending `“;%CLASSPATH%”` to the end of the `SET CLASSPATH` statement in `setEnv.cmd`. Alternatively, you can set the SS for JMS environment after the WLS environment has been set. You can type the following to set the SS for JMS environment manually or add it to `setEnv.cmd`:

```
C:\PROGRA~1\Talarian\jms11\bin\i86_w32\JMSVAR~1.BAT
```

This assumes that you are using SS for JMS 1.1 and that you have it installed in the default location.

After you've set up both environments, you're ready to compile.

From this command prompt you can compile the supplied `t.java` by typing:

```
javac t.java
```

If SmartSockets for JMS has been installed somewhere other than `C:\Program Files\Talarian`, you will need to edit the `MQurl` entry in `t.java` to reflect the location of SmartSockets for JMS and recompile.

You will need to start WLS 6.x by either finding it in the start menu or starting it from the command prompt. To start WLS 6.x from a command prompt, open a command prompt window, change to `C:\bea\wlserver6.1\config\mydomain\` and type:

```
StartWeblogic.cmd
```

**Note: You must be in the mydomain directory to start WebLogic.**

Now go back the other command prompt window where you compiled `t.java` and run it by typing:

```
java t
```

You should see something similar to the following output:

```
Sending the message on queue mqs-default-local-queue
Receiving the message on queue mqs-default-local-queue
Received message: type = jms_text
sender = null
dest = null
max = n/a
size = 32
current = 0
read only = false
priority = 0
delivery_mode = some
ref count = 1
seq num = 0
resend mode = false
user prop = 0
message id = 000000000000000000@{e13faf3d-d458-076f-f68d-a8a5591fe43a}
correlation id = fix
sender timestamp = Fri Sep 14 15:32:51 CDT 2001
data (num_fields = 1):
str "Test String"
```

```
Sending the message on queue TestQueue1
Receiving the message on queue TestQueue1
Received message:
TextMessage[id=ID:N<912871.1000499572536.0>,text=Test String]
Sending the message on queue mqs-default-local-queue
Receiving the message on queue mqs-default-local-queue
Received message: type = jms_text
sender = null
dest = null
```

```

max = n/a
size = 32
current = 0
read only = false
priority = 0
delivery_mode = some
ref count = 1
seq num = 0
resend mode = false
user prop = 0
message id = 0000000000000001@{e13faf3d-d458-076f-f68d-a8a5591fe43a}
correlation id = fix
sender timestamp = Fri Sep 14 15:32:52 CDT 2001
data (num_fields = 1):
str "Test String"

```

## Using SmartSockets for JMS instead of WebLogic JMS

Most Talarian customers using BEA Weblogic will want to exclusively use SmartSockets for JMS as the JMS provider for WebLogic Server. In the case of a standard Java application such as the bridge example from the previous section, this can be accomplished by simply creating only a SmartSockets for JMS connection factory instead of creating both types of connection factories. It is also possible for an EJB loaded inside of WebLogic server to replace WLS JMS by creating a SmartSockets for JMS connection factory instead of a WLS JMS connection factory.

The following example shows you how to compile and deploy your own message-driven bean (MDB) in WLS that has its own message listener for asynchronously receiving messages from SmartSockets for JMS.

### Running a SmartSockets MDB Inside WLS

The WebLogic integration examples are packaged into the zip file `WebLogicIntegrationExamples.zip` for distribution. Please unzip these files into the desired location on your machine if you haven't already done so. The distribution contains two folders: `bridge`, `mdb`, `mdbsend`. The `mdb` directory contains the files needed for this example. They are as follows:

```

build.bat - script used for building the MDB
ejb-jar.xml - used for building MDB.jar
weblogic-ejb-jar.xml - used for building MDB.jar
MDB.java - source code for the MDB
send.class - standard Java App used for sending messages point-to-point to the MDB
send.java - source code for send App
publish.class - standard Java App used for sending messages pub-sub to the MDB
publish.java - source code for publish App
MDB.jar - the built MDB that WLS will be instructed to load on start up
Ejbcgen - a directory for intermediate files that build.bat uses
Build - a directory of intermediate files that build.bat uses

