Introduction to WSO2 Message Broker

Introduction

WSO2 Message Broker (MB) is a new Open Source project and product from WSO2 that provides messaging functionality within the WSO2 Carbon platform and to other clients in various languages. It works either standalone or in conjunction with products and components such as the WSO2 ESB and WSO2 Complex Event Processing Server.

MB is based on the Apache Qpid/Java project (http://qpid.apache.org). From Apache Qpid, MB gets core support for the AMQP protocol and JMS API. On top of that WSO2 has added support for Amazon SQS APIs and WS-Eventing support.

Understanding how the MB broker fits into Enterprise Architecture

The Message Broker provides three main capabilities into an overall Enterprise Architecture:

- A queueing/persistent message facility
- An event distribution (pub/sub) model
- An intermediary where multiple systems can connect irrespective of the direction of messages.

To give some concrete examples of these benefits, here are some scenarios:

1) In the WSO2 ESB, a common pattern is to persist the message from an incoming HTTP request into a persistent message queue, and then process onbound from there. MB can provide the persistent queue.
2) The WSO2 ESB already has an event distribution model and eventing support, but the QPid-based broker provides higher performance as well as supporting the JMS API.
3) For example, you may wish to send messages from outside a firewall to a server inside. You could connect an ESB or Service Host within the firewall to a Message Broker running outside the firewall (for example on Amazon EC2). This model is used by the WSO2 Cloud Services Gateway.

Where does AMQP fit?

AMQP (www.amqp.org) is an open protocol for messaging. Whilst the AMQP protocol is still under development, it has released three stable releases (0-8, 0-9-1, and 0-10), with a 1.0 due during 2011. There are a number of implementations of the AMQP standard in production, including Apache Qpid (both Java and C++ versions), RabbitMQ, OpenAMQ and others.

WSO2 has been a member of the AMQP working group for several years, and we strongly support AMQP as the way to introduce interoperability and greater openness into the messaging space.

The Qpid broker supports a variety of clients on top of the AMQP protocol. The most useful of these for Carbon is the Java JMS 1.1 API, which provides a portable API as well as the main interface with the WSO2 ESB. In addition there are C#

and other APIs. WSO2 MB also extends these with WS-Eventing and Amazon SQS APIs for interoperability using HTTP, REST and SOAP.

Installing the WSO2 MB

You can download the WSO2 MB Beta from:
http://people.wso2.com/~manjular/wso2mb-1.0.0-beta.zip

Once you have downloaded and unzipped, simply switch to the install directory

```
cd wso2mb-1.0.0-SNAPSHOT
bin\wso2server.bat  [ON WINDOWS]
bin/wso2server.sh [ON LINUX/MACOSX]
```

Let's refer to the install directory as <WSO2MB> from now on.

You should see the server startup:

```
[2011-03-16 14:00:12,471]  INFO {org.wso2.carbon.server.Main} -  Initializing
system...
[2011-03-16 14:00:12,840]  INFO {org.wso2.carbon.server.TomcatCarbonWebappDeployer} -
Deployed Carbon webapp:
StandardEngine[Tomcat].StandardHost[defaulthost].StandardContext[/]
[2011-03-16 14:00:14,147]  INFO {org.wso2.carbon.atomikos.TransactionFactory} -
Starting Atomikos Transaction Manager 3.7.0
[2011-03-16 14:00:19,952]  INFO {org.wso2.carbon.core.internal.CarbonCoreActivator} -
Starting WSO2 Carbon...
[2011-03-16 14:00:19,983]  INFO {org.wso2.carbon.core.internal.CarbonCoreActivator} -
Operating System : Mac OS X 10.6.6, x86_64
[2011-03-16 14:00:19,984]  INFO {org.wso2.carbon.core.internal.CarbonCoreActivator} -
Java Home        : /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
[2011-03-16 14:00:19,984]  INFO {org.wso2.carbon.core.internal.CarbonCoreActivator} -
Java Version     : 1.6.0_24
[2011-03-16 14:00:19,985]  INFO {org.wso2.carbon.core.internal.CarbonCoreActivator} -
Java VM          : Java HotSpot(TM) 64-Bit Server VM 19.1-b02-334,Apple Inc.
[2011-03-16 14:00:19,985]  INFO {org.wso2.carbon.core.internal.CarbonCoreActivator} -
Carbon Home      : /Users/paul/wso2/wso2mb-1.0.0-SNAPSHOT
[2011-03-16 14:00:19,985]  INFO {org.wso2.carbon.core.internal.CarbonCoreActivator} -
Java Temp Dir    : /Users/paul/wso2/wso2mb-1.0.0-SNAPSHOT/tmp
[2011-03-16 14:00:19,986]  INFO {org.wso2.carbon.core.internal.CarbonCoreActivator} -
User             : paul, en-US, Europe/London
2011-03-16 14:00:12,471]  INFO {org.wso2.carbon.server.Main} -  Initializing system...
```

