

1 Introduction to WSO2 Message Broker

3 Introduction

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

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

14 Understanding how the MB broker fits into Enterprise Architecture

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

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

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

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

34 Where does AMQP fit?

36 AMQP (www.amqp.org) is an open protocol for messaging. Whilst the AMQP
37 protocol is still under development, it has released three stable releases (0-8, 0-
38 9-1, and 0-10), with a 1.0 due during 2011. There are a number of
39 implementations of the AMQP standard in production, including Apache Qpid
40 (both Java and C++ versions), RabbitMQ, OpenAMQ and others.

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

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

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

51

52 **Installing the WSO2 MB**

53

54 You can download the WSO2 MB Beta from:

55 <http://people.wso2.com/~manjular/wso2mb-1.0.0-beta.zip>

56

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

58

59 `cd wso2mb-1.0.0-SNAPSHOT`

60 `bin\wso2server.bat` [ON WINDOWS]

61 `bin/wso2server.sh` [ON LINUX/MACOSX]

62

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

64

65 You should see the server startup:

```
66 [2011-03-16 14:00:12,471] INFO {org.wso2.carbon.server.Main} - Initializing
67 system...
68 [2011-03-16 14:00:12,840] INFO {org.wso2.carbon.server.TomcatCarbonWebappDeployer} -
69 Deployed Carbon webapp:
70 StandardEngine[Tomcat].StandardHost[defaultHost].StandardContext[/]
71 [2011-03-16 14:00:14,147] INFO {org.wso2.carbon.atomikos.TransactionFactory} -
72 Starting Atomikos Transaction Manager 3.7.0
73 [2011-03-16 14:00:19,952] INFO {org.wso2.carbon.core.internal.CarbonCoreActivator} -
74 Starting WSO2 Carbon...
75 [2011-03-16 14:00:19,983] INFO {org.wso2.carbon.core.internal.CarbonCoreActivator} -
76 Operating System : Mac OS X 10.6.6, x86_64
77 [2011-03-16 14:00:19,984] INFO {org.wso2.carbon.core.internal.CarbonCoreActivator} -
78 Java Home : /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
79 [2011-03-16 14:00:19,984] INFO {org.wso2.carbon.core.internal.CarbonCoreActivator} -
80 Java Version : 1.6.0_24
81 [2011-03-16 14:00:19,985] INFO {org.wso2.carbon.core.internal.CarbonCoreActivator} -
82 Java VM : Java HotSpot(TM) 64-Bit Server VM 19.1-b02-334, Apple Inc.
83 [2011-03-16 14:00:19,985] INFO {org.wso2.carbon.core.internal.CarbonCoreActivator} -
84 Carbon Home : /Users/paul/wso2/wso2mb-1.0.0-SNAPSHOT
85 [2011-03-16 14:00:19,985] INFO {org.wso2.carbon.core.internal.CarbonCoreActivator} -
86 Java Temp Dir : /Users/paul/wso2/wso2mb-1.0.0-SNAPSHOT/tmp
87 [2011-03-16 14:00:19,986] INFO {org.wso2.carbon.core.internal.CarbonCoreActivator} -
88 User : paul, en-US, Europe/London
89 2011-03-16 14:00:12,471] INFO {org.wso2.carbon.server.Main} - Initializing system...
90
```

91 *some logs deleted*

92

```
93 [2011-03-16 14:00:41,691] INFO
94 {org.wso2.carbon.core.transports.http.HttpsTransportListener} - HTTPS port :
95 9443
96 [2011-03-16 14:00:41,691] INFO
97 {org.wso2.carbon.core.transports.http.HttpTransportListener} - HTTP port :
98 9763
99 [2011-03-16 14:00:42,422] INFO {org.wso2.carbon.ui.internal.CarbonUIServiceComponent}
100 - Mgt Console URL : https://192.168.1.100:9443/carbon/
101 [2011-03-16 14:00:42,499] INFO
102 {org.wso2.carbon.core.internal.StartupFinalizerServiceComponent} - Started Transport
103 Listener Manager
104 [2011-03-16 14:00:42,500] INFO
105 {org.wso2.carbon.core.internal.StartupFinalizerServiceComponent} - Server :
106 WSO2 MB -1.0.0-SNAPSHOT
107 [2011-03-16 14:00:42,506] INFO
108 {org.wso2.carbon.core.internal.StartupFinalizerServiceComponent} - WSO2 Carbon
109 started in 27 sec
110 2011-03-16 14:00:12,471] INFO {org.wso2.carbon.server.Main} - Initializing system...
111
```

112 WSO2 Message Broker is installable in more ways for production systems.

113 Typically it is either registered as a Linux Daemon or as a Windows Service – but

114 for now we will stick with the command-line version for simplicity.