```

In order to run this example, you first need to download and install WebLogic Application Server (WLS) 6.x. For this tutorial, WLS 6.1 is recommended. Please refer to the following URL:

[http://commerce.beasys.com/downloads/weblogic\\_server.jsp](http://commerce.beasys.com/downloads/weblogic_server.jsp)

You can install WLS in `C:\bea\wlserver6.1` to be consistent with this tutorial or in another location of your choosing. If you are using WLS 6.0 sp2 instead of WLS 6.1, you will also need to install the EJB 2.0 upgrade for WebLogic 6.0. To download the upgrade, please go to the following URL:

[http://commerce.beasys.com/downloads/weblogic\\_server.jsp#wls](http://commerce.beasys.com/downloads/weblogic_server.jsp#wls)

Under WebLogic Server 6.0 where it says “Modules for WebLogic Server 6.0”, you can select the EJB 2.0 upgrade. The file `ejb20.zip` includes an `ejb20.jar` file that needs to be added to the CLASSPATH for Weblogic server. Again, if you are using WLS 6.1 you don’t need to do this.

Next, you need to download and install Talarian SmartSockets for JMS 1.1. Please go to <http://www.talarian.com> and select “Download, SmartSockets for JMS 1.1 – for Windows NT” (save the self-extracting exe file to your hard drive, and install in `C:\Program Files\Talarian\`). Shortly after downloading SmartSockets for JMS, a Talarian representative will contact you and provide you with temporary licenses for both SmartSockets 6.x and SmartMQ 2.1, both of which are required to run SmartSockets for JMS.

You may want to use a JDK other than the JDK that comes with WLS 6.x (JDK 1.3). You can download the latest JDK from <http://java.sun.com>.

Now that everything is installed, you are ready to start the middleware components. You should start Talarian MQserver, which is the server shipped with SmartMQ 2.1, necessary for point-to-point (message queuing) communication between JMS clients. To start MQserver, from the Start Menu, select Programs ? SmartSockets for JMS ? SmartMQ ? MQserver. You will need MQserver up and running in order to run the example.

In order to use publish-subscribe, you should also start the Talarian RTserver, which is the server shipped with SmartSockets 6.x, necessary for keeping track of JMS topics and routing JMS messages based on those topics. To start RTserver, from the Start Menu, select Programs ? SmartSockets for JMS ? SmartSockets ? RTserver to start the RTserver. You will need RTserver up and running in order to run the example.

Just as a side-note, it is not always necessary to have both middleware components up if you are only using the features of one of them. For example if you are only using point-to-point, then you only need MQserver up and running. Likewise if you are only using publish-subscribe, then you only need RTserver up and running. This was not the case with SS for JMS 1.0 (formerly Workbench for JMS), which always required both middleware components to be up and running.

You also need to make some JMS queues and topics for sending and receiving messages. You can do this using the SmartSockets for JMS administration tool, which has been provided in order to create and maintain topics, queues, topic connection factories, and queue connection factories for your JMS applications. To start the SmartSockets for JMS Admin tool, from the Start Menu, select Programs ? SmartSockets for JMS ? JMS ? SSJMS Admin. Under `Point-to-Point`, create a `QueueConnectionFactory` called “myQCF” and a `Queue` called “myQueue”, giving it a SmartMQ name of “`mqs-default-local-queue`”. This means that the JMS Queue named `myQueue` will map to the SmartMQ queue name `mqs-default-local-queue`. Now under `Publish-Subscribe`, create a `TopicConnectionFactory` called “myTCF” and a `Topic` called “myTopic”, setting its SmartSockets subject name to something like “`/mytopic`”. This means that the JMS Topic named `myTopic` will map to the SmartSockets subject name `/mytopic`.

