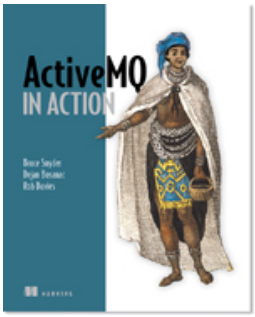
MANNING PUBLICATIONS



ActiveMQ in Action
By Bruce Snyder, Dejan Bosanac, and Rob Davies

Together ActiveMQ and Spring make an excellent Java Message Service (JMS) development platform, making many common tasks extremely easy to accomplish. In this article based on chapter 7 of ActiveMQ in Action, the authors discuss how you can configure JMS connections and destinations, define JMS consumers, and implement JMS producers.

You may also be interested in…

Creating Applications with ActiveMQ: Writing JMS clients Using Spring

ActiveMQ uses the Spring Framework to ease the various aspects of client-to-broker communication, but the Spring framework goes much further with its API and container design specifically for Java Message Service (JMS) messaging. Together, ActiveMQ and Spring make an excellent JMS development platform, making many common tasks extremely easy to accomplish. The tasks we'll cover in this article are:

- Configuring JMS connections—ActiveMQ provides classes that can be used to configure URLs and other parameters of connections to brokers. The connection factory could later be used by your application to get the appropriate connection.
- Configuring JMS destinations—ActiveMQ destination objects can be configured simply as beans representing JMS destinations used by your producers and consumers.
- Defining JMS consumers—Spring provides helper classes that allows you to easily configure a message listener container and hook message listeners to it.
- Implementing JMS producers—Spring also provides helper bean classes for creating new producers

In the following sections, these tasks will be demonstrated used to rewrite the portfolio application to use all benefits of the ActiveMQ and Spring integration.

Configuring JMS connections

The first step in creating a JMS application is to create a connection to the ActiveMQ broker. `ActiveMQConnectionFactory` is a factory from which to create an `ActiveMQConnection`, both of which can easily be used with Spring. The following snippet shows how to define an `ActiveMQConnectionFactory` using a Spring XML config:

```
<bean id="jmsConnectionFactory"
   class="org.apache.activemq.ActiveMQConnectionFactory">
 <property name="brokerURL" value="tcp://localhost:61616" />
 <property name="userName" value="admin" />
 <property name="password" value="password" />
</bean>
```

In the snippet above, notice the properties that are configured on the `ActiveMQConnectionFactory`.

As you will see further in this article, in certain use cases, a pool of connections is necessary in order to achieve a desired performance. For this purpose, ActiveMQ provides the `PooledConnectionFactory` class which mains a pool of JMS connections and sessions. Below is an example Spring XML configuration for the `PooledConnectionFactory`:

```
<bean id="pooledJmsConnectionFactory"
  class="org.apache.activemq.pool.PooledConnectionFactory"
  destroy-method="stop">
 <property name="connectionFactory" ref="jmsConnectionFactory" />
</bean>
```

There is only one property configured on the `PooledConnectionFactory`—the `connectionFactory` property. The `connectionFactory` property is used to define the underlying connection factory to the ActiveMQ broker that will be used by the pooled connection factory. In this case, we have used a previously defined `jmsConnectionFactory` bean.

Since the pooled connection factory has a dependency on the Apache Commons Pool project (http://commons.apache.org/pool/), you will need to add the JAR to the classpath. Or, if you use Maven for your project management, just add the following dependency to the `pom.xml` file:

```
<dependency>
 <groupId>commons-pool</groupId>
 <artifactId>commons-pool</artifactId>
 <version>1.4</version>
</dependency>
```

The XML above defines a Maven dependency on the commons-pool-1.4.jar file and it even fetches it for you automatically.

Once the JMS connection has been defined, you can move on to defining the JMS destinations, producers, and consumers.

Configuring JMS destinations

JMS destinations can be predefined in the `activemq.xml` file using `ActiveMQTopic` and the `ActiveMQQueue` classes. The following snippet contains two topic definitions to be used in the portfolio example.

```
<bean id="cscoDest" class="org.apache.activemq.command.ActiveMQTopic">
  <constructor-arg value="STOCKS.CSCO" />
</bean>

<bean id="orclDest" class="org.apache.activemq.command.ActiveMQTopic">
  <constructor-arg value="STOCKS.ORCL" />
</bean>
```

As you can see, these classes use just constructor injection for setting a desired destination name on the `ActiveMQTopic` class. Predefining topics is not required in ActiveMQ, but it can be handy for environments where the broker requires clients to authenticate for various operations. Now that a connection and a couple of destinations exist, you can begin sending and receiving messages.

