Java Message Service on HornetQ

PREVIOUS NOTE: In the fourth assignment you may want to use a JMS provider different from the one we present in this document. Be aware that some of the details (like creating queues or managing durable subscriptions) are very likely to be different. An ESB project in JBoss Developer Studio and the JBoss ESB 4.10 should be enough for most examples we provide here.

NOTE: your main source of documentation should be the complete user manual. You can also use the quick start guide, for a faster reference.

HornetQ is a messaging system that can run integrated in JBoss 6[1]. The default startup configuration of JBoss also initializes the HornetQ messaging system.
Let us start by creating an example with two programs that exchange a pair of text messages (Ping and Pong). For that purpose you should create a Java project and import the following source code, using a class named `Agent` in the `code` package.

```
package code;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Agent {
      protected InitialContext init;
      protected ConnectionFactory cf;
      protected Connection conn;
      protected Session session;

      public Agent() throws NamingException, JMSException {
```

[1] At the moment we write this document, JBoss 7 is out there already, but it seems to be poorly documented. Use it at your own risk.

```
            init = new InitialContext();
            cf = (ConnectionFactory) init.lookup("ConnectionFactory");
            conn = cf.createConnection();
            createSession();
            conn.start();
      }

      protected void createSession() throws JMSException {
            session = conn.createSession(false,
Session.AUTO_ACKNOWLEDGE);
      }

      public void send(String outboxname, String text)
                        throws NamingException, JMSException {
            Destination outbox = (Destination) init.lookup(outboxname);
            MessageProducer sndr = session.createProducer(outbox);
            TextMessage msg = session.createTextMessage(text);
            sndr.send(msg);
            sndr.close();
      }

      public String receive(String inboxname)
                        throws NamingException, JMSException {
            Destination inbox = (Destination) init.lookup(inboxname);
            MessageConsumer cnsmr = session.createConsumer(inbox);
            TextMessage replymsg = (TextMessage) cnsmr.receive();
            cnsmr.close();
            return replymsg.getText();
      }

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

}
```

This Agent class creates all objects up to the session in the constructor, and includes one method to send messages, another one to receive, and a final one to close the connection. You should never forget to close the connection or otherwise you may have to wait for a relatively long period of time before being able to connect to the same queue again. The class also includes a method called `createSession()`, which you can override to create a new agent with transacted sessions, if you want to try more complex examples.

Based on the Agent class we define Ping:

```
package code;


public class Ping extends Thread {
	private Agent ba;

	public Ping(Agent ba) {
		this.ba = ba;
	}

	public void run() {
		try {
			ba.send("/queue/Pong", "Hello Pong, my name is
Ping");
			String res = ba.receive("/queue/Ping");
			System.out.println("Ping got: " + res);
			ba.finish();
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
}
```

The Pong:

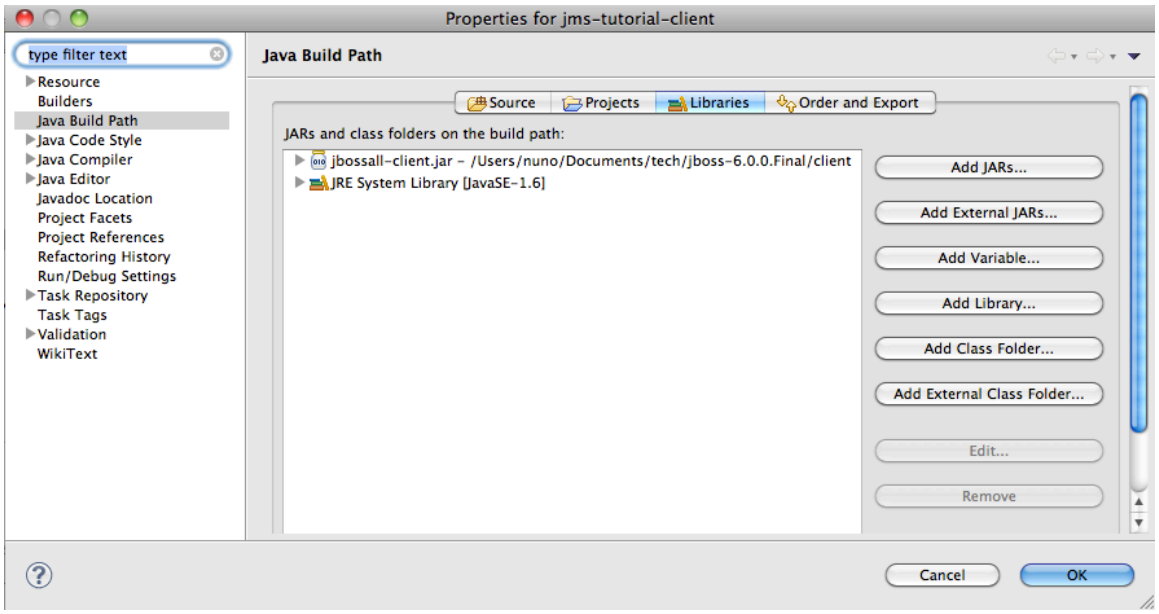
```
package code;

public class Pong extends Thread {
	Agent ba;

	public Pong(Agent ba) {
		this.ba = ba;
	}

	public void run() {
		try {
			String res = ba.receive("/queue/Pong");
			System.out.println("Pong got: " + res);
			ba.send("/queue/Ping", "Hello Ping, my name is
Pong");
			ba.finish();
		}
		catch (Exception e) {
			e.printStackTrace();
		}
```

```
	}
}
```