Whenever WLS starts up, it parses an XML file called `config.xml` located in `C:\bea\wlserver6.1\config\mydomain`. This file is how we tell WebLogic server to load our MDB. Optionally, you can configure WLS through its console GUI, which will automatically update the `config.xml` file for you. See the WLS documentation for more information. Assuming that your MDB is located at `c:\src\jmsintegration`, add the following text to `config.xml` before the final `</Domain>` line.

```
<Application Name="MDB" Path="c:\src\jmsintegration">
```

```
<EJBComponent Name="MDB" URI="MDB.jar" Targets="myserver"/>
</Application>
```

**Note: Sometimes WebLogic 6.1 moves these XML entries around automatically. No need to be alarmed – this is perfectly normal.**

If you are using WLS 6.0 sp2 instead of WLS 6.1, there is an important difference in the syntax of the `ejb-jar.xml` file (located where you unzipped the files). The field `destination-type` used to be called `jms-destination-type`, hence the line where `javax.jms.Queue` is defined should read as follows:

```
<jms-destination-type>javax.jms.Queue</jms-destination-type>
```

Instead of the following:

```
<destination-type>javax.jms.Queue</destination-type>
```

Also, the line where a `javax.jms.Topic` is defined should read as follows:

```
<jms-destination-type>javax.jms.Topic</jms-destination-type>
```

Instead of the following:

```
<destination-type>javax.jms.Topic</destination-type>
```

Again, if you are using WLS 6.1, you won't have to change `ejb-jar.xml`.

You may also want to check the fields in `weblogic-ejb-jar.xml` (located wherever you unzipped the files) to make sure everything is correct. For example, if SmartSockets for JMS is not located in `C:\Program Files\Talarian` on your machine, you will need to change the entry for `<provider-url>` in `weblogic-ejb-jar.xml` to reflect the location of SmartSockets for JMS. You can also change the name of the queue to use for sending messages. In order for changes in this file to take effect, you must rebuild the MDB (using `build.bat`).

You should also change the provider URL defined in the `jndi.properties` file that comes with SS for JMS. The `jndi.properties` file is located in `C:\Program Files\Talarian\jms11\lib`. Assuming that you have installed JMS in the default install location, the entry should read:

```
java.naming.provider.url=file://localhost/C:/PROGRA~1/Talarian/jms11/bi
ndings
```

WLS cannot find the SS for JMS JNDI if the `“//localhost/”` portion of the provider URL isn't there. If it isn't there, insert it and save the file. In later releases of SS for JMS, it will be there by default.

Before compiling the MDB, make sure that both SmartSockets for JMS and WebLogic environments are present. There are at least two ways to do this: start with a WebLogic environment and set the SS for JMS environment or start with an SS for JMS environment and set the Weblogic environment.

To get the SS for JMS environment settings, it is convenient to simply open up a SSJMS command prompt. From the Start Menu, select Programs? SmartSockets for JMS? JMS? SSJMS Command Prompt.

To get the WebLogic environment settings, it is convenient to use `setEnv.cmd`, located in `C:\bea\wlserver6.1\config\mydomain`. However, beware that `setEnv.cmd` destructively sets the `CLASSPATH`, overwriting its previous value. You can modify your `setEnv.cmd` to non-destructively set the `CLASSPATH`, by appending `“;%CLASSPATH%”` to the end of the `SET CLASSPATH`

statement in `setEnv.cmd`. Alternatively, you can set the SS for JMS environment after the WLS environment has been set. You can type the following to set the SS for JMS environment manually or add it to `setEnv.cmd`:

```
C:\PROGRA~1\Talarian\jms11\bin\i86_w32\JMSVAR~1.BAT
```

This assumes that you are using SS for JMS 1.1 and that you have it installed in the default location.

After you've set up both environments, you're ready to compile the MDB. Make sure that `javac.exe` is in your path and from the command prompt type:

```
build.bat
```

In order to start WebLogic Server, once again you'll need both environments set up. This time the command file that invokes WLS sets the environment for you. This command file, `StartWebLogic.cmd`, is located in `C:\bea\wlserver6.1\config\mydomain`. Like with `setEnv.cmd`, you have the choice of either modifying this file to non-destructively set the `CLASSPATH` or modifying it to set the SS for JMS environment after the `CLASSPATH` has been set.