some logs deleted

```
[2011-03-16 14:00:41,691]  INFO
{org.wso2.carbon.core.transports.http.HttpsTransportListener} -  HTTPS port       :
9443
[2011-03-16 14:00:41,691]  INFO
{org.wso2.carbon.core.transports.http.HttpTransportListener} -  HTTP port        :
9763
[2011-03-16 14:00:42,422]  INFO {org.wso2.carbon.ui.internal.CarbonUIServiceComponent}
-  Mgt Console URL  : https://192.168.1.100:9443/carbon/
[2011-03-16 14:00:42,499]  INFO
{org.wso2.carbon.core.internal.StartupFinalizerServiceComponent} -  Started Transport
Listener Manager
[2011-03-16 14:00:42,500]  INFO
{org.wso2.carbon.core.internal.StartupFinalizerServiceComponent} -  Server           :
WSO2 MB -1.0.0-SNAPSHOT
[2011-03-16 14:00:42,506]  INFO
{org.wso2.carbon.core.internal.StartupFinalizerServiceComponent} -  WSO2 Carbon
started in 27 sec
2011-03-16 14:00:12,471]  INFO {org.wso2.carbon.server.Main} -  Initializing system...
```

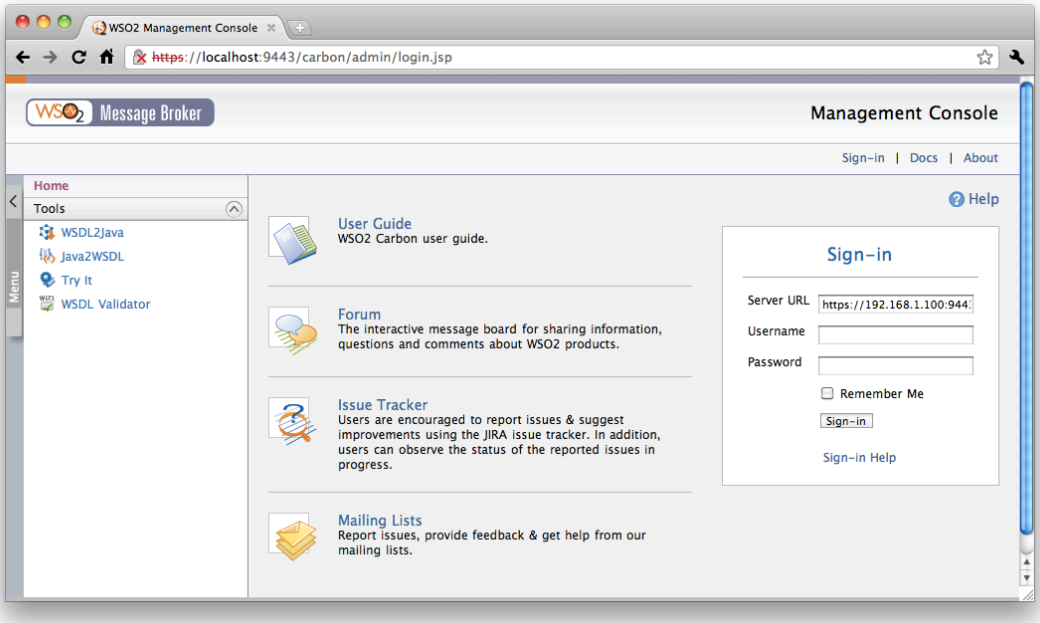
WSO2 Message Broker is installable in more ways for production systems. Typically it is either registered as a Linux Daemon or as a Windows Service – but for now we will stick with the command-line version for simplicity.

