Messaging on JBoss

JMS Configuration and Architecture

The JMS API stands for Java Message Service Application Programming Interface, and it is used by applications to send asynchronous *business-quality* messages to other applications. In the messaging world, messages are not sent directly to other applications. Instead, messages are sent to destinations, known as queues or topics. Applications sending messages do not need to worry if the receiving applications are up and running, and conversely, receiving applications do not need to worry about the sending application's status. Both senders, and receivers only interact with the destinations.

The JMS API is the standardized interface to a JMS provider, sometimes called a Message Oriented Middleware (MOM) system. JBoss comes with a JMS 1.1 compliant JMS provider called JBoss Messaging or JBossMQ. When you use the JMS API with JBoss, you are using the JBoss Messaging engine transparently. JBoss Messaging fully implements the JMS specification; therefore, the best JBoss Messaging user guide is the JMS specification. For more information about the JMS API please visit the JMS Tutorial or JMS Downloads & Specifications.

This chapter focuses on the JBoss specific aspects of using JMS and message driven beans as well as the JBoss Messaging configuration and MBeans.

6.1. JMS Examples

In this section we discuss the basics needed to use the JBoss JMS implementation. JMS leaves the details of accessing JMS connection factories and destinations as provider specific details. What you need to know to use the JBoss Messaging layer is:

- The location of the queue and topic connect factories: In JBoss both connection factory implementations are located under the JNDI name `ConnectionFactory`.
- How to lookup JMS destinations (queues and topics): Destinations are configured via MBeans as we will see when we discuss the messaging MBeans. JBoss comes with a few queues and topics preconfigured. You can find them under the `jboss.mq.destination` domain in the JMX Console..
- Which JARS JMS requires: These include `concurrent.jar`, `jbossmq-client.jar`, `jboss-common-client.jar`, `jboss-system-client.jar`, `jnp-client.jar` and `log4j.jar`.

In the following sections we will look at examples of the various JMS messaging models and message driven beans. The chapter example source is located under the`src/main/org/jboss/chap6` directory of the book examples.

6.1.1. A Point-To-Point Example

Let's start out with a point-to-point (P2P) example. In the P2P model, a sender delivers messages to a queue and a single receiver pulls the message off of the queue. The receiver does not need to be listening to the queue at the time the message is

sent. Example 6.1, "A P2P JMS client example" shows a complete P2P example that sends a javax.jms.TextMessage to the queue queue/testQueue and asynchronously receives the message from the same queue.

Example 6.1. A P2P JMS client example

```
package org.jboss.chap6.ex1;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueReceiver;
import javax.jms.QueueSender;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import EDU.oswego.cs.dl.util.concurrent.CountDown;
import org.apache.log4j.Logger;
import org.jboss.util.ChapterExRepository;

/**
 * A complete JMS client example program that sends a
 * TextMessage to a Queue and asynchronously receives the
 * message from the same Queue.
 *
 * @author  Scott.Stark@jboss.org
 * @version $Revision: 1.9 $
 */
public class SendRecvClient
{
   static Logger log;
   static CountDown done = new CountDown(1);

   QueueConnection conn;
   QueueSession session;
   Queue que;

   public static class ExListener
      implements MessageListener
   {
      public void onMessage(Message msg)
      {
         done.release();
         TextMessage tm = (TextMessage) msg;
         try {
```

```
            log.info("onMessage, recv text=" + tm.getText());
         } catch(Throwable t) {
            t.printStackTrace();
         }
      }
   }

   public void setupPTP()
      throws JMSException,
            NamingException
   {
      InitialContext iniCtx = new InitialContext();
      Object tmp = iniCtx.lookup("ConnectionFactory");
      QueueConnectionFactory qcf = (QueueConnectionFactory) tmp;
      conn = qcf.createQueueConnection();
      que = (Queue) iniCtx.lookup("queue/testQueue");
      session = conn.createQueueSession(false,
                                 QueueSession.AUTO_ACKNOWLEDGE);
      conn.start();
   }

   public void sendRecvAsync(String text)
      throws JMSException,
            NamingException
   {
      log.info("Begin sendRecvAsync");
      // Setup the PTP connection, session
      setupPTP();

      // Set the async listener
      QueueReceiver recv = session.createReceiver(que);
      recv.setMessageListener(new ExListener());

      // Send a text msg
      QueueSender send = session.createSender(que);
      TextMessage tm = session.createTextMessage(text);
      send.send(tm);
      log.info("sendRecvAsync, sent text=" + tm.getText());
      send.close();
      log.info("End sendRecvAsync");
   }

   public void stop()
      throws JMSException
   {
      conn.stop();
      session.close();
      conn.close();
   }
```

```
   public static void main(String args[])
      throws Exception
   {
      ChapterExRepository.init(SendRecvClient.class);
      log = Logger.getLogger("SendRecvClient");

      log.info("Begin SendRecvClient, now=" + System.currentTimeMillis());
      SendRecvClient client = new SendRecvClient();
      client.sendRecvAsync("A text msg");
      client.done.acquire();
      client.stop();
      log.info("End SendRecvClient");
      System.exit(0);
   }
}
```

The client may be run using the following command line:

```
[examples]$ ant -Dchap=chap6 -Dex=1p2p run-example
...
run-example1p2p:
    [java] [INFO,SendRecvClient] Begin SendRecvClient, now=1102808673386
    [java] [INFO,SendRecvClient] Begin sendRecvAsync
    [java] [INFO,SendRecvClient] onMessage, recv text=A text msg
    [java] [INFO,SendRecvClient] sendRecvAsync, sent text=A text msg
    [java] [INFO,SendRecvClient] End sendRecvAsync
    [java] [INFO,SendRecvClient] End SendRecvClient
```

6.1.2. A Pub-Sub Example

The JMS publish/subscribe (Pub-Sub) message model is a one-to-many model. A publisher sends a message to a topic and all active subscribers of the topic receive the message. Subscribers that are not actively listening to the topic will miss the published message. shows a complete JMS client that sends a javax.jms.TextMessage to a topic and asynchronously receives the message from the same topic.

Example 6.2. A Pub-Sub JMS client example

```
package org.jboss.chap6.ex1;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicPublisher;
import javax.jms.TopicSubscriber;
```

```
import javax.jms.TopicSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import EDU.oswego.cs.dl.util.concurrent.CountDown;

/**
 *  A complete JMS client example program that sends a TextMessage to
 *  a Topic and asynchronously receives the message from the same
 *  Topic.
 *
 *  @author Scott.Stark@jboss.org
 *  @version $Revision: 1.9 $
 */

public class TopicSendRecvClient
{
   static CountDown done = new CountDown(1);
   TopicConnection conn = null;
   TopicSession session = null;
   Topic topic = null;

   public static class ExListener implements MessageListener
   {
      public void onMessage(Message msg)
      {
         done.release();
         TextMessage tm = (TextMessage) msg;
         try {
            System.out.println("onMessage, recv text=" + tm.getText());
         } catch(Throwable t) {
            t.printStackTrace();
         }
      }
   }

   public void setupPubSub()
      throws JMSException, NamingException
   {
      InitialContext iniCtx = new InitialContext();
      Object tmp = iniCtx.lookup("ConnectionFactory");
      TopicConnectionFactory tcf = (TopicConnectionFactory) tmp;
      conn = tcf.createTopicConnection();
      topic = (Topic) iniCtx.lookup("topic/testTopic");
      session = conn.createTopicSession(false,
                                        TopicSession.AUTO_ACKNOWLEDGE);
      conn.start();
   }
```

```
   public void sendRecvAsync(String text)
      throws JMSException, NamingException
   {
      System.out.println("Begin sendRecvAsync");
      // Setup the PubSub connection, session
      setupPubSub();
      // Set the async listener

      TopicSubscriber recv = session.createSubscriber(topic);
      recv.setMessageListener(new ExListener());
      // Send a text msg
      TopicPublisher send = session.createPublisher(topic);
      TextMessage tm = session.createTextMessage(text);
      send.publish(tm);
      System.out.println("sendRecvAsync, sent text=" + tm.getText());
      send.close();
      System.out.println("End sendRecvAsync");
   }

   public void stop() throws JMSException
   {
      conn.stop();
      session.close();
      conn.close();
   }

   public static void main(String args[]) throws Exception
   {
      System.out.println("Begin TopicSendRecvClient, now=" +
                         System.currentTimeMillis());
      TopicSendRecvClient client = new TopicSendRecvClient();
      client.sendRecvAsync("A text msg, now="+System.currentTimeMillis());
      client.done.acquire();
      client.stop();
      System.out.println("End TopicSendRecvClient");
      System.exit(0);
   }

}
```

The client may be run using the following command line:

```
[examples]$ ant -Dchap=chap6 -Dex=1ps run-example
...
run-example1ps:
    [java] Begin TopicSendRecvClient, now=1102809427043
    [java] Begin sendRecvAsync
    [java] onMessage, recv text=A text msg, now=1102809427071
    [java] sendRecvAsync, sent text=A text msg, now=1102809427071
```

```
    [java] End sendRecvAsync
    [java] End TopicSendRecvClient
```

Now let's break the publisher and subscribers into separate programs to demonstrate that subscribers only receive messages while they are listening to a topic. Example 6.3, "A JMS publisher client" shows a variation of the previous pub-sub client that only publishes messages to the topic/testTopic topic. The subscriber only client is shown inExample 6.3, "A JMS publisher client".