115

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

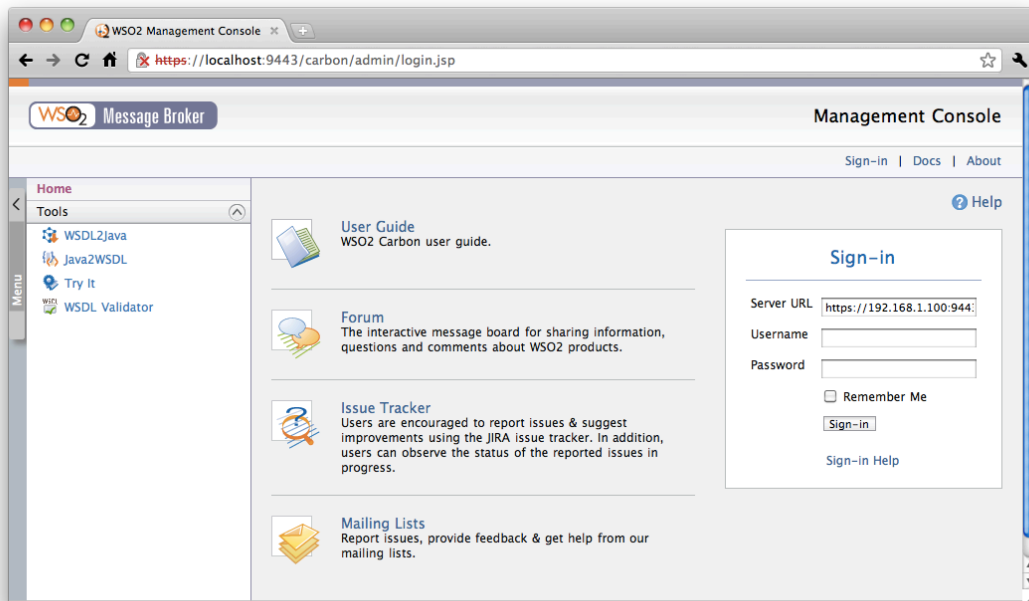
118 <https://localhost:9443>

119

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

124

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

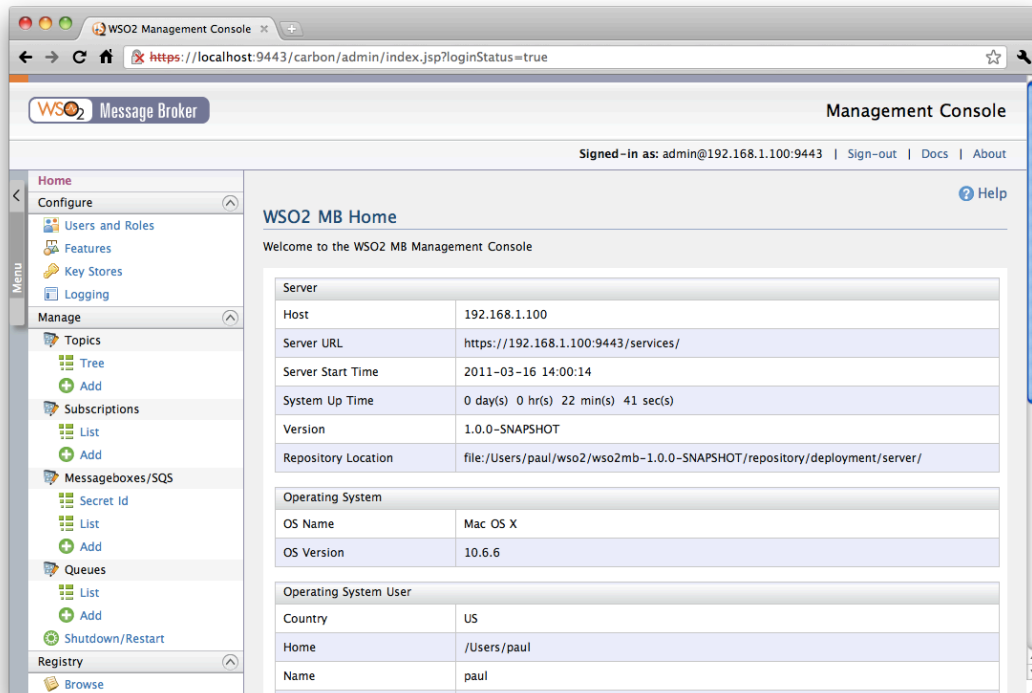


126

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

128

129 Once you login you should see the following screen:



130

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

133

134 **Getting Started with JMS**

135 The Java Message Service (JMS) specification -

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

143

144 The WSO2 Message Broker is based on the Apache Qpid project
145 (<http://qpid.apache.org>) and is a compliant implementation of the JMS
146 specification, as well as various levels of the AMQP specification (0-8, 0-9-1, 0-
147 10).

148

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

153

154 Here is a sample JMS application that can be used to test access to the Message
155 Broker. You can find this code here:

156 <http://people.apache.org/~pzf/MB/JMSEExample.java>

157

158

159

160 First are some required imports.

```
161
162 import javax.jms.*;
163 import javax.naming.Context;
164 import javax.naming.InitialContext;
165 import java.util.Properties;
166
```

167 Next is a simple “main” class definition:

```
168
169 public class JMSEExample {
170
171     public static void main(String[] args) {
172         JMSEExample producer = new JMSEExample();
173         producer.runTest();
174     }
175
176
177     private void runTest() {
178
```

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

```
180
181 try {
182
```

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

```
187
188 Properties properties = new Properties();
189
```

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

```
192
193 properties.put("connectionfactory.cf",
194              "amqp://admin:admin@carbon/carbon?brokerlist='tcp://localhost:5672'");
195 ;
196
```

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

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

206

207 For more information on this URL syntax please see:

208 <https://cwiki.apache.org/qpid/connection-url-format.html>

209

210 The virtual host name is part of the definition in:

```
211 <WSO2MB>/repository/conf/qpid/etc/virtualhosts.xml
212
```

212

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

215

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

```
217
218 properties.put("destination.samplequeue", "samplequeue; {create:always}");
219
```

220

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

224

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

228

229

230

231

232