Once the server is running you can access the management console. Point your browser at:

https://localhost:9443

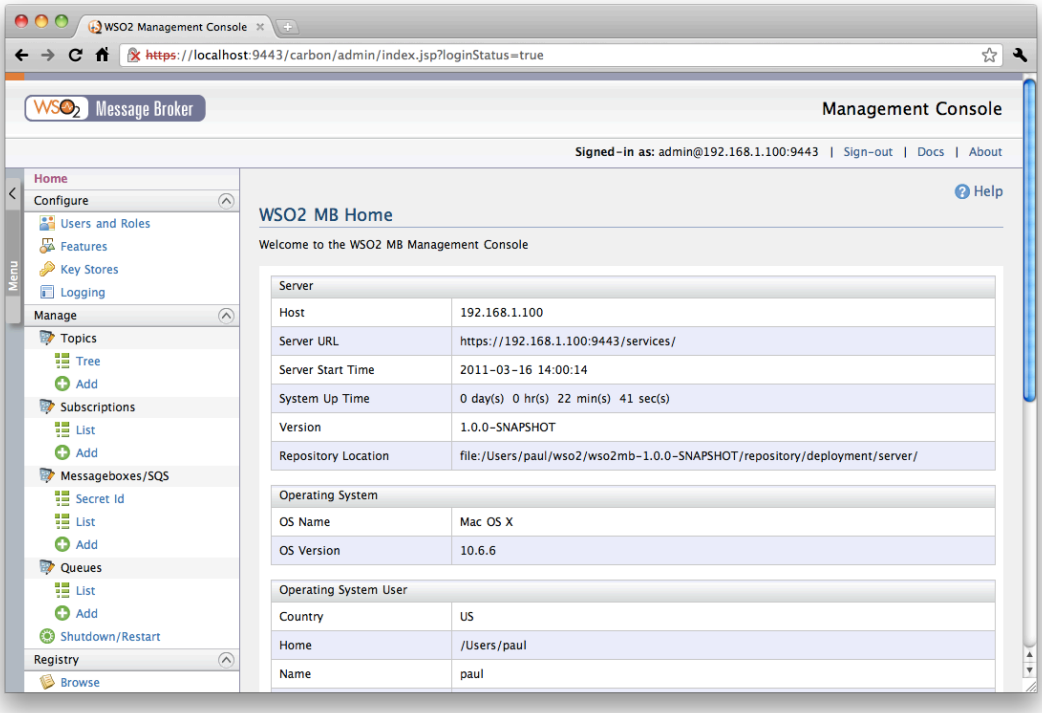
Initially you will see a browser screen warning you about the certificates. Please tell your browser to continue (For a production server you would normally install a proper SSL/TLS certificate, but for initial install we generate a self-signed certificate that you need to agree to use).

Once you have accepted the certificate, you should see a screen like:



You can login using the default user/password which is *admin/admin.*

Once you login you should see the following screen:



Before we examine the admin console, lets first create a simple JMS client that will communicate with the server via AMQP on TCP/IP.

Getting Started with JMS

The Java Message Service (JMS) specification - http://www.oracle.com/technetwork/java/index-jsp-142945.html - is a specification for talking to message brokers. It is unfortunately poorly named: the word "service" implies this is an implementation, but JMS does not define an actual messaging service, instead just the API which is used to access JMS providers. "Java Messaging API" would more accurately express what JMS is. The result is that there are a variety of JMS providers, and they often have quite different approaches to their core model.

The WSO2 Message Broker is based on the Apache Qpid project (http://qpid.apache.org) and is a compliant implementation of the JMS specification, as well as various levels of the AMQP specification (0-8, 0-9-1, 0-10).

To write completely standard portable JMS code, you need to use a JNDI provider to gain access to the JMS connection, queues, etc. In this example we will use a Qpid JNDI provider backed by a simple set of properties. This makes the overall system simple and highly portable.