Example 6.3. A JMS publisher client

```
package org.jboss.chap6.ex1;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicPublisher;
import javax.jms.TopicSlistubscriber;
import javax.jms.TopicSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/**
 *  A JMS client example program that sends a TextMessage to a Topic
 *
 *  @author Scott.Stark@jboss.org
 *  @version $Revision: 1.9 $
 */
public class TopicSendClient
{
   TopicConnection conn = null;
   TopicSession session = null;
   Topic topic = null;

   public void setupPubSub()
      throws JMSException, NamingException
   {
      InitialContext iniCtx = new InitialContext();
      Object tmp = iniCtx.lookup("ConnectionFactory");
      TopicConnectionFactory tcf = (TopicConnectionFactory) tmp;
      conn = tcf.createTopicConnection();
      topic = (Topic) iniCtx.lookup("topic/testTopic");
      session = conn.createTopicSession(false,
                                        TopicSession.AUTO_ACKNOWLEDGE);
      conn.start();
```

```
    }

    public void sendAsync(String text)
        throws JMSException, NamingException
    {
        System.out.println("Begin sendAsync");
        // Setup the pub/sub connection, session
        setupPubSub();
        // Send a text msg
        TopicPublisher send = session.createPublisher(topic);
        TextMessage tm = session.createTextMessage(text);
        send.publish(tm);
        System.out.println("sendAsync, sent text=" +  tm.getText());
        send.close();
        System.out.println("End sendAsync");
    }

    public void stop()
        throws JMSException
    {
        conn.stop();
        session.close();
        conn.close();
    }

    public static void main(String args[])
        throws Exception
    {
        System.out.println("Begin TopicSendClient, now=" +
                                    System.currentTimeMillis());
        TopicSendClient client = new TopicSendClient();
                client.sendAsync("A text msg, now="+System.currentTimeMillis());
        client.stop();
        System.out.println("End TopicSendClient");
        System.exit(0);
    }

}
```

Example 6.4. A JMS subscriber client

```
package org.jboss.chap6.ex1;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
```

```
import javax.jms.TopicPublisher;
import javax.jms.TopicSubscriber;
import javax.jms.TopicSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/**
 * A JMS client example program that synchronously receives a message a Topic
 *
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.9 $
 */
public class TopicRecvClient
{
   TopicConnection conn = null;
   TopicSession session = null;
   Topic topic = null;

   public void setupPubSub()
      throws JMSException, NamingException
   {
      InitialContext iniCtx = new InitialContext();
      Object tmp = iniCtx.lookup("ConnectionFactory");
      TopicConnectionFactory tcf = (TopicConnectionFactory) tmp;
      conn = tcf.createTopicConnection();
      topic = (Topic) iniCtx.lookup("topic/testTopic");
      session = conn.createTopicSession(false,
                                        TopicSession.AUTO_ACKNOWLEDGE);
      conn.start();
   }

   public void recvSync()
      throws JMSException, NamingException
   {
      System.out.println("Begin recvSync");
      // Setup the pub/sub connection, session
      setupPubSub();

      // Wait upto 5 seconds for the message
      TopicSubscriber recv = session.createSubscriber(topic);
      Message msg = recv.receive(5000);
      if (msg == null) {
         System.out.println("Timed out waiting for msg");
      } else {
         System.out.println("TopicSubscriber.recv, msgt="+msg);
      }
   }

   public void stop()
```

```
        throws JMSException
    {
        conn.stop();
        session.close();
        conn.close();
    }

    public static void main(String args[])
        throws Exception
    {
        System.out.println("Begin TopicRecvClient, now=" +
                           System.currentTimeMillis());
        TopicRecvClient client = new TopicRecvClient();
        client.recvSync();
        client.stop();
        System.out.println("End TopicRecvClient");
        System.exit(0);
    }

}
```

Run the TopicSendClient followed by the TopicRecvClient as follows:

```
[examples]$ ant -Dchap=chap6 -Dex=1ps2 run-example
...
run-example1ps2:
     [java] Begin TopicSendClient, now=1102810007899
     [java] Begin sendAsync
     [java] sendAsync, sent text=A text msg, now=1102810007909
     [java] End sendAsync
     [java] End TopicSendClient
     [java] Begin TopicRecvClient, now=1102810011524
     [java] Begin recvSync
     [java] Timed out waiting for msg
     [java] End TopicRecvClient
```

The output shows that the topic subscriber client (TopicRecvClient) fails to receive the message sent by the publisher due to a timeout.

6.1.3. A Pub-Sub With Durable Topic Example

JMS supports a messaging model that is a cross between the P2P and pub-sub models. When a pub-sub client wants to receive all messages posted to the topic it subscribes to even when it is not actively listening to the topic, the client may achieve this behavior using a durable topic. Let's look at a variation of the preceding subscriber client that uses a durable topic to ensure that it receives all messages, include those published when the client is not listening to the topic. Example 6.5, "A durable topic JMS client

example" shows the durable topic client with the key differences between the Example 6.4, "A JMS subscriber client" client highlighted in bold.

Example 6.5. A durable topic JMS client example

```
package org.jboss.chap6.ex1;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicPublisher;
import javax.jms.TopicSubscriber;
import javax.jms.TopicSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/**
 *  A JMS client example program that synchronously receives a message a Topic
 *
 *  @author Scott.Stark@jboss.org
 *  @version $Revision: 1.9 $
 */
public class DurableTopicRecvClient
{
   TopicConnection conn = null;
   TopicSession session = null;
   Topic topic = null;

   public void setupPubSub()
      throws JMSException, NamingException
   {
      InitialContext iniCtx = new InitialContext();
      Object tmp = iniCtx.lookup("ConnectionFactory");

      TopicConnectionFactory tcf = (TopicConnectionFactory) tmp;
      conn = tcf.createTopicConnection("john", "needle");
      topic = (Topic) iniCtx.lookup("topic/testTopic");

      session = conn.createTopicSession(false,
                                        TopicSession.AUTO_ACKNOWLEDGE);
      conn.start();
   }

   public void recvSync()
      throws JMSException, NamingException
   {
```

```
        System.out.println("Begin recvSync");
        // Setup the pub/sub connection, session
        setupPubSub();
        // Wait upto 5 seconds for the message
        TopicSubscriber recv = session.createDurableSubscriber(topic, "chap6-ex1dtps");
        Message msg = recv.receive(5000);
        if (msg == null) {
            System.out.println("Timed out waiting for msg");
        } else {
            System.out.println("DurableTopicRecvClient.recv, msgt=" + msg);
        }
    }

    public void stop()
        throws JMSException
    {
        conn.stop();
        session.close();
        conn.close();
    }

    public static void main(String args[])
        throws Exception
    {
        System.out.println("Begin DurableTopicRecvClient, now=" +
                           System.currentTimeMillis());
        DurableTopicRecvClient client = new DurableTopicRecvClient();
        client.recvSync();
        client.stop();
        System.out.println("End DurableTopicRecvClient");
        System.exit(0);
    }

}
```

Now run the previous topic publisher with the durable topic subscriber as follows:

```
[examples]$ ant -Dchap=chap6 -Dex=1psdt run-example
...
run-example1psdt:
     [java] Begin DurableTopicSetup
     [java] End DurableTopicSetup
     [java] Begin TopicSendClient, now=1102899834273
     [java] Begin sendAsync
     [java] sendAsync, sent text=A text msg, now=1102899834345
     [java] End sendAsync
     [java] End TopicSendClient
     [java] Begin DurableTopicRecvClient, now=1102899840043
     [java] Begin recvSync
```

```
    [java] DurableTopicRecvClient.recv, msgt=SpyTextMessage {
    [java] Header {
    [java]    jmsDestination  : TOPIC.testTopic.DurableSubscription[
           clientId=DurableSubscriberExample name=chap6-ex1dtps selector=null]
    [java]    jmsDeliveryMode : 2
    [java]    jmsExpiration   : 0
    [java]    jmsPriority     : 4
    [java]    jmsMessageID    : ID:3-11028998375501
    [java]    jmsTimeStamp    : 1102899837550
    [java]    jmsCorrelationID: null
    [java]    jmsReplyTo      : null
    [java]    jmsType         : null
    [java]    jmsRedelivered  : false
    [java]    jmsProperties   : {}
    [java]    jmsPropReadWrite: false
    [java]    msgReadOnly     : true
    [java]    producerClientId: ID:3
    [java] }
    [java] Body {
    [java]    text           :A text msg, now=1102899834345
    [java] }
    [java] }
    [java] End DurableTopicRecvClient
```

Items of note for the durable topic example include:

- The TopicConnectionFactory creation in the durable topic client used a username and password, and the TopicSubscriber creation was done using thecreateDurableSubscriber(Topic, String) method. This is a requirement of durable topic subscribers. The messaging server needs to know what client is requesting the durable topic and what the name of the durable topic subscription is. We will discuss the details of durable topic setup in the configuration section.
- An org.jboss.chap6.DurableTopicSetup client was run prior to the TopicSendClient. The reason for this is a durable topic subscriber must have registered a subscription at some point in the past in order for the messaging server to save messages. JBoss supports dynamic durable topic subscribers and the DurableTopicSetup client simply creates a durable subscription receiver and then exits. This leaves an active durable topic subscriber on the topic/testTopic and the messaging server knows that any messages posted to this topic must be saved for latter delivery.
- The TopicSendClient does not change for the durable topic. The notion of a durable topic is a subscriber only notion.
- The DurableTopicRecvClient sees the message published to the topic/testTopic even though it was not listening to the topic at the time the message was published.

6.1.4. A Point-To-Point With MDB Example

Example 6.6, "A TextMessage processing MDB" shows an message driven bean (MDB) that transforms the TextMessages it receives and sends the transformed messages to the queue found in the incoming message JMSReplyTo header.