Creating JMS consumers

The next two sections touch upon the basic use of Spring JMS for creating consumers and producers. We'll show you some of the basic concepts to get you up and running quickly with the portfolio example.

The basic abstraction for receiving messages in Spring is the message listener container (MLC). The MLC design provides an intermediary between your message listener and broker to handle connections, threading, and more, leaving you to worry only about your business logic that lives in the listener. In listing 1, a portfolio message listener is used by two message listener containers for two destinations.

Listing 1 Defining two Spring message listener containers and a message listener

```
<!-- The message listener -->
<bean id="portfolioListener"
  class="org.apache.activemq.book.ch3.portfolio.Listener">
</bean>

<!-- Spring DMLC -->
<bean id="cscoConsumer"
  class=
"org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="jmsConnectionFactory" />
    <property name="destination" ref="cscoDest" />
    <property name="messageListener" ref="portfolioListener" />
</bean>
```

```
  <!-- Spring DMLC -->
  <bean id="orclConsumer"
    class=
"org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="jmsConnectionFactory" />
    <property name="destination" ref="orclDest" />
    <property name="messageListener" ref="portfolioListener" />
  </bean>
```

Each MLC instance in listing 1 requires a connection factory, a destination, and a message listener. So, all you have to do is to implement a message listener bean and leave everything else to the Spring MLC. Note that in this example we have used the plain (not pooled) connection factory. This is because no connection pooling is needed for this simple example. This example makes use of the Spring which is the `DefaultMessageListenerContainer` most commonly used MLC. Although there are numerous other properties on the DMLC that can be configured, this example is using only the basics. When these two DMLC instances start up, they will be ready to receive messages and hand them off to the message listener.

Now let's send some messages to ActiveMQ.

Creating JMS producers

As in the case of receiving messages, Spring provides conveniences for sending messages. The crucial abstraction for sending messages is the Spring `JmsTemplate` class. `JmsTemplate` follows the standard template pattern to provide a convenience class for sending messages.

One of the most common ways to send a message using Spring is by implementing the Spring `MessageCreator` interface and utilizing it with the appropriate `send()` method of the `JmsTemplate` class. Listing 2 demonstrates this by implementing all message creation logic using a stock portfolio publisher.

Listing 2 Implementation of a MessageCreator for sending messages using Spring

```
public class StockMessageCreator implements MessageCreator {

    private int MAX_DELTA_PERCENT = 1;
    private Map<Destination, Double> LAST_PRICES =
            new Hashtable<Destination, Double>();

    Destination stock;

    public StockMessageCreator(Destination stock) {
            this.stock = stock;
    }

    public Message createMessage(Session session) throws JMSException {
            Double value = LAST_PRICES.get(stock);
            if (value == null) {
                    value = new Double(Math.random() * 100);
            }

            // lets mutate the value by some percentage
            double oldPrice = value.doubleValue();
            value = new Double(mutatePrice(oldPrice));
            LAST_PRICES.put(stock, value);
            double price = value.doubleValue();

            double offer = price * 1.001;

            boolean up = (price > oldPrice);
            MapMessage message = session.createMapMessage();
            message.setString("stock", stock.toString());
            message.setDouble("price", price);
            message.setDouble("offer", offer);
            message.setBoolean("up", up);
            System.out.println("Sending: "
                + ((ActiveMQMapMessage)message).getContentMap()
                    + " on destination: " + stock);
            return message;
```

```
    }

    protected double mutatePrice(double price) {
            double percentChange = (2 * Math.random() * MAX_DELTA_PERCENT)
                            - MAX_DELTA_PERCENT;

            return price * (100 + percentChange) / 100;
    }

}
```

The `MessageCreator` interface defines only the `createMessage()` method, which returns a JMS message. Here, we have implemented some logic for creating random stock prices and creating an appropriate JMS map message to hold all of the relevant data. To send the message, the `JmsTemplate`'s `send()` method will utilize the `StockMessageCreator` as shown in listing 4.

Listing 4 JMS Publisher Implementation in Spring

```
public class SpringPublisher {

 private JmsTemplate template;
 private int count = 10;
 private int total;
 private Destination[] destinations;
 private HashMap<Destination,StockMessageCreator>
    creators = new HashMap<Destination,StockMessageCreator>();

 public void start() {
  while (total < 1000) {
   for (int i = 0; i < count; i++) {
    sendMessage();
   }
   total += count;
   System.out.println("Published '" + count + "' of '"
      + total + "' price messages");
   try {
   Thread.sleep(1000);
   } catch (InterruptedException x) {
   }
  }
 }

 protected void sendMessage() {
  int idx = 0;
   while (true) {
    idx = (int)Math.round(destinations.length * Math.random());
     if (idx < destinations.length) {
      break;
     }
   }
   Destination destination = destinations[idx];
   template.send(destination, getStockMessageCreator(destination));   #A
 }

 private StockMessageCreator getStockMessageCreator(Destination dest) {
  if (creators.containsKey(dest)) {
   return creators.get(dest);
  } else {
   StockMessageCreator creator = new StockMessageCreator(dest);
   creators.put(dest, creator);
   return creator;
  }
 }

 // getters and setters goes here
}
```

#A Send with JmsTemplate

The important thing to note in listing 4 is how the send() method uses the message creator. Now, you have all the necessary components to publish messages to ActiveMQ using Spring. All that is left to be done is to configure it properly, as demonstrated in listing 5.