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

233

Now we can do our JNDI lookups:

234

235

236

237

238

```
Context context = new InitialContext(properties);  
ConnectionFactory connectionFactory =  
    (ConnectionFactory) context.lookup("cf");
```

239

240

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

242

243

244

245

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

246

And now we can create a JMS Session:

247

248

249

250

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

251

One more lookup from JNDI will lookup our queue:

252

253

254

255

256

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

257

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

258

259

260

261

262

263

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

264

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

267

268

269

270

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

271

And close up the connection and the initial context:

272

273

274

275

276

277

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

278

279

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

280

281

In the beta release you will find:

282

283

284

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

285

You also need to reference

286

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

287

288

289

290

291

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

```
294  
295 log4j:WARN No appenders could be found for logger  
296 (org.apache.qpid.jndi.PropertiesFileInitialContextFactory).  
297 log4j:WARN Please initialize the log4j system properly.  
298 Hello World!
```

299
300 If you got that far, congratulations!

301
302 In the next section we are going to look at using the ESB with the Message
303 Broker.

304
305 There are two approaches for this:

306 1) If you are using the existing WSO2 ESB 3.0.1 or similar, you can deploy the MB
307 client libraries and communicate using the network.

308
309 2) As of the next WSO2 ESB release (3.1.0) it will include the Qpid/MB features
310 as part of the release and you can utilize the Message Broker/JMS runtime locally
311 in the same JVM.

312
313 **WSO2 MB and WSO2 ESB together**

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

318 <http://wso2.org/downloads/esb>

319
320 The install procedure is similar: unzip the ESB, but don't start it up yet. Let's
321 name (for this guide) the directory where you installed the ESB as <WSO2ESB>.

322
323 First let's edit the ports on which the ESB listens. (Alternatively you could do the
324 same to the MB instead).

325
326 Edit the <WSO2ESB>\repository\conf\mgt-transport.xml

327
328 This file defines which ports the management console runs (HTTP and HTTPS).