Example 6.6. A TextMessage processing MDB

```
package org.jboss.chap6.ex2;

import javax.ejb.MessageDrivenBean;
import javax.ejb.MessageDrivenContext;
import javax.ejb.EJBException;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSender;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/**
 * An MDB that transforms the TextMessages it receives and send the
 * transformed messages to the Queue found in the incoming message
 * JMSReplyTo header.
 *
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.9 $
 */
public class TextMDB
   implements MessageDrivenBean, MessageListener
{
   private MessageDrivenContext ctx = null;
   private QueueConnection conn;
   private QueueSession session;

   public TextMDB()
   {
      System.out.println("TextMDB.ctor, this="+hashCode());
   }

   public void setMessageDrivenContext(MessageDrivenContext ctx)
   {
      this.ctx = ctx;
      System.out.println("TextMDB.setMessageDrivenContext, this=" +
                         hashCode());
   }
```

```
    public void ejbCreate()
    {
        System.out.println("TextMDB.ejbCreate, this="+hashCode());
        try {
            setupPTP();
        } catch (Exception e) {
            throw new EJBException("Failed to init TextMDB", e);
        }
    }

    public void ejbRemove()
    {
        System.out.println("TextMDB.ejbRemove, this="+hashCode());
        ctx = null;
        try {
            if (session != null) {
                session.close();
            }
            if (conn != null) {
                conn.close();
            }
        } catch(JMSException e) {
            e.printStackTrace();
        }
    }

    public void onMessage(Message msg)
    {
        System.out.println("TextMDB.onMessage, this="+hashCode());
        try {
            TextMessage tm = (TextMessage) msg;
            String text = tm.getText() + "processed by: "+hashCode();
            Queue dest = (Queue) msg.getJMSReplyTo();
            sendReply(text, dest);
        } catch(Throwable t) {
            t.printStackTrace();
        }
    }

    private void setupPTP()
        throws JMSException, NamingException
    {
        InitialContext iniCtx = new InitialContext();
        Object tmp = iniCtx.lookup("java:comp/env/jms/QCF");
        QueueConnectionFactory qcf = (QueueConnectionFactory) tmp;
        conn = qcf.createQueueConnection();
        session = conn.createQueueSession(false,
                                    QueueSession.AUTO_ACKNOWLEDGE);
        conn.start();
    }
```

```
   private void sendReply(String text, Queue dest)
      throws JMSException
   {
      System.out.println("TextMDB.sendReply, this=" +
                         hashCode() + ", dest="+dest);
      QueueSender sender = session.createSender(dest);
      TextMessage tm = session.createTextMessage(text);
      sender.send(tm);
      sender.close();
   }
}
```

The MDB ejb-jar.xml and jboss.xml deployment descriptors are shown in Example 6.7, "The MDB ejb-jar.xml descriptor" and Example 6.8, "The MDB jboss.xml descriptor".

Example 6.7. The MDB ejb-jar.xml descriptor

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC
       "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
       "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
   <enterprise-beans>
      <message-driven>
         <ejb-name>TextMDB</ejb-name>
         <ejb-class>org.jboss.chap6.ex2.TextMDB</ejb-class>
         <transaction-type>Container</transaction-type>
         <acknowledge-mode>AUTO_ACKNOWLEDGE</acknowledge-mode>
         <message-driven-destination>
            <destination-type>javax.jms.Queue</destination-type>
         </message-driven-destination>
         <res-ref-name>jms/QCF</res-ref-name>
         <resource-ref>
            <res-type>javax.jms.QueueConnectionFactory</res-type>
            <res-auth>Container</res-auth>
         </resource-ref>
      </message-driven>
   </enterprise-beans>
</ejb-jar>
```

Example 6.8. The MDB jboss.xml descriptor

```
<?xml version="1.0"?>
<jboss>
   <enterprise-beans>
      <message-driven>
         <ejb-name>TextMDB</ejb-name>
```

```
         <destination-jndi-name>queue/B</destination-jndi-name>
         <resource-ref>
            <res-ref-name>jms/QCF</res-ref-name>
            <jndi-name>ConnectionFactory</jndi-name>
         </resource-ref>
      </message-driven>
   </enterprise-beans>
</jboss>
```

Example 6.9, "A JMS client that interacts with the TextMDB" shows a variation of the P2P client that sends several messages to the queue/B destination and asynchronously receives the messages as modified by TextMDB from queue A.

Example 6.9. A JMS client that interacts with the TextMDB

```
package org.jboss.chap6.ex2;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueReceiver;
import javax.jms.QueueSender;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import EDU.oswego.cs.dl.util.concurrent.CountDown;

/**
 *  A complete JMS client example program that sends N TextMessages to
 *  a Queue B and asynchronously receives the messages as modified by
 *  TextMDB from Queue A.
 *
 *  @author Scott.Stark@jboss.org
 *  @version $Revision: 1.9 $
 */
public class SendRecvClient
{
   static final int N = 10;
   static CountDown done = new CountDown(N);

   QueueConnection conn;
   QueueSession session;
   Queue queA;
   Queue queB;
```

```
   public static class ExListener
      implements MessageListener
   {
      public void onMessage(Message msg)
      {
         done.release();
         TextMessage tm = (TextMessage) msg;
         try {
            System.out.println("onMessage, recv text="+tm.getText());
         } catch(Throwable t) {
            t.printStackTrace();
         }
      }
   }

   public void setupPTP()
      throws JMSException, NamingException
   {
      InitialContext iniCtx = new InitialContext();
      Object tmp = iniCtx.lookup("ConnectionFactory");
      QueueConnectionFactory qcf = (QueueConnectionFactory) tmp;
      conn = qcf.createQueueConnection();
      queA = (Queue) iniCtx.lookup("queue/A");
      queB = (Queue) iniCtx.lookup("queue/B");
      session = conn.createQueueSession(false,
                                 QueueSession.AUTO_ACKNOWLEDGE);
      conn.start();
   }

   public void sendRecvAsync(String textBase)
      throws JMSException, NamingException, InterruptedException
   {
      System.out.println("Begin sendRecvAsync");

      // Setup the PTP connection, session
      setupPTP();

      // Set the async listener for queA
      QueueReceiver recv = session.createReceiver(queA);
      recv.setMessageListener(new ExListener());

      // Send a few text msgs to queB
      QueueSender send = session.createSender(queB);

      for(int m = 0; m < 10; m ++) {
         TextMessage tm = session.createTextMessage(textBase+"#"+m);
         tm.setJMSReplyTo(queA);
         send.send(tm);
         System.out.println("sendRecvAsync, sent text=" + tm.getText());
```

```
        }
        System.out.println("End sendRecvAsync");
    }

    public void stop()
        throws JMSException
    {
        conn.stop();
        session.close();
        conn.close();
    }

    public static void main(String args[])
        throws Exception
    {
        System.out.println("Begin SendRecvClient,now=" +
                           System.currentTimeMillis());
        SendRecvClient client = new SendRecvClient();
        client.sendRecvAsync("A text msg");
        client.done.acquire();
        client.stop();
        System.exit(0);
        System.out.println("End SendRecvClient");
    }

}
```

Run the client as follows:

```
[examples]$ ant -Dchap=chap6 -Dex=2 run-example
...
run-example2:
     [copy] Copying 1 file to /tmp/jboss-4.0.2/server/default/deploy
     [echo] Waiting 5 seconds for deploy...
     [java] Begin SendRecvClient, now=1102900541558
     [java] Begin sendRecvAsync
     [java] sendRecvAsync, sent text=A text msg#0
     [java] sendRecvAsync, sent text=A text msg#1
     [java] sendRecvAsync, sent text=A text msg#2
     [java] sendRecvAsync, sent text=A text msg#3
     [java] sendRecvAsync, sent text=A text msg#4
     [java] sendRecvAsync, sent text=A text msg#5
     [java] sendRecvAsync, sent text=A text msg#6
     [java] sendRecvAsync, sent text=A text msg#7
     [java] sendRecvAsync, sent text=A text msg#8
     [java] sendRecvAsync, sent text=A text msg#9
     [java] End sendRecvAsync
     [java] onMessage, recv text=A text msg#0processed by: 12855623
     [java] onMessage, recv text=A text msg#5processed by: 9399816
```

```
    [java] onMessage, recv text=A text msg#9processed by: 6598158
    [java] onMessage, recv text=A text msg#3processed by: 8153998
    [java] onMessage, recv text=A text msg#4processed by: 10118602
    [java] onMessage, recv text=A text msg#2processed by: 1792333
    [java] onMessage, recv text=A text msg#7processed by: 14251014
    [java] onMessage, recv text=A text msg#1processed by: 10775981
    [java] onMessage, recv text=A text msg#8processed by: 6056676
    [java] onMessage, recv text=A text msg#6processed by: 15679078
```

The corresponding JBoss server console output is:

```
19:15:40,232 INFO  [EjbModule] Deploying TextMDB
19:15:41,498 INFO  [EJBDeployer] Deployed: file:/private/tmp/jboss-
4.0.2/server/default/deplo
y/chap6-ex2.jar
19:15:45,606 INFO  [TextMDB] TextMDB.ctor, this=10775981
19:15:45,620 INFO  [TextMDB] TextMDB.ctor, this=1792333
19:15:45,627 INFO  [TextMDB] TextMDB.setMessageDrivenContext, this=10775981
19:15:45,638 INFO  [TextMDB] TextMDB.ejbCreate, this=10775981
19:15:45,640 INFO  [TextMDB] TextMDB.setMessageDrivenContext, this=1792333
19:15:45,640 INFO  [TextMDB] TextMDB.ejbCreate, this=1792333
19:15:45,649 INFO  [TextMDB] TextMDB.ctor, this=12855623
19:15:45,658 INFO  [TextMDB] TextMDB.setMessageDrivenContext, this=12855623
19:15:45,661 INFO  [TextMDB] TextMDB.ejbCreate, this=12855623
19:15:45,742 INFO  [TextMDB] TextMDB.ctor, this=8153998
19:15:45,744 INFO  [TextMDB] TextMDB.setMessageDrivenContext, this=8153998
19:15:45,744 INFO  [TextMDB] TextMDB.ejbCreate, this=8153998
19:15:45,763 INFO  [TextMDB] TextMDB.ctor, this=10118602
19:15:45,764 INFO  [TextMDB] TextMDB.setMessageDrivenContext, this=10118602
19:15:45,764 INFO  [TextMDB] TextMDB.ejbCreate, this=10118602
19:15:45,777 INFO  [TextMDB] TextMDB.ctor, this=9399816
19:15:45,779 INFO  [TextMDB] TextMDB.setMessageDrivenContext, this=9399816
19:15:45,779 INFO  [TextMDB] TextMDB.ejbCreate, this=9399816
19:15:45,792 INFO  [TextMDB] TextMDB.ctor, this=15679078
19:15:45,798 INFO  [TextMDB] TextMDB.setMessageDrivenContext, this=15679078
19:15:45,799 INFO  [TextMDB] TextMDB.ejbCreate, this=15679078
19:15:45,815 INFO  [TextMDB] TextMDB.ctor, this=14251014
19:15:45,816 INFO  [TextMDB] TextMDB.setMessageDrivenContext, this=14251014
19:15:45,817 INFO  [TextMDB] TextMDB.ejbCreate, this=14251014
19:15:45,829 INFO  [TextMDB] TextMDB.ctor, this=6056676
19:15:45,831 INFO  [TextMDB] TextMDB.setMessageDrivenContext, this=6056676
19:15:45,864 INFO  [TextMDB] TextMDB.ctor, this=6598158
19:15:45,903 INFO  [TextMDB] TextMDB.ejbCreate, this=6056676
19:15:45,906 INFO  [TextMDB] TextMDB.setMessageDrivenContext, this=6598158
19:15:45,906 INFO  [TextMDB] TextMDB.ejbCreate, this=6598158
19:15:46,236 INFO  [TextMDB] TextMDB.onMessage, this=12855623
19:15:46,238 INFO  [TextMDB] TextMDB.sendReply, this=12855623, dest=QUEUE.A
19:15:46,734 INFO  [TextMDB] TextMDB.onMessage, this=9399816
19:15:46,736 INFO  [TextMDB] TextMDB.onMessage, this=8153998
```