Listing 5 JMS Publisher Configuration in Spring

```
<!-- Spring JMS Template -->
  <bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="pooledJmsConnectionFactory" />
  </bean>

  <bean id="stockPublisher"
class="org.apache.activemq.book.ch7.spring.SpringPublisher">
    <property name="template" ref="jmsTemplate" />
    <property name="destinations">
      <list>
        <ref local="cscoDest" />
        <ref local="orclDest" />
      </list>
    </property>
  </bean>
```

Listing 5 shows an instance of the Spring `JmsTemplate` and the publisher. The publisher simply needs a reference to the JMS destinations being used and the JmsTemplate requires a connection factory.

> **NOTE** The pooled connection factory is used with the `JmsTemplate`. This is very important because the `JmsTemplate` is designed for use with Java EE containers in mind, which typically provide connection pooling capabilities as required by the Java EE specifications. So, every call to the `JmsTemplate.send()` method creates and destroys all the JMS resources (connections, consumers, and producers). So, if you are not using a Java EE container, make sure to use a pooled connection factory for sending messages with `JmsTemplate`.

The connections and destinations are defined; the consumers and producer have been created; now, let's run the example.

Putting it all together

After implementing all of the pieces of the example, the application should be ready to run. Take a look at listing 6 to see the main method that will execute the example.

Listing 6 The main method for the Spring example

```
public class SpringClient {

 public static void main(String[] args) {
  BrokerService broker = new BrokerService();
  broker.addConnector("tcp://localhost:61616");
  broker.setPersistent(false);
  broker.start();                                    #A

  FileSystemXmlApplicationContext context =
    new FileSystemXmlApplicationContext(
      "src/main/resources/org/apache/activemq/book/ch7/spring-client.xml"
    );                                               #B
  SpringPublisher publisher =
    (SpringPublisher)context.getBean("stockPublisher");
  publisher.start();
 }
}
```

#A Start broker
#B Initialize Spring clients

This simple class just starts a minimal ActiveMQ broker configuration and initializes the Spring application context to start the JMS clients.

The example can be run from the command line using the following command:

```
$ mvn exec:java \
-Dexec.mainClass=org.apache.activemq.book.ch7.spring.SpringClient \ -
```

```
Dlog4j.configuration=file:src/main/java/log4j.properties
...
Sending: {price=65.958996694, stock=CSCO, offer=66.0249556914, up=false}
  on destination: topic://STOCKS.CSCO
topic://STOCKS.IONA 79.97 80.05 down
Sending: {price=80.67595675108, stock=ORCL, offer=80.7566327078, up=true}
  on destination: topic://STOCKS.ORCL
topic://STOCKS.JAVA 65.96 66.02 down
Sending: {price=65.63333898492, stock=CSCO, offer=65.69897232391, up=false}
  on destination: topic://STOCKS.CSCO
topic://STOCKS.IONA 80.68 80.76 up
Sending: {price=80.50525969261, stock=ORCL, offer=80.58576495231, up=false}
  on destination: topic://STOCKS.ORCL
topic://STOCKS.JAVA 65.63 65.70 down
Sending: {price=81.2186806051, stock=ORCL, offer=81.29989928577, up=true}
  on destination: topic://STOCKS.ORCL
topic://STOCKS.IONA 80.51 80.59 down
Sending: {price=65.48960846536, stock=CSCO, offer=65.5550980738, up=false}
  on destination: topic://CSCO
topic://STOCKS.IONA 81.22 81.30 up
topic://STOCKS.JAVA 65.49 65.56 down
```

As you can see, both producer and consumer print their messages to standard output as the example runs.

Summary

In this article, you used Spring to augment a stock portfolio application. You used some Spring utilities to simplify the example quite a lot. This example simply touched upon the basics of using Spring JMS.

Here are some other Manning titles you might be interested in:

Spring Integration in Action
Mark Fisher, Jonas Partner, Marius Bogoevici, and Iwein Fuld

Camel in Action
Claus Ibsen and Jonathan Anstey

Spring in Action, Third Edition
Craig Walls

Last updated: April 13, 2011