If you chose to edit `StartWebLogic.cmd` to non-destructively set the environment, open up an SS for JMS window; otherwise any command prompt window will do. From the command prompt, make sure that `c:\bea\wlserver6.1\bin` is in the path. If it's not, then add it by typing the following at the prompt:

```
set PATH=%PATH%;c:\bea\wlserver6.1\bin
```

Change to `C:\bea\wlserver6.1\config\mydomain` and type "StartWeblogic" and press Enter. After typing in your password in the command prompt window, you should see something like the following:

```
Starting WebLogic Server ....
<Oct 22, 2001 3:29:28 PM MDT> <Notice> <Management> <Loading configuration file
.\config\mydomain\config.xml ...>
log file: C:\bea\wlserver6.0\.\config\mydomain\logs\weblogic.log
<Oct 22, 2001 3:29:30 PM MDT> <Info> <Logging> <Only log messages of severity "E
rror" or worse will be displayed in this window. This can be changed at Admin Co
nsole> mydomain> Servers> myserver> Logging> General> Stdout severity threshold>

<Oct 22, 2001 3:29:43 PM MDT> <Notice> <WebLogicServer> <WebLogic Server started
>
<Oct 22, 2001 3:29:43 PM MDT> <Notice> <WebLogicServer> <ListenThread listening
on port 7001>
<Oct 22, 2001 3:29:43 PM MDT> <Notice> <WebLogicServer> <SSLListenThread listeni
ng on port 7002>
```

If you are having trouble loading the MDB, please contact technical support.

Now that WebLogic is started and the MDB is successfully loaded, go back to your other command prompt window and type "java send" at the command prompt to send messages to the MDB via point-to-point. You can type "java publish" to send messages to the MDB via publish-subscribe. If for some reason you need to rebuild `send.java` or `publish.java`, you can do so using the same environment you used to build the MDB.

If you typed "java send" the application should respond with the following:

```
Sending the message on queue mqs-default-local-queue
```

If you typed "java publish" the application should respond with the following:

```
Attempting connection to <tcp:_node:5101> RTserver.  
Connected to <tcp:_node:5101> RTserver.  
Sending the message on topic /mytopic
```

Now look at the other command prompt window where WLS is running. In the command prompt window, WLS prints the following:

```
MDB: hello world
```

This tells us that the MDB has received a message whose contents are “hello world”.

# Conclusion

BEA's Weblogic Server is a best of breed J2EE application server, and Talarian's SmartSockets for JMS is a best of breed enterprise JMS product. This white paper has demonstrated how these products can interoperate, some of the business reasons for doing so, and provided technical details that should be sufficient to allow customers to try out the integration of these products themselves.

While a basic JMS product will meet the needs of many applications, for high end enterprise applications, an enterprise JMS product will likely be required. Talarian's SmartSockets product line provides the highest performance, global scalability, robustness and proven deployment of any publish/subscribe product in the industry. It also provides a set of advanced features not yet in the JMS specification, such as cross language support, firewall tunneling, security, real-time fault tolerance, detailed management and monitoring capabilities, on-the-fly server configuration, load balancing, adaptive multicast, and other features. For more details on the features and benefits of SmartSockets, please see <http://www.talarian.com/products/smartsockets/index.shtml>

If you have any questions, or would like more information, please feel free to contact either [sales@talarian.com](mailto:sales@talarian.com), Brian Whetten (Chief Scientist, [whetten@talarian.com](mailto:whetten@talarian.com)), or Jaymes Wilks (Consultant, [jwilks@talarian.com](mailto:jwilks@talarian.com))

# Appendix A: Sending from Within an MDB

Due to the fact that certain parties have expressed interest in sending messages from within MDBs, this appendix has been added as a tutorial for those interested. Interested individuals are encouraged to seek out other sources for guidance in designing solutions that utilize EJBs. There are some great EJB examples on <http://java.sun.com/> that demonstrate how MDBs can work together with session beans and entity beans in a Java enterprise solution. For example, chapter 8 of the JMS tutorial on Sun's website, entitled "A J2EE Application that Uses the JMS API with a Session Bean," shows how to write a client app and a session bean that sends messages to a message-driven bean through JMS. There is also another example that demonstrates how entity beans can interact with MDBs.