```
19:15:46,737 INFO  [TextMDB] TextMDB.onMessage, this=6598158
19:15:46,768 INFO  [TextMDB] TextMDB.sendReply, this=9399816, dest=QUEUE.A
19:15:46,768 INFO  [TextMDB] TextMDB.sendReply, this=6598158, dest=QUEUE.A
19:15:46,774 INFO  [TextMDB] TextMDB.sendReply, this=8153998, dest=QUEUE.A
19:15:46,903 INFO  [TextMDB] TextMDB.onMessage, this=10118602
19:15:46,904 INFO  [TextMDB] TextMDB.sendReply, this=10118602, dest=QUEUE.A
19:15:46,927 INFO  [TextMDB] TextMDB.onMessage, this=1792333
19:15:46,928 INFO  [TextMDB] TextMDB.sendReply, this=1792333, dest=QUEUE.A
19:15:47,002 INFO  [TextMDB] TextMDB.onMessage, this=14251014
19:15:47,007 INFO  [TextMDB] TextMDB.sendReply, this=14251014, dest=QUEUE.A
19:15:47,051 INFO  [TextMDB] TextMDB.onMessage, this=10775981
19:15:47,051 INFO  [TextMDB] TextMDB.sendReply, this=10775981, dest=QUEUE.A
19:15:47,060 INFO  [TextMDB] TextMDB.onMessage, this=6056676
19:15:47,061 INFO  [TextMDB] TextMDB.sendReply, this=6056676, dest=QUEUE.A
19:15:47,064 INFO  [TextMDB] TextMDB.onMessage, this=15679078
19:15:47,065 INFO  [TextMDB] TextMDB.sendReply, this=15679078, dest=QUEUE.A
```

Items of note in this example include:

- The JMS client has no explicit knowledge that it is dealing with an MDB. The client simply uses the standard JMS APIs to send messages to a queue and receive messages from another queue.
- The MDB declares whether it will listen to a queue or topic in the ejb-jar.xml descriptor. The name of the queue or topic must be specified using a jboss.xml descriptor. In this example the MDB also sends messages to a JMS queue. MDBs may act as queue senders or topic publishers within their onMessage callback.
- The messages received by the client include a "processed by: NNN" suffix, where NNN is the hashCode value of the MDB instance that processed the message. This shows that many MDBs may actively process messages posted to a destination. Concurrent processing is one of the benefits of MDBs.

6.2. JBoss Messaging Overview

JBossMQ is composed of several services working together to provide JMS API level services to client applications. The services that make up the JBossMQ JMS implementation are introduced in this section.

6.2.1. Invocation Layer

The Invocation Layer (IL) services are responsible for handling the communication protocols that clients use to send and receive messages. JBossMQ can support running different types of Invocation Layers concurrently. All Invocation Layers support bidirectional communication which allows clients to send and receive messages concurrently. ILs only handle the transport details of messaging. They delegate messages to the JMS server JMX gateway service known as the invoker. This is similar to how the detached invokers expose the EJB container via different transports.

Each IL service binds a JMS connection factory to a specific location in the JNDI tree. Clients choose the protocol they wish to use by the JNDI location used to obtain the JMS connection factory. JBossMQ currently has several different invocation layers.

- **UIL2 IL**: The Unified Invocation Layer version 2(UIL2) is the preferred invocation layer for remote messaging. A multiplexing layer is used to provide bidirectional communication. The multiplexing layer creates two virtual sockets over one physical socket. This allows communication with clients that cannot have a connection created from the server back to the client due to firewall or other restrictions. Unlike the older UIL invocation layer which used a blocking round-trip message at the socket level, the UIL2 protocol uses true asynchronous send and receive messaging at the transport level, providing for improved throughput and utilization.
- **JVM IL**: The Java Virtual Machine (JVM) Invocation Layer was developed to cut out the TCP/IP overhead when the JMS client is running in the same JVM as the server. This IL uses direct method calls for the server to service the client requests. This increases efficiency since no sockets are created and there is no need for the associated worker threads. This is the IL that should be used by Message Driven Beans (MDB) or any other component that runs in the same virtual machine as the server such as servlets, MBeans, or EJBs.
- **HTTP IL**: The HTTP Invocation Layer (HTTPIL) allows for accessing the JBossMQ service over the HTTP or HTTPS protocols. This IL relies on the servlet deployed in thedeploy/jms/jbossmq-httpil.sar to handle the http traffic. This IL is useful for access to JMS through a firewall when the only port allowed requires HTTP.

6.2.2. Security Manager

The JBossMQ SecurityManager is the service that enforces an access control list to guard access to your destinations. This subsystem works closely with the StateManagerservice.

6.2.3. Destination Manager

The DestinationManager can be thought as being the central service in JBossMQ. It keeps track of all the destinations that have been created on the server. It also keeps track of the other key services such as the MessageCache, StateManager, and PersistenceManager.

6.2.4. Message Cache

Messages created in the server are passed to the MessageCache for memory management. JVM memory usage goes up as messages are added to a destination that does not have any receivers. These messages are held in the main memory until the receiver picks them up. If the MessageCache notices that the JVM memory usage starts passing the defined limits, the MessageCache starts moving those messages from memory to persistent storage on disk. The MessageCache uses a least recently used (LRU) algorithm to determine which messages should go to disk.

6.2.5. State Manager

The StateManager (SM) is in charge of keeping track of who is allowed to log into the server and what their durable subscriptions are.

6.2.6. Persistence Manager

The PersistenceManager (PM) is used by a destination to store messages marked as being persistent. JBossMQ has several different implementations of the persistent manager, but only one can be enabled per server instance. You should enable the persistence manager that best matches your requirements.

- **JDBC2 persistence manager**: The JDBC2 persistence manager allows you to store persistent messages to a relational database using JDBC. The performance of this PM is directly related to the performance that can be obtained from the database. This PM has a very low memory overhead compared to the other persistence managers. Furthermore it is also highly integrated with the MessageCache to provide efficient persistence on a system that has a very active MessageCache.
- **Null Persistence Manager**: A wrapper persistence manager that can delegate to a real persistence manager. Configuration on the destinations decide whether persistence and caching is actually performed. The example configuration can be found in docs/examples/jms. To use the null persistence manager backed by a real persistence manager, you need to change the ObjectName of the real persistence manager and link the new name to the null persistence manager.

6.2.7. Destinations

A destination is the object on the JBossMQ server that clients use to send and receive messages. There are two types of destination objects, Queues and Topics. References to the destinations created by JBossMQ are stored in JNDI.

6.2.7.1. Queues

Clients that are in the point-to-point paradigm typically use queues. They expect that message sent to a queue will be receive by only one other client once and only once. If multiple clients are receiving messages from a single queue, the messages will be load balanced across the receivers. Queue objects, by default, will be stored under the JNDIqueue/ sub context.

6.2.7.2. Topics

Topics are used in the publish-subscribe paradigm. When a client publishes a message to a topic, he expects that a copy of the message will be delivered to each client that has subscribed to the topic. Topic messages are delivered in the same manner a television show is delivered. Unless you have the TV on and are watching the show, you will miss it. Similarly, if the client is not up, running and receiving messages from the topics, it will miss messages published to the topic. To get around this problem of missing messages, clients can start a durable subscription. This is like having a VCR record a show you cannot watch at its scheduled time so that you can see what you missed when you turn your TV back on.

6.3. JBoss Messaging Configuration and MBeans

This section defines the MBean services that correspond to the components introduced in the previous section along with their MBean attributes. The configuration and service files that make up the JBossMQ system include:

- **deploy/hsqldb-jdbc-state-service.xml**: This configures the JDBC state service for storing state in the embedded Hypersonic database.
- **deploy/jms/hsqldb-jdbc2-service.xml**: This service descriptor configures the DestinationManager, MessageCache, and jdbc2 PersistenceManager for the embedded Hypersonic database.
- **deploy/jms/jbossmq-destinations-service.xml**: This service describes defines default JMS queue and topic destination configurations used by the testsuite unit tests. You can add/remove destinations to this file, or deploy another *-service.xml descriptor with the destination configurations.
- **jbossmq-httpil.sar**: This SAR file configures the HTTP invocation layer.
- **deploy/jms/jbossmq-service.xml**: This service descriptor configures the core JBossMQ MBeans like the Invoker, SecurityManager, DynamicStateManager, and core interceptor stack. It also defines the MDB default dead letter queue DLQ.
- **deploy/jms/jms-ds.xml**: This is a JCA connection factory and JMS provider MDB integration services configuration which sets JBossMQ as the JMS provider.
- **deploy/jms/jms-ra.rar**: This is a JCA resource adaptor for JMS providers.
- **deploy/jms/jvm-il-service.xml**: This service descriptor configures the JVMServerILService which provides the JVM IL transport.
- **deploy/jms/rmi-il-service.xml**: This service descriptor configures the RMIServerILService which provides the RMI IL. The queue and topic connection factory for this IL is bound under the name RMIConnectionFactory.
- **deploy/jms/uil2-service.xml**: This service descriptor configures the UILServerILService which provides the UIL2 transport. The queue and topic connection factory for this IL is bound under the name UIL2ConnectionFactory as well as UILConnectionFactory to replace the deprecated version 1 UIL service.

We will discuss the associated MBeans in the following subsections.

6.3.1. org.jboss.mq.il.jvm.JVMServerILService

The org.jboss.mq.il.jvm.JVMServerILService MBean is used to configure the JVM IL. The configurable attributes are as follows:

- **Invoker**: This attribute specifies JMX ObjectName of the JMS entry point service that is used to pass incoming requests to the JMS server. This is not something you would typically change from the jboss.mq:service=Invoker setting unless you change the entry point service.
- **ConnectionFactoryJNDIRef**: The JNDI location that this IL will bind a ConnectionFactory setup to use this IL.
- **XAConnectionFactoryJNDIRef**: The JNDI location that this IL will bind a XAConnectionFactory setup to use this IL.
- **PingPeriod**: How often, in milliseconds, the client should send a ping message to the server to validate that the connection is still valid. If this is set to zero,

then no ping message will be sent. Since it is impossible for JVM IL connection to go bad, it is recommended that you keep this set to 0.

6.3.2. org.jboss.mq.il.uil2.UILServerILService

The org.jboss.mq.il.uil2.UILServerILService is used to configure the UIL2 IL. The configurable attributes are as follows:

- **Invoker**: This attribute specifies JMX ObjectName of the JMS entry point service that is used to pass incoming requests to the JMS server. This is not something you would typically change from the jboss.mq:service=Invoker setting unless you change the entry point service.
- **ConnectionFactoryJNDIRef**: The JNDI location that this IL will bind a ConnectionFactory setup to use this IL.
- **XAConnectionFactoryJNDIRef**: The JNDI location that this IL will bind a XAConnectionFactory setup to use this IL.
- **PingPeriod**: How often, in milliseconds, the client should send a ping message to the server to validate that the connection is still valid. If this is set to zero, then no ping message will be sent.
- **ReadTimeout**: The period in milliseconds is passed onto as the SoTimeout value of the UIL2 socket. This allows detection of dead sockets that are not responsive and are not capable of receiving ping messages. Note that this setting should be longer in duration than the PingPeriod setting.
- **BufferSize**: The size in bytes used as the buffer over the basic socket streams. This corresponds to the java.io.BufferedOutputStream buffer size.
- **ChunkSize**: The size in bytes between stream listener notifications. The UIL2 layer uses the org.jboss.util.stream.NotifyingBufferedOutputStream andNotifyingBufferedInput Stream implementations that support the notion of a heartbeat that is triggered based on data read/written to the stream. Whenever ChunkSizebytes are read/written to a stream. This allows serves as a ping or keepalive notification when large reads or writes require a duration greater than the PingPeriod.
- **ServerBindPort**: The protocol listening port for this IL. If not specified default is 0, which means that a random port will be chosen.
- **BindAddress**: The specific address this IL listens on. This can be used on a multi-homed host for a java.net.ServerSocket that will only accept connection requests on one of its addresses.
- **EnableTcpNoDelay**: TcpNoDelay causes TCP/IP packets to be sent as soon as the request is flushed. This may improve request response times. Otherwise request packets may be buffered by the operating system to create larger IP packets.
- **ServerSocketFactory**: The javax.net.ServerSocketFactory implementation class name to use to create the service java.net.ServerSocket. If not specified the default factory will be obtained from javax.net.ServerSocketFactory.getDefault().
- **ClientAddress**: The address passed to the client as the address that should be used to connect to the server.
- **ClientSocketFactory**: The javax.net.SocketFactory implementation class name to use on the client. If not specified the default factory will be obtained fromjavax.net.SocketFactory.getDefault().

- **SecurityDomain**: Specify the security domain name to use with JBoss SSL aware socket factories. This is the JNDI name of the security manager implementation as described for the security-domain element of the jboss.xml and jboss-web.xml descriptors in Section 8.3.1, "Enabling Declarative Security in JBoss Revisited".

6.3.2.1. Configuring UIL2 for SSL

The UIL2 service support the use of SSL through custom socket factories that integrate JSSE using the security domain associated with the IL service. An example UIL2 service descriptor fragment that illustrates the use of the custom JBoss SSL socket factories is shown in Example 6.10, "An example UIL2 config fragment for using SSL".

Example 6.10. An example UIL2 config fragment for using SSL

