

HP Total-e-Server Integration



Integrating Talarian's SmartSockets for JMS product with the
HP Total-e-Server version 7.3

Contents



Contents	1
Introduction	2
Background on Application Servers	2
Background on JMS	2
Business Reasons for Integration	3
Background on Message Driven Beans	3
Product Integration	5
What does HP Total-e-Server Integration Mean?	5
Using SmartSockets for JMS as the JMS provider for TeS	5
Appendix A: Sending from Within an MDB	10
Appendix B: Code Samples	12

Introduction

This white paper provides an overview of how the HP Total-e-Server and Talarian SmartSockets for JMS products can interoperate, the business advantages for doing so, as well as 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.

HP's Total-e-Server (TeS) is a widely deployed J2EE application server. TeS implements the J2EE standard by 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 ("SQL for messaging") for publish/subscribe and message queuing products.

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.

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 HP 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 and simplicity. 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 scaleable 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 the HP Total-e-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 HP Total-e-Server Integration Mean?

The term “HP Total-e-Server Integration” means to use Smartsockets JMS as the JMS provider for HP Total-e-Server. Specifically, this involves loading a Smartsockets-JMS-linked EJB into HP Total-e-Server that can communicate with other J2EE applications via JMS while maintaining tightly coupled communications with other J2EE components.

Using SmartSockets for JMS as the JMS provider for TeS

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

First, you need to download and install all the necessary software. HP's Total-e-Server 7.3 (TeS) requires at least JDK 1.2.2. You can download the latest JDK from <http://java.sun.com>. It is recommended that you install JDK before proceeding with the TeS 7.3 installation.

In order to download and install the developer edition and documentation of TeS 7.3, please go to <http://www.bluestone.com> and click on “downloads”. This tutorial assumes that TeS has been installed in the default location, `c:\tes`.

In order to download and install Talarian Workbench for JMS 1.1, please go to <http://www.talarian.com> and select Download, Workbench for JMS 1.1 – for Windows NT (download the self extracting.exe file, and install in `C:\Program Files\Talarian\`). Shortly after downloading Smartsockets JMS, a Talarian representative will contact you and provide you with temporary licenses for both Smartsockets 6.0 and SmartMQ 2.1, both of which are required to run Smartsockets JMS.

You will probably want to download the TeS documentation using the following URL:

```
http://developer.bluestone.com/scripts/SaISAPI.dll/Gallery.class/control.jsp?action=TeS_73_docs_download
```

The documentation can be viewed by loading “`blsw_documentation.htm`” in the “docs” directory.

TeS comes with an MDB example that is suitable for our purposes. This example is located in:

```
C:\TeS\Samples\contrib\mdb\stock_example\source\com\bluestone\ejbtest
```

You need to open up a command prompt window in order to compile and run the MDB. This command prompt window will need to have the environment of both TeS 7.3 and SmartSockets for JMS. There are at least two ways to do this: start with a TeS environment and set the SS for JMS environment, or start with an SS for JMS environment and set the TeS 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 TeS environment settings, it is convenient to use `common.bat`, located in `C:\TeS\bin`. However, beware that `common.bat` destructively sets the `CLASSPATH`, overwriting the previous value. You can modify your `common.bat` to non-destructively set the `CLASSPATH`, by changing the first set statement from

```
set CLASSPATH=.
```

to the following:

```
set CLASSPATH=.;%CLASSPATH%
```

Alternatively, you can set the `SS` for `JMS` environment after the `TeS` environment has been set. You can type the following to set the `SS` for `JMS` environment manually or add it to `common.bat`:

```
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 the command prompt window, change to the following directory:

```
C:\TeS\Samples\contrib\mdb\stock_example\source\com\bluestone\ejbtest
```

Now compile the source by typing the following:

```
javac StockListenerMDB.java
```

Now you're ready to package the compiled `class` file into a `jar` file for deployment. You will need to use the `J2EE Developer Tool` that comes with `TeS`.

From the Start Menu, select `Programs > HPBluestone > Total-e-Server > Start J2EE Developer`. This opens the `J2EE Developer Tool`. From the main tool bar on the left, select `"J2EE Developer Options"`. This opens up the `J2EE Developer Options` dialog box. From the `Locate Deployment Descriptor DTD` drop-down list, select the item that corresponds to `EJB 2.0`, which contains the following text:

