Message Driven POJOs

# Messaging Made Easy

by Mark Richards

*Spring provides a simple yet powerful messaging framework for receiving and sending messages. In this article I will demonstrate how to develop messaging-based applications using message-driven POJOs (MDPs) and describe some ways to use them in messaging-based applications. Given that no framework is entirely perfect, I will also discuss some of the limitations with MDPs and how to get around them.*

## *Introduction*

Before I begin describing the Spring messaging framework let me first say that although I very much like what the Spring Framework has to offer, particularly in terms of its messaging framework, I do not consider myself a "Spring Fan-Boy". On the contrary, with the possible exception of my MacBook Pro I pride myself on being largely technology and product agnostic. Using the right tool for the right job is an important ability that avoids the all-too-common "Golden Hammer" anti-pattern. When it comes to messaging and JMS, I find the Spring messaging framework is usually the right tool for the job.

The JMS API (used natively without Spring) is very straightforward and relatively simple to use. Assuming you are using queues with the point-to-point messaging model, you first obtain a QueueConnectionFactory and a Queue destination from JNDI. You then create a QueueConnection from the QueueConnectionFactory, a QueueSession from the QueueConnection, a QueueSender from the QueueSession, and a JMS Message object (e.g., TextMessage) from the QueueSession. Finally, you create your message content and send it to the queue using the send( ) method on the QueueSender. These steps are illustrated in *RIC-1, Sending a message using the JMS API* below.

```
public void sendMessage( ) throws Exception {
   Context ctx = new InitialContext( );
   QueueConnectionFactory factory = (QueueConnectionFactory)ctx.lookup("QueueCF");
   Queue queue = (Queue)ctx.lookup("queue");
   QueueConnection connection = factory.createQueueConnection();
   QueueSession session = connection.createQueueSession
        (false, Session.AUTO_ACKNOWLEDGE);
   connection.start();

   StringBuffer xml = new StringBuffer("\n");
   xml.append("<trade>"                  + "\n");
   xml.append("  <side>BUY</side>"       + "\n");
   xml.append("  <symbol>AAPL</symbol>" + "\n");
   xml.append("  <shares>100</shares>"   + "\n");
   xml.append("</trade>"                 + "\n");

   QueueSender sender = session.createSender(queue);
   TextMessage msg = session.createTextMessage(xml.toString());
   sender.send(msg);
   connection.close( );
}
```



**Listing RIC-1**

### Listing RIC-2: Sending a message using the Spring messaging framework

Although the JMS API is straightforward and simple, it is rather verbose. This is where the Spring messaging framework come in handy. From a Java coding standpoint, accomplishing the same thing as shown in Listing RIC-1 using the Spring messaging framework is simply a matter of invoking one of the send methods on the Spring JmsTemplate object. Listing RIC-2 illustrates the use of the Spring messaging framework for sending a message as a Java Object:

```
public void sendMessage( ) throws
Exception {

   StringBuffer xml = new
StringBuffer("\n");

   xml.append("<trade>"
+ "\n");

   xml.append("  <side>BUY</side>"
+ "\n");

   xml.append("  <symbol>AAPL</
symbol>" + "\n");

   xml.append("  <shares>100</
shares>"  + "\n");

   xml.append("</trade>"
+ "\n");


   jmsTemplate.convertAndSend(xml.
toString( ));

}
```

***Listing RIC-2***

### Listing RIC-3: Sending a JMS message object using the Spring messaging framework

The Spring messaging framework takes care of establishing a JMS connection, performing JNDI lookups for queues and topics, managing the JMS session, and all the cleanup associated with the JMS. This allows the developer to focus on application business logic rather than JMS code.

Of course, it isn't all that simple. There is a fair amount of configuration associated with using the Spring messaging framework. In contrast with most articles and blogs I have read regarding the Spring messaging framework, I really don't mind the necessary (and verbose) configuration required for the Spring messaging framework. I rather like having the connection information, JMS destinations, and tuning parameters in configuration rather than code. In addition, Spring 2.5 offers some significant improvements in the configuration for messaging, particularly for Message-Driven POJOs.

Listing RIC-2 above also shows another great feature of the Spring messaging framework: the ability to send and receive Objects rather than JMS Messages. Spring will automatically convert a TextMessage into a String object, an ObjectMessage into an Object, a BytesMessage into a byte[ ], and finally a MapMessage into a java.util.Map (and visa-versa). Of course, Spring will allow you to send a regular JMS Message just as easily, as shown in RIC-3 below:

```
public void sendMessage( ) throws
Exception {

   final StringBuffer xml = new
StringBuffer("\n");

   xml.append("<trade>"
+ "\n");

   xml.append("  <side>BUY</side>"
+ "\n");

   xml.append("  <symbol>AAPL</
symbol>" + "\n");

   xml.append("  <shares>100</
shares>"  + "\n");

   xml.append("</trade>"
+ "\n");


   jmsTemplate.send(new
MessageCreator( ) {

      public Message
createMessage(Session session)
throws JMSException {

         TextMessage msg = session.
createTextMessage(xml.toString( ));

         return msg;

      }

   });

}
```

***Listing RIC-3***

Notice in Listing RIC-3 above I had to make the StringBuffer final so that it could be used in the anonymous MessageCreator class needed to build the actual JMS message object (in this case a TextMessage).