```
<mbean code="org.jboss.mq.il.uil2.UILServerILService"
   name="jboss.mq:service=InvocationLayer,type=HTTPSUIL2">
   <depends optional-attribute-
name="Invoker">jboss.mq:service=Invoker</depends>
   <attribute name="ConnectionFactoryJNDIRef">SSLConnectionFactory</attribute>
   <attribute
name="XAConnectionFactoryJNDIRef">SSLXAConnectionFactory</attribute>

   <!-- ... -->

   <!-- SSL Socket Factories -->
   <attribute name="ClientSocketFactory">
      org.jboss.security.ssl.ClientSocketFactory
   </attribute>
   <attribute name="ServerSocketFactory">
      org.jboss.security.ssl.DomainServerSocketFactory
   </attribute>
   <!-- Security domain - see below -->
   <attribute name="SecurityDomain">java:/jaas/SSL</attribute>
</mbean>

<!-- Configures the keystore on the "SSL" security domain
    This mbean is better placed in conf/jboss-service.xml where it
    can be used by other services, but it will work from anywhere.
    Use keytool from the sdk to create the keystore. -->

<mbean code="org.jboss.security.plugins.JaasSecurityDomain"
      name="jboss.security:service=JaasSecurityDomain,domain=SSL">
   <!-- This must correlate with the java:/jaas/SSL above -->
   <constructor>
      <arg type="java.lang.String" value="SSL"/>
   </constructor>
   <!-- The location of the keystore resource: loads from the
        classpath and the server conf dir is a good default -->
```

```
   <attribute name="KeyStoreURL">resource:uil2.keystore</attribute>
   <attribute name="KeyStorePass">changeme</attribute>
</mbean>
```

6.3.2.2. JMS client properties for the UIL2 transport

There are several system properties that a JMS client using the UIL2 transport can set to control the client connection back to the server

- **org.jboss.mq.il.uil2.useServerHost**: This system property allows a client to connect to the server InetAddress.getHostName rather than theInetAddress.getHostAddressvalue. This will only make a difference if name resolution differs between the server and client environments.
- **org.jboss.mq.il.uil2.localAddr**: This system property allows a client to define the local interface to which its sockets should be bound.
- **org.jboss.mq.il.uil2.localPort**: This system property allows a client to define the local port to which its sockets should be bound
- **org.jboss.mq.il.uil2.serverAddr**: This system property allows a client to override the address to which it attempts to connect to. This is useful for networks where NAT is occcurring between the client and JMS server.
- **org.jboss.mq.il.uil2.serverPort**: This system property allows a client to override the port to which it attempts to connect. This is useful for networks where port forwarding is occurring between the client and jms server.
- **org.jboss.mq.il.uil2.retryCount**: This system property controls the number of attempts to retry connecting to the JMS server. Retries are only made forjava.net.ConnectException failures. A value <= 0 means no retry attempts will be made.
- **org.jboss.mq.il.uil2.retryDelay**: This system property controls the delay in milliseconds between retries due to ConnectException failures.

6.3.3. org.jboss.mq.il.http.HTTPServerILService

The org.jboss.mq.il.http.HTTPServerILService is used to manage the HTTP/S IL. This IL allows for the use of the JMS service over HTTP or HTTPS connections. The relies on the servlet deployed in the deploy/jms/jbossmq-httpil.sar to handle the HTTP traffic. The configurable attributes are as follows:

- **TimeOut**: The default timeout in seconds that the client HTTP requests will wait for messages. This can be overridden on the client by setting the system propertyorg.jboss.mq.il.http.timeout to the number of seconds.
- **RestInterval**: The number of seconds the client will sleep after each request. The default is 0, but you can set this value in conjunction with the TimeOut value to implement a pure timed based polling mechanism. For example, you could simply do a short lived request by setting the TimeOut value to 0 and then setting theRestInterval to 60. This would cause the client to send a single non-blocking request to the server, return any messages if available, then sleep for 60 seconds, before issuing another request. Like the TimeOut value, this can be explicitly overridden on a given client by specifying

the org.jboss.mq.il.http.restinterval with the number of seconds you wish to wait between requests.
- **URL**: Set the servlet URL. This value takes precedence over any individual values set (i.e. the URLPrefix, URLSuffix, URLPort, etc.) It my be a actual URL or a property name which will be used on the client side to resolve the proper URL by calling System.getProperty(propertyname). If not specified the URL will be formed from URLPrefix + URLHostName + ":" + URLPort + "/" + URLSuffix.
- **URLPrefix**: The prefix portion of the servlet URL.
- **URLHostName**: The hostname portion of the servlet URL.
- **URLPort**: The port portion of the URL.
- **URLSuffix**: The trailing path portion of the URL.
- **UseHostName**: A flag that if set to true the default setting for the URLHostName attribute will be taken from InetAddress.getLocalHost().getHostName(). If false the default setting for the URLHostName attribute will be taken from InetAddress.getLocalHost().getHostAddress().

6.3.4. org.jboss.mq.server.jmx.Invoker

The org.jboss.mq.server.jmx.Invoker is used to pass IL requests down to the destination manager service through an interceptor stack. The configurable attributes are as follows:

- **NextInterceptor**: The JMX ObjectName of the next request interceptor. This attribute is used by all the interceptors to create the interceptor stack. The last interceptor in the chain should be the DestinationManager.

6.3.5. org.jboss.mq.server.jmx.InterceptorLoader

The org.jboss.mq.server.jmx.InterceptorLoader is used to load a generic interceptor and make it part of the interceptor stack. This MBean is typically used to load custom interceptors like org.jboss.mq.server.TracingInterceptor, which is can be used to efficiently log all client requests via trace level log messages. The configurable attributes are as follows:

- **NextInterceptor**: The JMX ObjectName of the next request interceptor. This attribute is used by all the interceptors to create the interceptor stack. The last interceptor in the chain should be the DestinationManager. This attribute should be setup via a <depends optional-attribute-name="NextInterceptor"> XML tag.
- **InterceptorClass**: The class name of the interceptor that will be loaded and made part of the interceptor stack. This class specified here must extend theorg.jboss.mq.server.JMSServerInterceptor class.

6.3.6. org.jboss.mq.sm.jdbc.JDBCStateManager

The JDBCStateManager MBean is used as the default state manager assigned to the DestinationManager service. It stores user and durable subscriber information in the database. The configurable attributes are as follows:

- **ConnectionManager**: This is the ObjectName of the datasource that the JDBC state manager will write to. For Hypersonic, it isjboss.jca:service=DataSourceBinding,name=DefaultDS.
- **SqlProperties**: The SqlProperties define the SQL statements to be used to persist JMS state data. If the underlying database is changed, the SQL statements used may need to change.

6.3.7. org.jboss.mq.security.SecurityManager

If the org.jboss.mq.security.SecurityManager is part of the interceptor stack, then it will enforce the access control lists assigned to the destinations. The SecurityManager uses JAAS, and as such requires that at application policy be setup for in the JBoss login-config.xml file. The default configuration is shown below.