And finally the `main()` method. Note the differences in the packages, as this latter class belongs to the default package.

```
import javax.jms.JMSException;
import javax.naming.NamingException;

import code.Agent;
import code.Ping;
import code.Pong;


public class RunNonTransacted1 {

	public static void main(String[] args)
			throws NamingException, JMSException {
		Agent aping = new Agent();
		Ping pi = new Ping(aping);
		pi.start();

		Agent apong = new Agent();
		Pong po = new Pong(apong);
		po.start();
	}

}
```

If you used NetBeans or Eclipse, your source code should now be heavily underlined in red. This happens because you need a library to your project – jbossall-client.jar. You can find this library at $JBOSS_HOME/client (you can use the "Add External JARs" option to add it):



Unfortunately, the code is still not working… If you pay attention to the source, you will notice that the sender and receiver exchange messages through two queues named `/queue/Ping` and `/queue/Pong`. This can only work if we configure this queue in the file `hornetq-jms.xml` in the directory `$JBOSS_HOME/server/default/deploy/hornetq`. Fortunately, this is fairly easy (you may find a couple of differences comparing to the initial state of the same file in your installation, but only the addition of the last two queue elements is important):

```
<configuration xmlns="urn:hornetq"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="urn:hornetq /schema/hornetq-jms.xsd">

      <connection-factory name="NettyConnectionFactory">
         <connectors>
         <connector-ref connector-name="netty"/>
         </connectors>
            <entries>
                  <entry name="/ConnectionFactory"/>
                  <entry name="/XAConnectionFactory"/>
            </entries>
      </connection-factory>


      <topic name="PlayJMSTopic">
            <entry name="/topic/PlayJMSTopic"/>
      </topic>
```

```
        <queue name="DLQ">
              <entry name="/queue/DLQ"/>
        </queue>
        <queue name="ExpiryQueue">
              <entry name="/queue/ExpiryQueue"/>
        </queue>

        <queue name="Ping">
              <entry name="/queue/Ping"/>
        </queue>

        <queue name="Pong">
              <entry name="/queue/Pong"/>
        </queue>

</ configuration>
```

Again, this is not enough. You still need to define the symbol `java.naming.factory.initial` required by the Java Naming and Directory Information (JNDI) client (the messaging server automatically launches the JNDI server). For this you will need a file named `jndi.properties` with the following content:

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://localhost:1099
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
```

You must include this file in the root of your source folder directory (typically the 'src' folder). After this step, your program should now run.

As an optional step, it can be useful to have the JMS API documentation directly in your IDE. For this you can use the zipped source code of the JMS API (jms-api-1.1-source.zip) and simply add it to your project build path as you did with any other regular jar file earlier. After this step you will have the JMS API documentation available in your IDE.

Temporary Queues

In a client/server environment it may be useful to create temporary queues. A client can send a message to a server and specify (in that message) a temporary queue that the server application can use to put a response later on. This is

partially illustrated by the following code (`session` corresponds to a regular javax.jms.Session):