Here is a sample JMS application that can be used to test access to the Message Broker. You can find this code here:
http://people.apache.org/~pzf/MB/JMSExample.java

First are some required imports.

```
import javax.jms.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Properties;
```

Next is a simple “main” class definition:

```
public class JMSExample {

public static void main(String[] args) {
       JMSExample producer = new JMSExample();
       producer.runTest();
}


private void runTest() {
```

Since this is just an example, we will place the complete logic in a try/catch block.

```
try {
```

Normally the JNDI is configured by a properties file, but you can also do it from an in-memory set of properties. To see a similar setup with a properties file, take a look at the ESB example below. Here is a properties object to store the properties:

```
Properties properties = new Properties();
```

In order to bootstrap the JNDI entries for the connection factory and queue, we set name/value pairs into the simple properties object:

```
properties.put("connectionfactory.cf",
       "amqp://admin:admin@carbon/carbon?brokerlist='tcp://localhost:5672'")
;
```

The property name “connectionfactory.cf” denotes that we are creating an object of type ConnectionFactory with name “cf”. The value is a URL that is used to bootstrap the ConnectionFactory: this URL points to the AMQP broker. The syntax is broken up as follows:

```
       amqp://              Indicates this is an AMQP URL
       admin:admin@   This is the username/password
       carbon/carbon  The client ID and virtual host
       ?                    separator for options
       brokerlist='tcp://localhost:5672'           A list of broker URLs to use
```

For more information on this URL syntax please see:
https://cwiki.apache.org/qpid/connection-url-format.html

The virtual host name is part of the definition in:
<WSO2MB>/repository/conf/qpid/etc/virtualhosts.xml

This file also defines aspects such as the maximum number of messages in a queue and the queue depth (maximum size in bytes of the queue).

Now we need to create a JNDI entry for the queue we are going to talk to:

```
properties.put("destination.samplequeue", "samplequeue; {create:always}");
```

The property name “destination.samplequeue” indicates creating a destination with a JNDI name of “samplequeue”. The property value “samplequeue; {create:always}” indicates a queue named “samplequeue” with an attribute which tells the broker to create the queue if it doesn’t exist.

These properties are specific to the particular JNDI implementation we are using, which is the Qpid "PropertiesFileInitialContextFactory". So now we need to configure JNDI to use this implementation:

```
properties.put("java.naming.factory.initial",
"org.apache.qpid.jndi.PropertiesFileInitialContextFactory");
```

Now we can do our JNDI lookups:

```
Context context = new InitialContext(properties);

ConnectionFactory connectionFactory =
      (ConnectionFactory) context.lookup("cf");
```

Having "found" a JMS Connection Factory in the JNDI, we can now create a connection to the broker:

```
Connection connection = connectionFactory.createConnection();
connection.start();
```

And now we can create a JMS Session:

```
Session session = connection.createSession(false,
                                  Session.AUTO_ACKNOWLEDGE);
```

One more lookup from JNDI will lookup our queue:

```
Destination destination = (Destination) context
                                  .lookup("samplequeue");
```

Now we can create a Producer, and send a message:

```
MessageProducer producer = session.createProducer(destination);
TextMessage outMessage = session.createTextMessage();
outMessage.setText("Hello World!");
producer.send(outMessage);
```

Of course, in real life you would most likely NOT now retrieve that same message from the same application, but for this example we will now retrieve the message:

```
MessageConsumer consumer = session.createConsumer(destination);
Message inMessage = consumer.receive();
System.out.println(((TextMessage)inMessage).getText());
```

And close up the connection and the initial context:

```
connection.close();
context.close();
} catch (Exception exp) {
      exp.printStackTrace();
}
```

To try out this client you need the correct client JARs.

In the beta release you will find:

```
      <WSO2MB>/jms-client-lib/geronimo-jms_1.1_spec-1.1.0.wso2v1.jar
      <WSO2MB>/jms-client-lib/qpid-client-0.9.wso2v2.jar
```

You also need to reference

```
      <WSO2MB>/lib/log4j-1.2.13.jar
```

Once you have those in your classpath you can run the program. You should see some simple output:

```
log4j:WARN No appenders could be found for logger
(org.apache.qpid.jndi.PropertiesFileInitialContextFactory).
log4j:WARN Please initialize the log4j system properly.
Hello World!
```

If you got that far, congratulations!

In the next section we are going to look at using the ESB with the Message Broker.

There are two approaches for this:
1) If you are using the existing WSO2 ESB 3.0.1 or similar, you can deploy the MB client libraries and communicate using the network.