329
330 Please change:

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

334
335 to read:

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

339
340
341

342 Similarly change the HTTPS port to be 9444:
343 <transport name="https"
344 class="org.wso2.carbon.server.transports.http.HttpsTransport">
345 <parameter name="port">9444</parameter>

346
347 Now the next step is to ensure that the ESB has the right drivers to talk to the
348 MB. Copy the following JARs into the <WSO2ESB>\repository\components\lib
349 directory:
350 <WSO2MB>/jms-client-lib/geronimo-jms_1.1_spec-1.1.0.wso2v1.jar
351 <WSO2MB>/jms-client-lib/qpid-client-0.9.wso2v2.jar
352

353
354 We also need to configure the JMS transport correctly. To do this we edit the
355 axis2.xml file:
356 <WSO2ESB>\repository\conf\axis2.xml
357

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

```
361 <transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">  
362 <parameter name="myTopicConnectionFactory" locked="false">  
363 <parameter name="java.naming.factory.initial"  
364 locked="false">org.apache.qpid.jndi.PropertiesFileInitialContextFactory</parameter>  
365 <parameter name="java.naming.provider.url"  
366 locked="false">resources/jndi.properties</parameter>  
367 <parameter name="transport.jms.ConnectionFactoryJNDIName"  
368 locked="false">TopicConnectionFactory</parameter>  
369 <parameter name="transport.jms.ConnectionFactoryType"  
370 locked="false">topic</parameter>  
371 </parameter>  
372  
373 <parameter name="myQueueConnectionFactory" locked="false">  
374 <parameter name="java.naming.factory.initial"  
375 locked="false">org.apache.qpid.jndi.PropertiesFileInitialContextFactory</parameter>  
376 <parameter name="java.naming.provider.url"  
377 locked="false">resources/jndi.properties</parameter>  
378 <parameter name="transport.jms.ConnectionFactoryJNDIName"  
379 locked="false">QueueConnectionFactory</parameter>  
380 <parameter name="transport.jms.ConnectionFactoryType"  
381 locked="false">queue</parameter>  
382 </parameter>  
383  
384 <parameter name="default" locked="false">  
385 <parameter name="java.naming.factory.initial"  
386 locked="false">org.apache.qpid.jndi.PropertiesFileInitialContextFactory</parameter>  
387 <parameter name="java.naming.provider.url"  
388 locked="false">resources/jndi.properties</parameter>  
389 <parameter name="transport.jms.ConnectionFactoryJNDIName"  
390 locked="false">QueueConnectionFactory</parameter>  
391 <parameter name="transport.jms.ConnectionFactoryType"  
392 locked="false">queue</parameter>  
393 </parameter>  
394 </transportReceiver>
```

395
396 You can find my copy of the edited axis2.xml here:
397 <http://people.wso2.com/~paul/mb-guide-1.0/>
398

399 If you have looked through the JMS config you will notice it references a JNDI
400 resource: resources/jndi.properties.
401

402 This is used to do the same thing the hard-coded properties we used above do –
403 configure the local JNDI that the JMS client inside the ESB will use. In a future
404 release of the ESB we expect to automatically configure this JNDI, but in the
405 meantime, we can simply create a file in the <WSO2ESB>/resources directory.
406

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

```
408  
409 connectionfactory.TopicConnectionFactory = \  
410 amqp://admin:admin@carbon/carbon?brokerlist='tcp://localhost:5672'  
411 connectionfactory.QueueConnectionFactory = \  
412 amqp://admin:admin@carbon/carbon?brokerlist='tcp://localhost:5672'  
413 destination.dynamicQueues/myqueue=jmsdestinationqueue; {create:always}  
414 destination.myqueue=jmsdestinationqueue; {create:always}
```

415
416

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

419

420 **Check if you need the last two lines?**

421

422 You can find this file here:

423 <http://people.apache.org/~pzf/MB/>

424

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

427

428

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

431

432

433

434

435

436

437

438

439

440

441

442

443

444

445

446

447

448

449

450

451

452

453

454

455

456

457

458

459

460

461

462

463

464

465

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.

466
467
468
469
470

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**.