```
<application-policy name="jbossmq">
   <authentication>
      <login-module code="org.jboss.security.auth.spi.DatabaseServerLoginModule"
                    flag="required">
         <module-option name="unauthenticatedIdentity">guest</module-option>
         <module-option name="dsJndiName">java:/DefaultDS</module-option>
         <module-option name="principalsQuery">SELECT PASSWD FROM JMS_USERS
            WHERE USERID=?</module-option>
         <module-option name="rolesQuery">SELECT ROLEID, 'Roles' FROM
            JMS_ROLES WHERE USERID=?</module-option>
      </login-module>
   </authentication>
</application-policy>
```

The configurable attributes of the SecurityManager are as follows:

- **NextInterceptor**: The JMX ObjectName of the next request interceptor. This attribute is used by all the interceptors to create the interceptor stack. The last interceptor in the chain should be the DestinationManager.
- **SecurityDomain**: Specify the security domain name to use for authentication and role based authorization. This is the JNDI name of the JAAS domain to be used to perform authentication and authorization against.
- **DefaultSecurityConfig**: This element specifies the default security configuration settings for destinations. This applies to temporary queues and topics as well as queues and topics that do not specifically specify a security configuration. The DefaultSecurityConfig should declare some number of role elements which represent each role that is allowed access to a destination. Each role should have the following attributes:
    - **name**: The name attribute defines the name of the role.
    - **create**: The create attribute is a true/false value that indicates whether the role has the ability to create durable subscriptions on the topic.
    - **read**: The read attribute is a true/false value that indicates whether the role can receive messages from the destination.
    - **write**: The write attribute is a true/false value that indicates whether the role can send messages to the destination.

6.3.8. org.jboss.mq.server.jmx.DestinationManager

The org.jboss.mq.server.jmx.DestinationManager must be the last interceptor in the interceptor stack. The configurable attributes are as follows:

- **PersistenceManager**: The JMX ObjectName of the persistence manager service the server should use.
- **StateManager**: The JMX ObjectName of the state manager service the server should use.
- **MessageCache**: The JMX ObjectName of the message cache service the server should use.

Additional read-only attributes and operations that support monitoring include:

- **ClientCount**: The number of clients connected to the server.
- **Clients**: A java.util.Map<org.jboss.mq.ConnectionToken, org.jboss.mq.server.ClientConsumer> instances for the clients connected to the server.
- **MessageCounter**: An array of org.jboss.mq.server.MessageCounter instances that provide statistics for a JMS destination.
- **listMessageCounter()**: This operation generates an HTML table that contains:
  - **Type**: Either Queue or Topic indicating the destination type.
  - **Name**: The name of the destination.
  - **Subscription**: The subscription ID for a topic.
  - **Durable**: A boolean indicating if the topic subscription is durable.
  - **Count**: The number of message delivered to the destination.
  - **CountDelta**: The change in message count since the previous access of count.
  - **Depth**: The number of messages in the destination.
  - **DepthDelta**: The change in the number of messages in the destination since the previous access of depth.
  - **Last Add**: The date/time string in DateFormat.SHORT/DateFormat.MEDIUM format of the last time a message was added to the destination.
- **resetMessageCounter()**: This zeros all destination counts and last added times.

Queues and topics can be created and destroyed at runtime through the DestinationManager MBean.
The DestinationManager provides createQueue and createTopic operations for this. Both methods have a one argument version which takes the destination name and a two argument version which takes the destination and the JNDI name of the destination. Queues and topics can be removed using the destroyQueue and destroyTopic operations, both of which take a destination name is input.

6.3.9. org.jboss.mq.server.MessageCache

The server determines when to move messages to secondary storage by using the org.jboss.mq.server.MessageCache MBean. The configurable attributes are as follows:

- **CacheStore**: The JMX ObjectName of the service that will act as the cache store. The cache store is used by the MessageCache to move messages to persistent storage. The value you set here typically depends on the type of persistence manager you are using.
- **HighMemoryMark**: The amount of JVM heap memory in megabytes that must be reached before the MessageCache starts to move messages to secondary storage.
- **MaxMemoryMark**: The maximum amount of JVM heap memory in megabytes that the MessageCache considers to be the max memory mark. As memory usage approaches the max memory mark, the MessageCache will move messages to persistent storage so that the number of messages kept in memory approaches zero.
- **MakeSoftReferences**: This controls whether or not the message cache will keep soft references to messages that need to be removed. The default is true.
- **MinimumHard**: The minimum number of the in memory cache. JBoss won't try to go below this number of messages in the cache. The default value is 1.
- **MaximumHard**: The upper bound on the number of hard references to messages in the cache. JBoss will soften messages to reduce the number of hard references to this level. A value of 0 means that there is no size based upper bound. The default is 0.
- **SoftenWaitMillis**: The maximum wait time before checking whether messages need softening. The default is 1000 milliseconds (1 second).
- **SoftenNoMoreOftenThanMillis**: The minimum amount of time between checks to soften messages. A value of 0 means that this check should be skipped. The default is 0 milliseconds.
- **SoftenAtLeastEveryMillis**: The maximum amount of time between checks to soften messages. A value of 0 means that this check should be skipped. The default is 0.

Additional read-only cache attribute that provide statistics include:

- **CacheHits**: The number of times a hard referenced message was accessed
- **CacheMisses**: The number of times a softened message was accessed.
- **HardRefCacheSize**: The number of messages in the cache that are not softened.
- **SoftRefCacheSize**: The number of messages that are currently softened.
- **SoftenedSize**: The total number of messages softened since the last boot.
- **TotalCacheSize**: The total number of messages that are being managed by the cache.

6.3.10. org.jboss.mq.pm.jdbc2.PersistenceManager

The org.jboss.mq.pm.jdbc.PersistenceManager should be used as the persistence manager assigned to the DestinationManager if you wish to store messages in a database. This PM has been tested against the HypersonSQL, MS SQL, Oracle, MySQL and Postgres databases. The configurable attributes are as follows:

- **MessageCache**: The JMX ObjectName of the MessageCache that has been assigned to the DestinationManager..
- **ConnectionManager**: The JMX ObjectName of the JCA data source that will be used to obtain JDBC connections.
- **ConnectionRetryAttempts**: An integer count used to allow the PM to retry attempts at getting a connection to the JDBC store. There is a 1500 millisecond delay between each connection failed connection attempt and the next attempt. This must be greater than or equal to 1 and defaults to 5.
- **SqlProperties**: A property list is used to define the SQL Queries and other JDBC2 Persistence Manager options. You will need to adjust these properties if you which to run against another database other than Hypersonic. Example 6.11, "Default JDBC2 PeristenceManager SqlProperties" shows default setting for this attribute for the Hypersonic database.

Example 6.11. Default JDBC2 PeristenceManager SqlProperties

```
<attribute name="SqlProperties">
    CREATE_TABLES_ON_STARTUP = TRUE
    CREATE_USER_TABLE = CREATE TABLE JMS_USERS \
        (USERID VARCHAR(32) NOT NULL, PASSWD VARCHAR(32) NOT NULL, \
        CLIENTID VARCHAR(128), PRIMARY KEY(USERID))
    CREATE_ROLE_TABLE = CREATE TABLE JMS_ROLES \
        (ROLEID VARCHAR(32) NOT NULL, USERID VARCHAR(32) NOT NULL, \
              PRIMARY KEY(USERID, ROLEID))
    CREATE_SUBSCRIPTION_TABLE = CREATE TABLE JMS_SUBSCRIPTIONS \
        (CLIENTID VARCHAR(128) NOT NULL, \
        SUBNAME VARCHAR(128) NOT NULL, TOPIC VARCHAR(255) NOT NULL, \
        SELECTOR VARCHAR(255), PRIMARY KEY(CLIENTID, SUBNAME))
    GET_SUBSCRIPTION = SELECT TOPIC, SELECTOR FROM JMS_SUBSCRIPTIONS \
        WHERE CLIENTID=? AND SUBNAME=?
    LOCK_SUBSCRIPTION = SELECT TOPIC, SELECTOR FROM JMS_SUBSCRIPTIONS \
        WHERE CLIENTID=? AND SUBNAME=?
    GET_SUBSCRIPTIONS_FOR_TOPIC =
        SELECT CLIENTID, SUBNAME, SELECTOR FROM JMS_SUBSCRIPTIONS WHERE
TOPIC=?
    INSERT_SUBSCRIPTION = \
        INSERT INTO JMS_SUBSCRIPTIONS (CLIENTID, SUBNAME, TOPIC, SELECTOR)
VALUES(?,?,?,?)
    UPDATE_SUBSCRIPTION = \
        UPDATE JMS_SUBSCRIPTIONS SET TOPIC=?, SELECTOR=? WHERE CLIENTID=?
AND SUBNAME=?
    REMOVE_SUBSCRIPTION = DELETE FROM JMS_SUBSCRIPTIONS WHERE
CLIENTID=? AND SUBNAME=?
    GET_USER_BY_CLIENTID = SELECT USERID, PASSWD, CLIENTID FROM
JMS_USERS WHERE CLIENTID=?
    GET_USER = SELECT PASSWD, CLIENTID FROM JMS_USERS WHERE USERID=?
    POPULATE.TABLES.01 = INSERT INTO JMS_USERS (USERID, PASSWD) \
              VALUES ('guest', 'guest')
    POPULATE.TABLES.02 = INSERT INTO JMS_USERS (USERID, PASSWD) \
              VALUES ('j2ee', 'j2ee')
```