```
DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise  
JavaBeans 2.0//EN" "CLASSPATH:/com/bluestone/EJB/ejb-jar_2_0.dtd
```

Click `"OK"` to close the `J2EE Developer Options` dialog box.

Now from the `J2EE Developer's` main tool bar, select `"EJB Configurator"`. The `EJB Configurator Dialog` appears. Now click on `"Add bean"` from the `EJB Configurator's` toolbar. The `Add Beans` dialog box appears. Click the `"Browse"` button next to the `"Specify Bean Class"` field and browse to the location of the compiled `MDB`, which is:

```
C:\TeS\Samples\contrib\mdb\stock_example\source\com\bluestone\ejbtest
```

Now select `StockListenerMDB.class` and click `"OK"` to add the `MDB`.

From the navigation pane, expand the `MDB` folder and click `"StockListenerMDB"` to display all the deployment information about the `MDB`. Using the `Specify destination type` drop-down list, select `javax.jms.Topic`. Now save the packaged `MDB` by clicking on the `"Save"` button on the `EJB Configurator` toolbar. You should name the file `"stockMDB.jar"` and save it in the same location as the `class` file, which is:

```
C:\TeS\Samples\contrib\mdb\stock_example\source\com\bluestone\ejbtest
```

Now you will need to edit the `sajava.ini` to set up TeS to use a SSJMS as the JMS provider. The `sajava.ini` file is located in `C:\TeS\config`. Cut and paste the following entries into `sajava.ini`:

```
;
; Modified sajava.ini entries for using SmartSockets for JMS.
;
[ss_for_jms]

class=com.bluestone.ejb11.mdb.SaGenericPluggableJMS
pingClass=none
jmsProviderUrl=file://localhost/C:/PROGRA~1/Talarian/jms11/bindings
jmsProviderContext=com.sun.jndi.fscontext.RefFSContextFactory
providerTcfLookupName=TopicConnectionFactory
providerQcfLookupName=QueueConnectionFactory
user=
password=

[mdb]

fastRetryAttempts=5
fastRetryDelay=60
maxRetryAttempts=-1
longRetryDelay=300
pingEnabled=false
pingDelay=600

[StockListenerMDB]

destination=quotesZrh
pluggableJMS=ss_for_jms

[resources]

objectFactories=com.smartsockets.jms.TopicConnectionFactoryFactory,com.smartsockets.jm
s.TopicFactory,com.smartsockets.jms.QueueConnectionFactoryFactory,com.smartsockets.jms
.QueueFactory

[jms/tcf]

type=javax.jms.TopicConnectionFactory
pluggableJMS=ss_for_jms

[quotesZrh]

type=javax.jms.Topic
pluggableJMS=ss_for_jms
destination=quotesZrh
```

Now you need to compile the client publisher. Instead of using `StockQuoteProducer.java` from the `C:\TeS\samples` directory, use the `StockQuoteProducer.java` listed in Appendix B of this white paper. Copy and past the code into a text file and save it as `StockQuoteProducer.java` or modify the one that comes with TeS to match it. Now go back to your command prompt window that has both TeS and SS for JMS environments set up. Change directories to the location of the modified `StockQuoteProducer.java` and compile the client application by typing:

```
javac StockQuoteProducer.java
```

In order to run this application successfully, you need 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.

For this example you will also need another middleware component called RTrms. RTrms is necessary for durable subscriptions and persistent messages. To start RTrms, from the Start Menu, select Programs ? SmartSockets for JMS ? SmartSockets ? RTrms to start the RTrms.

A Smartsockets JMS administration tool 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 JMS Admin tool, from the Start Menu, select Programs -> Smartsockets JMS -> JMS -> SSJMS Admin. Using the Admin tool, you will need to create a TopicConnectionFactory called "TopicConnectionFactory" and a Topic named "quotesZrh" with the SmartSockets subject being something like "/quotesZrh".

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/b
indings
```

TeS 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.

Now open up a second command prompt with both SSJMS and TeS environments set. You can do this by opening up an SSJMS command prompt and setting the TeS environment using `common.bat`, which is located in `C:\TeS\bin`.

With the environment set, you are ready to start TeS. Start it by typing the following:

```
java SaApi.SaAppServController -config C:\TeS\config\sajava.ini
```

You should see something similar to the following:

```
Starting UBS SaApi.SaUbs on port -1

Attempting connection to <tcp:_node:5101> RTserver.
Connected to <tcp:_node:5101> RTserver.
```

Depending on how TeS is configured, you may also see the following:

```
[2001/11/07 14:17:04][main      ][E]: Failed to open socket to SAM on :19999
[2001/11/07 14:17:04][main      ][E]: Okay to proceed if not using SAM management.
```

If you get this error, you can ignore it because SAM management is not necessary for this example.