The screenshot shows the WSO2 Message Broker web console interface. The top navigation bar includes the WSO2 logo and the text "Message Broker". On the right, it says "Signed-in as: admin@". The left-hand menu is divided into "Configure" and "Manage" sections. Under "Configure", there are links for "Users and Roles", "Features", "Key Stores", "Logging", and "Server Roles". Under "Manage", there are links for "Topics", "Subscriptions", "Messageboxes/SQS", and "Queues". The "Queues" link is highlighted. The main content area is titled "Queue Browser" and contains a table with the following data:

Queue Name	Queue Depth	Message Count	Created Time	Updated
Java2WSDLService	0(b)	0	Fri Apr 01 09:14:05 BST 2011	Fri Apr 01 09:14:05 BST 2011
WSDL2CodeService	0(b)	0	Fri Apr 01 09:14:05 BST 2011	Fri Apr 01 09:14:05 BST 2011
WSDLConverterService	0(b)	0	Fri Apr 01 09:14:05 BST 2011	Fri Apr 01 09:14:05 BST 2011
WSDLValidatorService	0(b)	0	Fri Apr 01 09:14:05 BST 2011	Fri Apr 01 09:14:05 BST 2011
XKMS	0(b)	0	Fri Apr 01 09:14:05 BST 2011	Fri Apr 01 09:14:05 BST 2011
echo	0(b)	0	Fri Apr 01 09:14:05 BST 2011	Fri Apr 01 09:14:05 BST 2011
samplequeue	0(b)	0	Thu Mar 31 17:01:01 BST 2011	Thu Mar 31 17:01:01 BST 2011
testJMS	0(b)	0	Fri Apr 01 09:14:05 BST 2011	Fri Apr 01 09:14:05 BST 2011
version	0(b)	0	Fri Apr 01 09:14:05 BST 2011	Fri Apr 01 09:14:05 BST 2011
wso2carbon-sts	0(b)	0	Fri Apr 01 09:14:05 BST 2011	Fri Apr 01 09:14:05 BST 2011

471
472

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.

473
474
475
476
477
478
479
480

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.

481
482

Here is the proxy definition for the ESB:

483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503

```
<?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&java.naming.factory.initial=org.apache.qpid.jndi.PropertiesFileIn
        itialContextFactory&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>
```

504
505

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

506
507
508

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

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

514

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

518

519 The JMS URL is made up of:

520

```
jms:/myqueue
```

521

Look for a JNDI entry "myqueue"

522

(see jndi properties above)

523

```
?
```

524

Separator indicating extra attributes

525

526

```
transport.jms.ConnectionFactoryJNDIName=QueueConnectionFactory
```

527

Look up ConnectionFactory in JNDI with name

528

QueueConnectionFactory

529

```
&
```

530

Separator (this will convert to '&')

531

```
java.naming.factory.initial=
```

532

```
org.apache.qpid.jndi.PropertiesFileInitialContextFactory
```

533

Use the Qpid properties-based JNDI we saw earlier

534

```
&
```

535

Another separator

536

```
java.naming.provider.url=resources/jndi.properties
```

537

*Look in **resources/jndi.properties** for the JNDI properties file*

538

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

541

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

542

543

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

545

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

546

547

All the rest of the config is completely default.

548

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

```
551  
552 samplexml.xml  
553 <test xmlns="http://fremantle.org">  
554   <sample>data</sample>  
555 </test>
```

556
557 Once again you can find this file here:
558 <http://people.apache.org/~pzf/MB/>

559
560 For the next step, please ensure you have a copy of *curl* installed. If you are on
561 Linux or Mac you will have it by default. On Windows you can find a free version
562 on the web. Let's try the request against the ESB:

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

565
566 Run this a few times just for fun. You won't see much. If all is going well, you
567 won't see any errors on the WSO2 ESB console either. If you add '-v' to the curl
568 command line you will see a lot more information about the HTTP section of the
569 flow and you should see a nice sign that things are going well:

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

571
572 Now go back to the MB console and look at the **Queue→List** page. You should
573 now see some messages in the queue:

574

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	

575

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

577

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

581

582

583 **Conclusion**

584 There is a lot more we can do with MB. In future articles I hope to cover using a
585 C# client to interact, using the SQS support, and how the MB code can be
586 embedded directly into the ESB to provide in-process queueing and eventing. In
587 the meantime, I hope this has provided a simple introduction to get you started
588 with WSO2 MB.
589