2) As of the next WSO2 ESB release (3.1.0) it will include the Qpid/MB features as part of the release and you can utilize the Message Broker/JMS runtime locally in the same JVM.

WSO2 MB and WSO2 ESB together

In this first instance we are going to get the WSO2 ESB and MB to work together. Assuming that you already have the MB installed and running, you will first need to install the ESB and change the ports of the admin console so that they don't clash. You can download WSO2 ESB 3.0.1 from:

http://wso2.org/downloads/esb

The install procedure is similar: unzip the ESB, but don't start it up yet. Let's name (for this guide) the directory where you installed the ESB as <WSO2ESB>.

First let's edit the ports on which the ESB listens. (Alternatively you could do the same to the MB instead).

Edit the <WSO2ESB>\repository\conf\mgt-transports.xml

This file defines which ports the management console runs (HTTP and HTTPS).

Please change:

```
<transport name="http"
class="org.wso2.carbon.server.transports.http.HttpTransport">
    <parameter name="port">9763</parameter>
```

to read:

```
<transport name="http"
class="org.wso2.carbon.server.transports.http.HttpTransport">
    <parameter name="port">9764</parameter>
```

Similarly change the HTTPS port to be 9444:

```
  <transport name="https"
class="org.wso2.carbon.server.transports.http.HttpsTransport">
     <parameter name="port">9444</parameter>
```

Now the next step is to ensure that the ESB has the right drivers to talk to the MB. Copy the following JARs into the <WSO2ESB>\repository\components\lib directory:

```
      <WSO2MB>/jms-client-lib/geronimo-jms_1.1_spec-1.1.0.wso2v1.jar
      <WSO2MB>/jms-client-lib/qpid-client-0.9.wso2v2.jar
```

We also need to configure the JMS transport correctly. To do this we edit the axis2.xml file:

<WSO2ESB>\repository\conf\axis2.xml

This file has the JMS transport commented out. It also needs the settings updated to use the Qpid libraries. Change the file so that the JMS receiver and sender sections look like this:

```
  <transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">
        <parameter name="myTopicConnectionFactory" locked="false">
                <parameter name="java.naming.factory.initial"
locked="false">org.apache.qpid.jndi.PropertiesFileInitialContextFactory</parameter>
                <parameter name="java.naming.provider.url"
locked="false">resources/jndi.properties</parameter>
                <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">TopicConnectionFactory</parameter>
                <parameter name="transport.jms.ConnectionFactoryType"
locked="false">topic</parameter>
        </parameter>

        <parameter name="myQueueConnectionFactory" locked="false">
                <parameter name="java.naming.factory.initial"
locked="false">org.apache.qpid.jndi.PropertiesFileInitialContextFactory</parameter>
                <parameter name="java.naming.provider.url"
locked="false">resources/jndi.properties</parameter>
                <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
                <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
        </parameter>

        <parameter name="default" locked="false">
                <parameter name="java.naming.factory.initial"
locked="false">org.apache.qpid.jndi.PropertiesFileInitialContextFactory</parameter>
                <parameter name="java.naming.provider.url"
locked="false">resources/jndi.properties</parameter>
                <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
                <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
        </parameter>
    </transportReceiver>
```

You can find my copy of the edited axis2.xml here:
http://people.wso2.com/~paul/mb-guide-1.0/

If you have looked through the JMS config you will notice it references a JNDI resource: resources/jndi.properties.

This is used to do the same thing the hard-coded properties we used above do – configure the local JNDI that the JMS client inside the ESB will use. In a future release of the ESB we expect to automatically configure this JNDI, but in the meantime, we can simply create a file in the <WSO2ESB>/resources directory.

Please create <WSO2ESB>/resources/jndi.properties to look like this:

```
connectionfactory.TopicConnectionFactory = \
amqp://admin:admin@carbon/carbon?brokerlist='tcp://localhost:5672'
connectionfactory.QueueConnectionFactory = \
amqp://admin:admin@carbon/carbon?brokerlist='tcp://localhost:5672'
destination.dynamicQueues/myqueue=jmsdestinationqueue; {create:always}
destination.myqueue=jmsdestinationqueue; {create:always}
```

Please note that the lines ending \ are actually split for formatting and should be one continuous line.

Check if you need the last two lines?