For most designs, it is sufficient to only have MDBs receive JMS messages. Even if an MDB needs to send information via JMS, a non-message-driven bean, tightly coupled with an MDB, can deliver the information on behalf of the MDB. However, some situations may not demand such an elaborate solution. The purpose of this tutorial is to demonstrate that an MDB can send JMS messages just like any other EJB.

With the addition of a couple more steps, this example is executed in much the same way as the non-sending MDB example is executed. You should set up that example first before attempting this one. The files for this tutorial are located in the `mdbsend` directory of the `WebLogicIntegrationExamples.zip` distribution. This directory has mostly the same files as the `mdb` directory with the contents of some of them changed. First of all, `MDB.java` now has code for sending JMS messages inside the message listener callback. Also, the `.xml` files have been modified so that the MDB will only receive messages via publish-subscribe. Furthermore, `publish.java` has been modified to wait for a reply message from the MDB. Finally, `send.java` is no longer present because we are not using point-to-point JMS in this example.

In order to run this example, you will need to open up the SSJMS Admin tool and create a new Topic called "myReplyTopic" and set the SmartSockets subject to something like "/myreplytopic". This provides a back channel for the MDB to publish reply messages to. You will also need to edit the Application field for the MDB in the `config.xml` file (located in `C:\bea\wlserver6.1\config\mydomain`) to reflect the path of the `ejb.jar` in the `mdbsend` directory. After doing these two things, you can compile and run the example just as was done for the non-sending MDB example.

Now try starting WebLogic with the loaded MDB and running the `publish` app to see what happens. In the `publish` app's console window, you should see something like this:

```
Attempting connection to <tcp:_node:5101> RTserver.  
Connected to <tcp:_node:5101> RTserver.  
Sending the message on topic /mytopic  
Receiving the message on topic /myreplytopic  
Message: reply
```

Looking in your WebLogic console window, you should see:

```
MDB: hello world
```

Sending the relpy message on topic /myreplytopic

If the message didn't appear make it to the MDB, it may be because the message was never flushed from the client. In order to achieve higher throughput rates, SmartSockets does not flush the client's message buffer after each message but instead flushes the buffer automatically when it reaches a certain size. This option is called `auto_flush_size`. You can set this option from the SSJMS Admin tool by looking under `Publishing-Subscribe` and `TopicConnectionFactory` and selecting `myTCF`, which you created earlier, and clicking on the `Advanced` tab. Scan down the list of options until you find `ss.auto_flush_size` and set the option to the value 0. This causes the client's message buffer to be flushed each time you send a message using `myTCF`. You shouldn't get too comfortable with doing this as it slows the performance.

# Appendix B: Code Samples

## The Bridge Example – t.java

```
import javax.jms.*;
import javax.naming.*;
import javax.naming.directory.*;
import java.util.Hashtable;

public class t
{
    public static final String MQqcf = "myQCF";
    public static final String WLSqcf =
"javax.jms.QueueConnectionFactory";

    public static final String MQqname = "myQueue";
    public static final String WLSqname = "jms.queue.TestQueue1";

    public static final String MQurl =
"file://localhost/c:/PROGRA~1/Talarian/jms11/bindings";
    public static final String WLSurl = "t3://localhost:7001";

    public static final String MQJNDIfactory =
    "com.sun.jndi.fscontext.RefFSContextFactory";
// Talarian Context Factory
    public static final String WLSJNDIfactory =
    "weblogic.jndi.WLInitialContextFactory";
// Weblogic Context Factory

    public static void main(String[] args) {
        JMSObject MQobject = null;
        JMSObject WLSobject = null;
        TextMessage msg;
        Context ctx;

        try {
// Create the Talarian connection factory, connection, session,
// queue
            MQobject = new JMSObject(MQurl, MQJNDIfactory, MQqcf, MQqname);

// Create the WLS connection factory, connection, session,
// queue
            WLSobject = new JMSObject(WLSurl, WLSJNDIfactory, WLSqcf,
WLSqname);

            msg = MQobject.JMSMessage("Test String");
```

```

MQObject.JMSSend(msg);
msg = MQObject.JMSReceive();
System.out.println("Received message: "+msg);

WLSubject.JMSSend(msg);
msg = WLSubject.JMSReceive();
System.out.println("Received message: "+msg);

MQObject.JMSSend(msg);
msg = MQObject.JMSReceive();
System.out.println("Received message: "+msg);

} catch(JMSEException je)
{
System.out.println("Caught JMSEException: "+je);
Exception le = je.getLinkedException();
if (le != null) System.out.println("Linked exception: "+le);
je.printStackTrace();
} catch(Exception e) {
e.printStackTrace();
System.out.println("Caught Exception: "+e);
} finally {
try {
if (MQObject != null)
MQObject.JMSCleanup();
if (WLSubject != null)
WLSubject.JMSCleanup();
} catch (Exception e) { }
}
}
}

class JMSObject {
private Queue ioQueue;
private QueueSession session;
private QueueConnection connection;
private QueueConnectionFactory factory;
private QueueSender queueSender;
private QueueReceiver queueReceiver;
private InitialContext ctx;

JMSObject(String url, String jndi, String qcf, String qname)
throws Exception {

Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, jndi);

if (url != null)
{
env.put(Context.PROVIDER_URL, url);
env.put(Context.REFERRAL, "throw");
}

ctx = new InitialContext(env);

factory = (QueueConnectionFactory)ctx.lookup(qcf);