Go back to the command prompt window that you used to compile the publisher code. Start the publisher by typing:

```
java StockQuoteProducer
```

You should see something similar to the following:

```
Attempting connection to <tcp:_node:5101> RTserver.  
Connected to <tcp:_node:5101> RTserver.  
Attempting connection to <tcp:_node:5101> RTserver.  
Connected to <tcp:_node:5101> RTserver.  
Published quote# 1  
Published quote# 2  
Published quote# 3  
Published quote# 4  
Published quote# 5  
Published quote# 6
```

In the command window that TeS is running in, you should see something like the following:

```
[2001/11/07 14:27:38][Thread-10 ][I]: StockListenerMDB: setMessageDrivenContext  
called.  
[2001/11/07 14:27:38][Thread-10 ][I]: StockListenerMDB: ejbCreate called. [2001/11/07  
14:27:38][Thread-10 ][I]: Got a stock quote: Quote# 1 ACME 118  
[2001/11/07 14:27:39][Thread-10 ][I]: Got a stock quote: Quote# 2 ACME 118  
[2001/11/07 14:27:40][Thread-10 ][I]: Got a stock quote: Quote# 3 ACME 118  
[2001/11/07 14:27:41][Thread-10 ][I]: Got a stock quote: Quote# 4 ACME 118  
[2001/11/07 14:27:42][Thread-10 ][I]: Got a stock quote: Quote# 5 ACME 118  
[2001/11/07 14:27:43][Thread-10 ][I]: Got a stock quote: Quote# 6 ACME 118
```

You may get the following error message if you haven't set up a user name and password in the sajava.ini file:

```
[2001/11/07 14:27:38][Thread-10 ][E]:com.bluestone.security.AuthenticationFailedException: invalid username/password  
[2001/11/07 14:27:38][Thread-10 ][E]: at com.bluestone.ejb11.SaMessageEjbBean.  
setupAppSecurity(SaMessageEjbBean.java:287)  
[2001/11/07 14:27:38][Thread-10 ][E]: at com.bluestone.ejb11.SaMessageEjbBean.  
setSecurity(SaMessageEjbBean.java:155)  
[2001/11/07 14:27:38][Thread-10 ][E]: at com.bluestone.ejb11.SaMessageEjbBean.  
onBeforeMessage(SaMessageEjbBean.java:105)  
[2001/11/07 14:27:38][Thread-10 ][E]: at com.bluestone.ejb11.SaMessageEjbBean.  
onMessage(SaMessageEjbBean.java:71)  
[2001/11/07 14:27:38][Thread-10 ][E]: at com.smartsockets.jms.TopicSubscriberI  
mpl.deliverMsg(TopicSubscriberImpl.java:179)  
[2001/11/07 14:27:38][Thread-10 ][E]: at com.smartsockets.jms.TopicSessionImpl  
.run(TopicSessionImpl.java:312)  
[2001/11/07 14:27:38][Thread-10 ][E]: at java.lang.Thread.run(Unknown Source)
```

Regardless, the MDB should still be receiving messages.

At this point you have integrated SS For JMS with HP Total-e-Server server.

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 `BluestoneIntegrationExamples.zip` distribution. This directory contains modified versions of `StockQuoteProducer.java` and `StockListenerMDB.java`. You should rename the old versions of these files and copy these new versions to their respective locations. `StockListenerMDB.java` now has code for sending JMS messages inside the message listener callback. `StockQuoteProducer.java` has been modified to wait for a reply message from the MDB.

In order to run this example, you will need to open up the SSJMS Admin tool and create a new Topic called "replyTopic" and set the SmartSockets subject to something like `/replytopic`. This provides a back channel for the MDB to publish reply messages to.

You can compile and run the example just as was done for the non-sending MDB example.

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

```
Attempting connection to <tcp:_node:5101> RTserver.  
Connected to <tcp:_node:5101> RTserver.  
Attempting connection to <tcp:_node:5101> RTserver.  
Connected to <tcp:_node:5101> RTserver.  
Published quote# 1  
Receiving the message on topic replytopic  
Message: reply  
Published quote# 2  
Receiving the message on topic replytopic  
Message: reply
```

```
Published quote# 3
Receiving the message on topic replytopic
Message: reply
...
```

Looking in your TeS console window, you should see:

```
Got a stock quote: Quote# 1 ACME 118
Sending a reply message
Got a stock quote: Quote# 2 ACME 118
Sending a reply message
Got a stock quote: Quote# 3 ACME 118
Sending a reply message
...
```

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 "TopicConnectionFactory", 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 `TopicConnectionFactory`. You shouldn't get too comfortable with doing this as it slows the performance.