You can find this file here:
http://people.apache.org/~pzf/MB/

Now we should be able to start the ESB. Of course it won't actually do anything yet.

Just for interest, you can try starting the WSO2 ESB with the MB **stopped.** Now that the JMS transport is enabled, you should see connection errors:

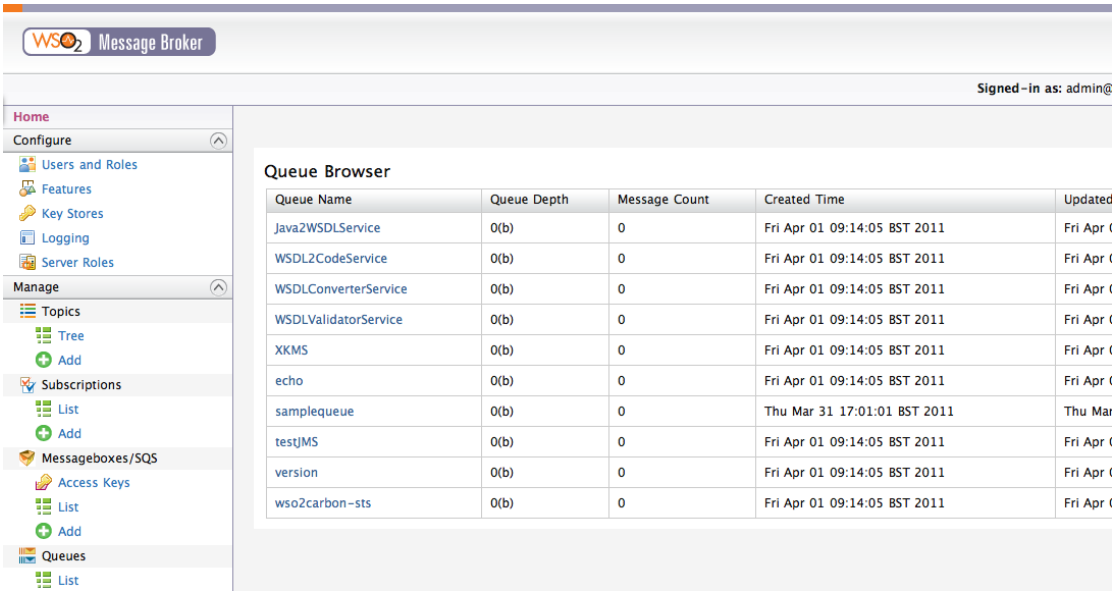
```
javax.jms.JMSException: Error creating connection: Connection refused
        at
org.apache.qpid.client.AMQConnectionFactory.createConnection(AMQConnectionF
actory.java:286)
        at
org.apache.axis2.transport.jms.JMSUtils.createConnection(JMSUtils.java:579)
        at
org.apache.axis2.transport.jms.ServiceTaskManager$MessageListenerTask.creat
eConnection(ServiceTaskManager.java:803)
        at
org.apache.axis2.transport.jms.ServiceTaskManager$MessageListenerTask.getCo
nnection(ServiceTaskManager.java:688)
        at
org.apache.axis2.transport.jms.ServiceTaskManager$MessageListenerTask.recei
veMessage(ServiceTaskManager.java:487)
        at
org.apache.axis2.transport.jms.ServiceTaskManager$MessageListenerTask.run(S
erviceTaskManager.java:412)
        at
org.apache.axis2.transport.base.threads.NativeWorkerPool$1.run(NativeWorker
Pool.java:58)
```

The ESB will still start but the JMS transport will be disabled.

If you start the MB, then the ESB should start fine. You will however see some warning lines:
[2011-04-01 09:14:05,320] WARN - JMSUtils Cannot locate destination : WSDLValidatorService

This is because the ESB is binding internal services to the JMS transport. In the most recent builds of the ESB this has been changed so that the ESB only binds internal services to HTTP/S transports to avoid this.

If you go to the Message Broker web console, you can now see the queues that have been created to support the ESB. Simply click on the left-hand menu item **Queues→List**.



Now we can create a simple proxy service that will test the JMS connectivity. This is a slight variation on one of the standard ESB Samples.

This proxy service expects a SOAP or XML message via HTTP POST from a client and simply puts the body of this message into a JMS queue. The server then responds with an **HTTP 202 Accepted** to the client. This is a great test, because we can use something as simple as **curl** to post messages into the ESB.