```

```

    // Create a QueueConnection, QueueSession
    connection = factory.createQueueConnection();
    session = connection.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);

    ioQueue = (Queue)ctx.lookup(qname);

    connection.start();

    queueSender = session.createSender(ioQueue);
    queueReceiver = session.createReceiver(ioQueue);
}

TextMessage JMSMessage(String text) throws Exception {
    TextMessage msg = session.createTextMessage();
    msg.setText(text);
    return(msg);
}

void JMSSend(TextMessage msg) throws Exception {
    System.out.println("Sending the message on queue " +
ioQueue.getQueueName());
    // Following three lines for WLS 5.0 and 6.0 (no service pack)
    msg.setJMSTDestination(null);
    // code around WLS bug - CR042458
    if (msg.getJMSCorrelationID() == null)
        msg.setJMSCorrelationID("fix");
    // code around WLS bug - CR042461
    queueSender.send(msg);
}

TextMessage JMSReceive() throws Exception {
    TextMessage msg;
    System.out.println("Receiving the message on queue " +
ioQueue.getQueueName());
    msg = (TextMessage)queueReceiver.receive(1000);

    if (msg == null)
        throw new JMSEException("Failed to receive message");
    return(msg);
}

void JMSCleanup() throws Exception {
    if (session != null) {
        session.close();
        session = null;
    }
    if (connection != null)
    {
        connection.close();
        connection = null;
    }
}
}

```

## The MDB Example – mdb.java

```
import javax.ejb.CreateException;
import javax.ejb.MessageDrivenBean;
import javax.ejb.MessageDrivenContext;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

public class MDB implements MessageDrivenBean, MessageListener {
    private MessageDrivenContext context;

    // Required - public constructor with no argument
    public MDB() {}

    // Required - ejbActivate
    public void ejbActivate() {}

    // Required - ejbRemove
    public void ejbRemove() {
        context = null;
    }

    // Required - ejbPassivate
    public void ejbPassivate() {}

    public void setMessageDrivenContext(MessageDrivenContext mycontext) {
        context = mycontext;
    }

    // Required - ejbCreate() with no arguments
    public void ejbCreate () throws CreateException {}

    // Implementation of MessageListener - throws no exceptions
    public void onMessage(Message msg) {
        try {
            System.out.println("MDB: " + ((TextMessage)msg).getText());
        }
        catch(Exception e) { // Catch any exception
            e.printStackTrace();
        }
    }
}
```

TALARIAN CORPORATE HEADQUARTERS  
333 Distel Circle  
Los Altos, CA 94022-1404  
(800) 883-8050  
FAX (800) 883-8057

TALARIAN LIMITED  
68 Lombard Street  
London EC3V 9LJ  
+44 (0) 20 7868 1630  
FAX +44 (0) 20 7868 1752

E-Mail [info@talarian.com](mailto:info@talarian.com)  
[www.talarian.com](http://www.talarian.com)