```
    POPULATE.TABLES.03 = INSERT INTO JMS_USERS (USERID, PASSWD, CLIENTID) \
                VALUES ('john', 'needle', 'DurableSubscriberExample')
    POPULATE.TABLES.04 = INSERT INTO JMS_USERS (USERID, PASSWD) \
                VALUES ('nobody', 'nobody')
    POPULATE.TABLES.05 = INSERT INTO JMS_USERS (USERID, PASSWD) \
                VALUES ('dynsub', 'dynsub')
    POPULATE.TABLES.06 = INSERT INTO JMS_ROLES (ROLEID, USERID) \
                VALUES ('guest','guest')
    POPULATE.TABLES.07 = INSERT INTO JMS_ROLES (ROLEID, USERID) \
                VALUES ('j2ee','guest')
    POPULATE.TABLES.08 = INSERT INTO JMS_ROLES (ROLEID, USERID) \
                VALUES ('john','guest')
    POPULATE.TABLES.09 = INSERT INTO JMS_ROLES (ROLEID, USERID) \
                VALUES ('subscriber','john')
    POPULATE.TABLES.10 = INSERT INTO JMS_ROLES (ROLEID, USERID) \
                VALUES ('publisher','john')
    POPULATE.TABLES.11 = INSERT INTO JMS_ROLES (ROLEID, USERID) \
                VALUES ('publisher','dynsub')
    POPULATE.TABLES.12 = INSERT INTO JMS_ROLES (ROLEID, USERID) \
                VALUES ('durpublisher','john')
    POPULATE.TABLES.13 = INSERT INTO JMS_ROLES (ROLEID, USERID) \
                VALUES ('durpublisher','dynsub')
    POPULATE.TABLES.14 = INSERT INTO JMS_ROLES (ROLEID, USERID) \
                VALUES ('noacc','nobody')
</attribute>
```

Example 6.12, “A sample JDBC2 PeristenceManager SqlProperties for Oracle” shows an alternate setting for Oracle.

Example 6.12. A sample JDBC2 PeristenceManager SqlProperties for Oracle

```
<attribute name="SqlProperties">
    BLOB_TYPE=BINARYSTREAM_BLOB
    INSERT_TX = INSERT INTO JMS_TRANSACTIONS (TXID) values(?)
    INSERT_MESSAGE = \
       INSERT INTO JMS_MESSAGES (MESSAGEID, DESTINATION, MESSAGEBLOB,
TXID, TXOP) \
       VALUES(?,?,?,?,?)
    SELECT_ALL_UNCOMMITED_TXS = SELECT TXID FROM JMS_TRANSACTIONS
    SELECT_MAX_TX = SELECT MAX(TXID) FROM JMS_MESSAGES
    SELECT_MESSAGES_IN_DEST = \
       SELECT MESSAGEID, MESSAGEBLOB FROM JMS_MESSAGES WHERE
DESTINATION=?
    SELECT_MESSAGE = \
       SELECT MESSAGEID, MESSAGEBLOB FROM JMS_MESSAGES WHERE
MESSAGEID=? AND DESTINATION=?
    MARK_MESSAGE = \
       UPDATE JMS_MESSAGES SET TXID=?, TXOP=? WHERE MESSAGEID=? AND
DESTINATION=?
```

```
    UPDATE_MESSAGE = \
       UPDATE JMS_MESSAGES SET MESSAGEBLOB=? WHERE MESSAGEID=? AND
DESTINATION=?
    UPDATE_MARKED_MESSAGES = UPDATE JMS_MESSAGES SET TXID=?, TXOP=?
WHERE TXOP=?
    UPDATE_MARKED_MESSAGES_WITH_TX = \
       UPDATE JMS_MESSAGES SET TXID=?, TXOP=? WHERE TXOP=? AND TXID=?
    DELETE_MARKED_MESSAGES_WITH_TX = \
       DELETE FROM JMS_MESSAGES MESS WHERE TXOP=:1 AND EXISTS \
       (SELECT TXID FROM JMS_TRANSACTIONS TX WHERE TX.TXID = MESS.TXID)
    DELETE_TX = DELETE FROM JMS_TRANSACTIONS WHERE TXID = ?
    DELETE_MARKED_MESSAGES = DELETE FROM JMS_MESSAGES WHERE TXID=?
AND TXOP=?
    DELETE_TEMPORARY_MESSAGES = DELETE FROM JMS_MESSAGES WHERE
TXOP='T'
    DELETE_MESSAGE = DELETE FROM JMS_MESSAGES WHERE MESSAGEID=? AND
DESTINATION=?
    CREATE_MESSAGE_TABLE = CREATE TABLE JMS_MESSAGES ( MESSAGEID
INTEGER NOT NULL, \
      DESTINATION VARCHAR(255) NOT NULL, TXID INTEGER, TXOP CHAR(1), \
      MESSAGEBLOB BLOB, PRIMARY KEY (MESSAGEID, DESTINATION) )
    CREATE_IDX_MESSAGE_TXOP_TXID = \
      CREATE INDEX JMS_MESSAGES_TXOP_TXID ON JMS_MESSAGES (TXOP, TXID)
    CREATE_IDX_MESSAGE_DESTINATION = \
      CREATE INDEX JMS_MESSAGES_DESTINATION ON JMS_MESSAGES
(DESTINATION)
    CREATE_TX_TABLE = CREATE TABLE JMS_TRANSACTIONS ( TXID INTEGER,
PRIMARY KEY (TXID) )
    CREATE_TABLES_ON_STARTUP = TRUE
</attribute>
```

Additional examples can be found in the docs/examples/jms directory of the distribution.

6.3.11. Destination MBeans

This section describes the destination MBeans used in the jbossmq-destinations-service.xml and jbossmq-service.xml descriptors.

6.3.11.1. org.jboss.mq.server.jmx.Queue

The Queue is used to define a queue destination in JBoss. The following shows the configuration of one of the default JBoss queues.

```
<mbean code="org.jboss.mq.server.jmx.Queue"
      name="jboss.mq.destination:service=Queue,name=testQueue">
   <depends optional-attribute-name="DestinationManager">
      jboss.mq:service=DestinationManager
   </depends>
```

```
   <depends optional-attribute-name="SecurityManager">
      jboss.mq:service=SecurityManager
   </depends>
   <attribute name="MessageCounterHistoryDayLimit">-1</attribute>
   <attribute name="SecurityConf">
      <security>
         <role name="guest"     read="true"  write="true"/>
         <role name="publisher" read="true"  write="true" create="false"/>
         <role name="noacc"     read="false" write="false" create="false"/>
      </security>
   </attribute>
</mbean>
```

The name attribute of the JMX object name of this MBean is used to determine the destination name. For example. In the case of the queue we just looked at, the name of the queue is testQueue. The configurable attributes are as follows:

- **DestinationManager**: The JMX ObjectName of the destination manager service for the server. This attribute should be set via a <depends optional-attribute-name="DestinationManager"> XML tag.
- **SecurityManager**: The JMX ObjectName of the security manager service that is being used to validate client requests.
- **SecurityConf**: This element specifies a XML fragment which describes the access control list to be used by the SecurityManager to authorize client operations against the destination. The content model is the same as for the SecurityManager SecurityConf attribute.
- **JNDIName**: The location in JNDI to which the queue object will be bound. If this is not set it will be bound under the queue context using the name of the queue. For thetestQueue shown above, the JNDI name would be queue/testQueue.
- **MaxDepth**: The MaxDepth is an upper limit to the backlog of messages that can exist for a destination. If exceeded, attempts to add new messages will result in aorg.jboss.mq.DestinationFullException. The MaxDepth can still be exceeded in a number of situations, e.g. when a message is placed back into the queue. Also transactions performing read committed processing, look at the current size of queue, ignoring any messages that may be added as a result of the current transaction or other transactions. This is because we don't want the transaction to fail during the commit phase when the message is physically added to the queue.
- **MessageCounterHistoryDayLimit**: Sets the destination message counter history day limit with a value less than 0 indicating unlimited history, a 0 value disabling history and a value greater than 0 giving the history days count.

Additional read-only attributes that provide statistics information include:

- **MessageCounter**: An array of org.jboss.mq.server.MessageCounter instances that provide statistics for this destination.
- **QueueDepth**: The current backlog of waiting messages.
- **ReceiversCount**: The number of receivers currently associated with the queue.

- **ScheduledMessageCount**: The number of messages waiting in the queue for their scheduled delivery time to arrive.

The following are some of the operations available on queues.

- **listMessageCounter()**: This operation generates an HTML table that contains the same data we as the listMessageCounter operation on the DestinationManager, but only for this one queue.
- **resetMessageCounter()**: This zeros all destination counts and last added times.
- **listMessageCounterHistory()**: This operation display an HTML table showing the hourly message counts per hour for each day in the history.
- **resetMessageCounterHistory()**: This operation resets the day history message counts.
- **removeAllMessages()**: This method removes all the messages on the queue.

6.3.11.2. org.jboss.mq.server.jmx.Topic

The org.jboss.mq.server.jmx.Topic is used to define a topic destination in JBoss. The following shows the configuration of one of the default JBoss topics.