Here is the proxy definition for the ESB:

```
<?xml version="1.0" encoding="UTF-8"?>
<proxy xmlns="http://ws.apache.org/ns/synapse" name="testJMS"
transports="https jms http" startOnLoad="true" trace="disable">
    <target>
        <endpoint name="jmsqueue">
            <address
uri="jms:/myqueue?transport.jms.ConnectionFactoryJNDIName=QueueConnectionFa
ctory&amp;java.naming.factory.initial=org.apache.qpid.jndi.PropertiesFileIn
itialContextFactory&amp;java.naming.provider.url=resources/jndi.properties"
/>
        </endpoint>
        <inSequence>
        <property action="set" name="OUT_ONLY" value="true"/>
        <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
        </inSequence>
        <outSequence>
            <send/>
        </outSequence>
    </target>
</proxy>
```

This file is available at http://people.apache.org/~pzf/MB/testJMS.xml

You need to place this file here:
`<WSO2ESB>/repository/conf/synapse-config/proxy-services/testJMS.xml`

This is a nice feature of the ESB. Effectively you can configure independent proxy services, each with their own config file or registry entry, and the ESB amalgamates them at runtime to create a single consistent ESB. This is great for doing incremental changes. You can even change this file at runtime and have the proxy hot-deployed.

The proxy service is really simple. Basically it just sets the destination to send the message on to the JMS queue, which is defined using a combination of JNDI and the JMS URL.

The JMS URL is made up of:

`jms:/myqueue`
Look for a JNDI entry "myqueue"
(see jndi properties above)

`?`
Separator indicating extra attributes

`transport.jms.ConnectionFactoryJNDIName=QueueConnectionFactory`
Look up ConnectionFactory in JNDI with name QueueConnectionFactory

`&`
Separator (this will convert to '&')

`java.naming.factory.initial=`
`org.apache.qpid.jndi.PropertiesFileInitialContextFactory`
Use the Qpid properties-based JNDI we saw earlier

`&`
Another separator

`java.naming.provider.url=resources/jndi.properties`
*Look in **resources/jndi.properties** for the JNDI properties file*

The next part of the proxy configuration simply tells the ESB this is a one-way flow and not to expect a response:

```
<property action="set" name="OUT_ONLY" value="true"/>
```

The next line ensures the ESB sends back an HTTP 202 Accepted response to the client:

```
<property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
```

All the rest of the config is completely default.

In order to try it out, there is a simple XML test file which you can send to the ESB using curl:

samplexml.xml

```
<test xmlns="http://fremantle.org">
   <sample>data</sample>
</test>
```

Once again you can find this file here:
http://people.apache.org/~pzf/MB/

For the next step, please ensure you have a copy of *curl* installed. If you are on Linux or Mac you will have it by default. On Windows you can find a free version on the web. Let's try the request against the ESB:

```
curl http://localhost:8280/services/testJMS/a -X POST -H
'Content-type: text/plain' --data @samplexml.xml
```

Run this a few times just for fun. You won't see much. If all is going well, you won't see any errors on the WSO2 ESB console either. If you add '–v' to the curl command line you will see a lot more information about the HTTP section of the flow and you should see a nice sign that things are going well:

```
      < HTTP/1.1 202 Accepted
```

Now go back to the MB console and look at the **Queue→List** page. You should now see some messages in the queue:

Queue Name	Queue Depth	Message Count	Created Time	Updated Time	Type
echo	0(b)	0	Fri Apr 01 09:14:05 BST 2011	Fri Apr 01 09:14:05 BST 2011	
jmsdestinationqueue	390(b)	6	Fri Apr 01 10:04:31 BST 2011	Fri Apr 01 10:04:31 BST 2011	

As you can see in my example I sent 6 messages.

As an exercise, why not try modifying the simple JMS code to pick up those messages from the JMS queue. If you get stuck there is a sample in the same place as the other code.

Conclusion

There is a lot more we can do with MB. In future articles I hope to cover using a C# client to interact, using the SQS support, and how the MB code can be embedded directly into the ESB to provide in-process queueing and eventing. In the meantime, I hope this has provided a simple introduction to get you started with WSO2 MB.