```
...
TextMessage message = session.createTextMessage("Hello!");
Queue replyQueue = session.createTemporaryQueue();
message.setJMSReplyTo(replyQueue);
...
```

After obtaining the message sent by the client, the server application can know where to send its reply by using `message.getJMSReplyTo()` to obtain a reference to the client's temporary queue.

Utilization of Topics

We now exemplify the utilization of topics in the Java Message Service. We will do it without touching our Agent class! This is a very interesting result from the Object Oriented features of JMS. Destination, Connection, Session, etc., are, in fact interfaces that can refer both to Queue objects and to Topic objects as well. For convenience, we reproduce here a table you can find in the Java EE 6 API. The only difference is set in the configuration file `hornetq-jms.xml` in the attribute `name` of the element `entry`. You can check for the topic `PlayJMSTopic` in the configuration file we showed above.

Relationship of PTP and Pub/Sub interfaces

JMS Common	PTP Domain	Pub/Sub Domain
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver	TopicSubscriber

You should be aware that these examples cover only the introductory utilization of JMS. You can find a lot more in other documentation.

In the topic example, we will have one publisher, and four subscribers (all running the same code). We start by the publisher:

```
package code;

public class Publisher extends Thread {
```

```
	private Agent ba;
	private String topicname;

	public Publisher(Agent ba, String topicname) {
		this.ba = ba;
		this.topicname = topicname;
	}

	public void run() {
		try {
			ba.send(this.topicname, "Hello folks, I think I am a
publisher");
			ba.finish();
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
}
```

Now the subscriber:

```
package code;

public class Subscriber extends Thread {
	private Agent ba;
	private String topicname;

	public Subscriber(Agent ba, String topicname) {
		this.ba = ba;
		this.topicname = topicname;
	}

	public void run() {
		try {
			String res = ba.receive(this.topicname);
			System.out.println("Subscriber 		"		+
Thread.currentThread().getName() + " got: " + res);
			ba.finish();
		}
		catch (Exception e) {
			e.printStackTrace();
		}
	}

}
```

And now, let's run them:

```
import javax.jms.JMSException;
import javax.naming.NamingException;

import code.Agent;
import code.Publisher;
import code.Subscriber;


public class RunNonTransacted2 {

	private final static int subscribers = 4;

	public static void main(String[] args) throws NamingException,
JMSException {
		String topic = "/topic/PlayJMSTopic";

		for (int i = 0; i < subscribers ; i++) {
			Agent sub = new Agent();
			Subscriber po = new Subscriber(sub, topic);
			po.start();
		}

		Agent pub = new Agent();
		Publisher pi = new Publisher(pub, topic);
		pi.start();

	}

}
```

Durable Subscriptions

By default, all messages arriving at a topic while the subscriber is switched off are lost. However, we can create durable subscriptions to force the server to store all messages going through a given topic for later delivery. For this to work, we must authenticate the client, set its identity and uniquely identify the subscription. The code highlighted in yellow shows what changes one has to do to the original code:

```
InitialContext initCtx = new InitialContext();
ConnectionFactory cf = (ConnectionFactory)
initCtx.lookup("ConnectionFactory");
```

```
this.conn = cf.createConnection(myname, passwd);
this.conn.setClientID(myname);
this.dest = (Destination) initCtx.lookup(topicName);
this.session = this.conn.createTopicSession(false,
TopicSession.AUTO_ACKNOWLEDGE);
this.conn.start();

MessageConsumer subs = this.session.createDurableSubscriber(this.dest,
subscriptionid);
```

To make this code runnable, students must now create the appropriate user and authorize this type of user to create durable subscriptions. Refer to the directory `conf/props`. The file `hornetq-users.properties` sets the users and their passwords, while the file `hornetq-roles.properties` assigns roles for the users. In the version of the JBoss we tried, a user named John existed already, with the password needle. You may try your luck with these credentials.

However, before it works you need to authorize the "guest" users to create durable subscriptions in the file `hornetq-configuration.xml`, (you should add the highlighted lines):

```
   <security-settings>
      <security-setting match="#">
         <permission type="createDurableQueue" roles="guest"/>
         <permission type="deleteDurableQueue" roles="guest"/>
         <permission type="createTempQueue" roles="guest"/>
         <permission type="deleteTempQueue" roles="guest"/>
         <permission type="consume" roles="guest"/>
         <permission type="send" roles="guest"/>
      </security-setting>
   </security-settings>
```

Bibliography

"Java Message Service". Mark Richards, Richard Monson-Haefel, David A. Chappell. O'Reilly Media; Second Edition edition (May 28, 2009)

http://download.oracle.com/javaee/6/api/