```
<mbean code="org.jboss.mq.server.jmx.Topic"
      name="jboss.mq.destination:service=Topic,name=testTopic">
   <depends optional-attribute-name="DestinationManager">
      jboss.mq:service=DestinationManager
   </depends>
   <depends optional-attribute-name="SecurityManager">
      jboss.mq:service=SecurityManager
   </depends>
   <attribute name="SecurityConf">
      <security>
         <role name="guest"       read="true" write="true" />
         <role name="publisher"    read="true" write="true" create="false" />
         <role name="durpublisher" read="true" write="true" create="true" />
      </security>
   </attribute>
</mbean>
```

The name attribute of the JMX object name of this MBean is used to determine the destination name. For example, in the case of the topic we just looked at, the name of the topic is testTopic. The configurable attributes are as follows:

- **DestinationManager**: The JMX object name of the destination manager configured for the server.
- **SecurityManager**: The JMX object name of the security manager that is being used to validate client requests.
- **SecurityConf**: This element specifies a XML fragment which describes the access control list to be used by the SecurityManager to authorize client

operations against the destination. The content model is the same as that for the SecurityManager SecurityConf attribute.
- **JNDIName**: The location in JNDI to which the topic object will be bound. If this is not set it will be bound under the topic context using the name of the queue. For thetestTopic shown above, the JNDI name would be topic/testTopic.
- **MaxDepth**: The MaxDepth is an upper limit to the backlog of messages that can exist for a destination, and if exceeded, attempts to add new messages will result in aorg.jboss.mq.DestinationFullException. The MaxDepth can still be exceeded in a number of situations, e.g. when a message is knacked back into the queue. Also transactions performing read committed processing, look at the current size of queue, ignoring any messages that may be added as a result of the current transaction or other transactions. This is because we don't want the transaction to fail during the commit phase when the message is physically added to the topic.
- **MessageCounterHistoryDayLimit**: Sets the destination message counter history day limit with a value < 0 indicating unlimited history, a 0 value disabling history, and a value > 0 giving the history days count.

Additional read-only attributes that provide statistics information include:

- **AllMessageCount**: The message count across all queue types associated with the topic.
- **AllSubscriptionsCount**: The count of durable and non-durable subscriptions.
- **DurableMessageCount**: The count of messages in durable subscription queues.
- **DurableSubscriptionsCount**: The count of durable subscribers.
- **MessageCounter**: An array of org.jboss.mq.server.MessageCounter instances that provide statistics for this destination.
- **NonDurableMessageCount**: The count on messages in non-durable subscription queues.
- **NonDurableSubscriptionsCount**: The count of non-durable subscribers.

The following are some of the operations available on queues.

- **listMessageCounter()**: This operation generates an HTML table that contains the same data we as the listMessageCounter operation on the DestinationManager, but only for this one topic.
- **resetMessageCounter()**: This zeros all destination counts and last added times.
- **listMessageCounterHistory()**: This operation display an HTML table showing the hourly message counts per hour for each day of history.
- **resetMessageCounterHistory()**: This operation resets the day history message counts.

6.4. Specifying the MDB JMS Provider

Up to this point we have looked at the standard JMS client/server architecture. The JMS specification defines an advanced set of interfaces that allow for concurrent processing of a destination's messages, and collectively this functionality is referred to as application server facilities (ASF). Two of the interfaces that support concurrent

message processing, javax.jms.ServerSessionPool and javax.jms.ServerSession, must be provided by the application server in which the processing will occur. Thus, the set of components that make up the JBossMQ ASF involves both JBossMQ components as well as JBoss server components. The JBoss server MDB container utilizes the JMS service's ASF to concurrently process messages sent to MDBs.

The responsibilities of the ASF domains are well defined by the JMS specification and so we won't go into a discussion of how the ASF components are implemented. Rather, we want to discuss how ASF components used by the JBoss MDB layer are integrated using MBeans that allow either the application server interfaces, or the JMS provider interfaces to be replaced with alternate implementations.

Let's start with the org.jboss.jms.jndi.JMSProviderLoader MBean. This MBean is responsible for loading an instance of the org.jboss.jms.jndi.JMSProviderAdaptor interface into the JBoss server and binding it into JNDI. The JMSProviderAdaptor interface is an abstraction that defines how to get the root JNDI context for the JMS provider, and an interface for getting and setting the JNDI names for the Context.PROVIDER_URL for the root InitialContext, and the QueueConnectionFactory and TopicConnectionFactory locations in the root context. This is all that is really necessary to bootstrap use of a JMS provider. By abstracting this information into an interface, alternate JMS ASF provider implementations can be used with the JBoss MDB container.
The org.jboss.jms.jndi.JBossMQProvider is the default implementation of JMSProviderAdaptor interface, and provides the adaptor for the JBossMQ JMS provider. To replace the JBossMQ provider with an alternate JMS ASF implementation, simply create an implementation of the JMSProviderAdaptor interface and configure the JMSProviderLoader with the class name of the implementation. We'll see an example of this in the configuration section.

In addition to being able to replace the JMS provider used for MDBs, you can also replace the javax.jms.ServerSessionPool interface implementation. This is possible by configuring the class name of the org.jboss.jms.asf.ServerSessionPoolFactory implementation using the org.jboss.jms.asf.ServerSessionPoolLoader MBean PoolFactoryClass attribute. The default ServerSessionPoolFactory factory implementation is the JBoss org.jboss.jms.asf.StdServerSessionPoolFactory class.

6.4.1. org.jboss.jms.jndi.JMSProviderLoader MBean

The JMSProviderLoader MBean service creates a JMS provider adaptor and binds it into JNDI. A JMS provider adaptor is a class that implements theorg.jboss.jms.jndi.JMSProviderAdapter interface. It is used by the message driven bean container to access a JMS service provider in a provider independent manner. The configurable attributes of the JMSProviderLoader service are:

- **ProviderName**: A unique name for the JMS provider. This will be used to bind the JMSProviderAdapter instance into JNDI under java:/<ProviderName> unless overridden by the AdapterJNDIName attribute.
- **ProviderAdapterClass**: The fully qualified class name of the org.jboss.jms.jndi.JMSProviderAdapter interface to create an instance of.

- **FactoryRef**: The JNDI name under which the provider javax.jms.ConnectionFactory will be bound.
- **QueueFactoryRef**: The JNDI name under which the provider javax.jms.QueueConnectionFactory will be bound.
- **TopicFactoryRef**: The JNDI name under which the javax.jms.TopicConnectionFactory will be bound.
- **Properties**: The JNDI properties of the initial context used to look up the factories.

Example 6.13. A JMSProviderLoader for accessing a remote JBossMQ server

```
<mbean code="org.jboss.jms.jndi.JMSProviderLoader"
      name="jboss.mq:service=JMSProviderLoader,name=RemoteJBossMQProvider">
   <attribute name="ProviderName">RemoteJMSProvider</attribute>
   <attribute name="ProviderUrl"></attribute>
   <attribute name="ProviderAdapterClass">
      org.jboss.jms.jndi.JBossMQProvider
   </attribute>
   <attribute name="FactoryRef">XAConnectionFactory</attribute>
   <attribute name="QueueFactoryRef">XAConnectionFactory</attribute>
   <attribute name="TopicFactoryRef">XAConnectionFactory</attribute>
   <attribute name="Properties>
      java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
      java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
      java.naming.provider.url=jnp://remotehost:1099
   </attribute>
</mbean>
```

The RemoteJMSProvider can be referenced on the MDB invoker config as shown in the jboss.xml fragment given in Example 6.14, " A jboss.xml fragment for specifying the MDB JMS provider adaptor".

Example 6.14. A jboss.xml fragment for specifying the MDB JMS provider adaptor

```
<proxy-factory-config>
   <JMSProviderAdapterJNDI>RemoteJMSProvider</JMSProviderAdapterJNDI>
   <ServerSessionPoolFactoryJNDI>StdJMSPool</ServerSessionPoolFactoryJNDI>
   <MaximumSize>15</MaximumSize>
   <MaxMessages>1</MaxMessages>
   <MDBConfig>
      <ReconnectIntervalSec>10</ReconnectIntervalSec>
      <DLQConfig>
         <DestinationQueue>queue/DLQ</DestinationQueue>
         <MaxTimesRedelivered>10</MaxTimesRedelivered>
         <TimeToLive>0</TimeToLive>
      </DLQConfig>
   </MDBConfig>
```

```
</proxy-factory-config>
```

Incidentally, because one can specify multiple invoker-proxy-binding elements, this allows an MDB to listen to the same queue/topic on multiple servers by configuring multiple bindings with different JMSProviderAdapterJNDI settings.

Alternatively, one can integrate the JMS provider using JCA configuration like that shown in Example 6.15, "A jms-ds.xml descriptor for integrating a JMS provider adaptor via JCA".

Example 6.15. A jms-ds.xml descriptor for integrating a JMS provider adaptor via JCA

```
<tx-connection-factory>
   <jndi-name>RemoteJmsXA</jndi-name>
   <xa-transaction/>
   <adapter-display-name>JMS Adapter</adapter-display-name>
   <config-property name="JMSProviderAdapterJNDI"
                type="java.lang.String">RemoteJMSProvider</config-property>
   <config-property name="SessionDefaultType"
                type="java.lang.String">javax.jms.Topic</config-property>

   <security-domain-and-application>JmsXARealm</security-domain-and-application>
</tx-connection-factory>
```

6.4.2. org.jboss.jms.asf.ServerSessionPoolLoader MBean

The ServerSessionPoolLoader MBean service manages a factory for javax.jms.ServerSessionPool objects used by the message driven bean container. The configurable attributes of the ServerSessionPoolLoader service are:

- **PoolName**: A unique name for the session pool. This will be used to bind the ServerSessionPoolFactory instance into JNDI under java:/PoolName.
- **PoolFactoryClass**: The fully qualified class name of the org.jboss.jms.asf.ServerSessionPoolFactory interface to create an instance of.
- **XidFactory**: The JMX ObjectName of the service to use for generating javax.transaction.xa.Xid values for local transactions when two phase commit is not required. TheXidFactory MBean must provide an Instance operation which returns a org.jboss.tm.XidFactoryMBean instance.

6.4.3. Integrating non-JBoss JMS Providers

We have mentioned that one can replace the JBossMQ JMS implementation with a foreign implementation. Here we summarize the various approaches one can take to do the replacement:

- Replace the JMSProviderLoader JBossMQProvider class with one that instantiates the correct JNDI context for communicating with the foreign JMS providers managed objects.
- Use the ExternalContext MBean to federate the foreign JMS providers managed objects into the JBoss JNDI tree.
- Use MBeans to instantiate the foreign JMS objects into the JBoss JNDI tree. An example of this approach can be found for Websphere MQ athttp://wiki.jboss.org/wiki/Wiki.jsp?page=IntegrationWithWebSphereMQSeries.