Appendix B: Code Samples

Stock Quote Producer – StockQuoteProducer.java

```
import javax.jms.*;
import javax.naming.NamingException;
import javax.naming.InitialContext;
import javax.naming.Context;

public class StockQuoteProducer {

    public static void main(String [] args) throws Exception
    {
        javax.naming.InitialContext context = null;
        java.util.Hashtable env = null;
        TopicConnectionFactory tcf = null;

        if (args.length != 0)
        {
            System.out.println("Usage: java StockQuoteProducer\n");
            return;
        }

        // If using JNDI for lookups, setup the environment,
        // pass it to the InitialContextFactory and
        // lookup the TopicConnectionFactory.
        context = new InitialContext();
        tcf = (TopicConnectionFactory)context.lookup("TopicConnectionFactory");

        // First, create the TopicSession, which is used to create
        // both publishers and their message objects.
        // We choose a non-transacted session with automatic
        // message acknowledgment for simplicity. See the programmer's
        // manual for more details on these parameters.

        TopicConnection connection = tcf.createTopicConnection();
        TopicSession session = connection.createTopicSession(false,
            TopicSession.AUTO_ACKNOWLEDGE);

        // Next convert the topic name into the corresponding
        // object, encapsulating the "radio station" that the
        // stock quote consumer can "tune in" to.
        Topic blueChipsZRH = (Topic) context.lookup("quotesZrh");

        // Next, create the TopicPublisher. This is the class that
        // assists with publishing events.
        TopicPublisher p = session.createPublisher(blueChipsZRH);

        // There is a local counter to tag the postings
        int pushCounter = 0;

        // That's it, the TopicPublisher is now ready for use
        // Publish stock quotes now
        while(true)
        {
            // Pack a string into a message. JMS insists on
            // having a StringBuffer, so do the conversion.
            TextMessage m=session.createTextMessage();
```

```

        m.setText("Quote# " + ++pushCounter + " ACME 118");
        try
        {
            // This simple call handles the rest.
            p.publish(m);
            System.out.println("Published quote# " + pushCounter);
        } catch(JMSEException e) {
            System.out.println("Error publishing quote# " + pushCounter);
            e.printStackTrace ();
        }

        Thread.currentThread().sleep(1000);
    }
}
}
}

```

Stock Listener MDB – StockListenerMDB.java

```

/**
 * Title:      StockListenerMDB
 * Description: This is the listener defined for Softwired's iBus example,
 *             converted to a MessageDrivenBean. In the Bluestone MDB example,
 *             any JMS implementation can be used.
 *
 *             The producer retains use of iBus specific classes, though this
 *             can be normalized to work with any JMS implementation by replacing
 *             the convenience classes with standard JNDI lookups. The example is
 *             unchanged to illustrate the ease of moving from a standard JMS
 *             listener to a Message Driven Bean.
 *
 * Copyright:  Copyright (c) 2000
 * Company:    Bluestone
 * @author     Greg
 * @version    1.0
 */
package com.bluestone.ejbtest;

import javax.ejb.MessageDrivenBean;
import javax.ejb.MessageDrivenContext;
import javax.jms.Message;
import javax.jms.TextMessage;
import javax.jms.JMSEException;

public class StockListenerMDB implements MessageDrivenBean
{

    public StockListenerMDB()
    {
    }

    public void onMessage(Message message) {
        // Pull out the quote from the message object.
        // We know that StockQuoteProducer sends TextMessages and
        // cast the object accordingly.
        try {
            String quote = ((TextMessage)message).getText();
            System.out.println("\nGot a stock quote: " + quote);
        } catch(JMSEException e) {
            System.out.println("Error processing message: "+message);
        }
    }

    public void ejbRemove() throws javax.ejb.EJBException
    {
        System.out.println("\nStockListenerMDB: ejbRemove called.");
    }

    public void setMessageDrivenContext(MessageDrivenContext parml) throws
    javax.ejb.EJBException

```

```
{
    System.out.println("\nStockListenerMDB: setMessageDrivenContext called.");
}

/** ejbCreate with no args required by spec, though not enforced by interface */
public void ejbCreate()
{
    System.out.println("\nStockListenerMDB: ejbCreate called.");
}
}
```

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
www.talarian.com