Another thing to note about the Spring messaging framework is that it is not a JMS Provider. Thus, even when using the Spring messaging framework you will still need to connect to an external JMS provider such as ActiveMQ, WebSphereMQ, SonicMQ, Tibco, or any Java EE 1.4 and above compliant application server.

## *Message-Driven POJOs*

Spring offers two ways to process JMS messages: the JmsTemplate for synchronously sending and receiving messages, and Message-Driven POJOs (MDP) for receiving messages asynchronously. In the introduction I showed a couple of coding examples using the JmsTemplate to illustrate the simplicity of the Spring messaging framework. However, what I want to focus on in this article is the more powerful Message-Driven POJOs available in Spring.

Message-Driven POJOs (or MDPs) are asynchronous message listeners that are managed by the Spring framework. They are similar to the Message-Driven Beans provided by the Enterprise JavaBean (EJB) specification, only much more powerful. MDPs also support the request/reply messaging model and event-driven processing, allowing you to send messages from an MDP using the JMS API or the Spring JmsTemplate.

There are three types of MDPs: Those that implement the javax.jms.MessageListener interface (similar to MDBs in EJB), those that implement the Spring Framework SessionAwareMessageListener interface, and finally those that use the Spring MessageListenerAdapter. Since the latter type are far more interesting and useful than the other types, I will be focusing this article on MDPs that use the MessageListenerAdapter.

### Configuration Basics

Message-Driven POJOs are just that – Plain Old Java Objects. They are not required to implement the javax.jms.MessageListener interface (although that is one of the options). From a developer standpoint they appear to be regular objects. The way they become MDPs is by wrapping the POJO class in a MessageListenerAdapter class provided by Spring. The MessageListenerAdapter is used to specify some of the behavioral aspects of the MDP, such as whether Spring should convert the JMS message object to a corresponding Java Object type, and what method name Spring should invoke on the MDP when receiving a message.

The MessageListenerAdapter is added to a MessageListenerContainer, which further specifies how the MDP should be configured. The MessageListenerContainer specifies many additional configuration options, including the JMS ConnectionFactory Spring should use, the Destination Resolver that Spring should use for resolving JNDI objects via lookups, and the number of instances of the message listener class Spring should create. The queue or topic the MDP is listening on is also specified when adding the MDP to the message listener container.

Version 2.5 of the Spring Framework introduced the new JMS Namespace as an easier and less verbose way of configuring MDPs. Prior to version 2.5, each message listener bean defined in Spring required its own message listener container. This is no longer true with version 2.5. You can now have as many message listener beans as you want in a single message listener container.

### Receiving JMS Message Objects using Default Handler Methods

The most basic MessageListenerAdapter MDP is one that receives a JMS message object (e.g., TextMessage) using the default handler method (handleMessage( )). This is similar to how Message-Driven Beans work in the EJB specification. I will start with the configuration and then show the java source code. To illustrate how MDPs work I will create a MDP called TradingMDP that receives a JMS TextMessage consisting of XML payload containing a stock trade order (similar to the one sent in Listing RIC-2.

The configuration for this type of MDP is fairly straightforward. First, the message listener class containing the application logic (in this case TradingMDP) must be wrapped in a MessageListenerAdapter class provided by Spring. This can be done by passing the class name of the MDP in the constructor-arg of the MessageListenerAdapter bean. Specifying a null value for the messageConverter property tells Spring to use the JMS Message objects rather than try to convert the message object into a Java object:

```
<bean id="messageListener1"
      class="org.springframework.jms.listener.
adapter.MessageListenerAdapter">
      <constructor-arg>
          <bean class="com.trading.TradingMDP"/>
      </constructor-arg>
      <property name="messageConverter"><nu
ll/></property>
</bean>
```

***Listing RIC-4***

***MDP Message Listener Adapter without message conversion***

### Listing RIC-5: MDP Message Listener Container configuration

Next, The message listener bean (in this case *messageListener1*) is added to a message listener container. The message listener container tells Spring which queue or topic the MDP will be listening on, how to connect to the JMS provider, and how many receivers to create. JMS supports multiple message listeners (called concurrent consumers) for a single queue. For example, if you have 5 message listeners (as indicated in the *concurrency* property show in the code below), this means you can process 5 messages from the queue at the same time. The configuration using the JMS Namespace is illustrated below:

```
<jms:listener-container connection-
factory="queueConnectionFactory"

destination-
resolver="destinationResolver"

concurrency="5">
     <jms:listener
destination="queue1"
ref="messageListener1"/>
</jms:listener-container>
```

**Listing RIC-5**

### Listing RIC-6: MDP without message conversion using the default listener method

Now it is time to code the TradingMDP class. By default Spring will look for a method called handleMessage with an argument matching the JMS message object (in this case TextMessage). The code for the TradingMDP is shown below:

```
package com.trading;
import javax.jms.TextMessage;
public class TradingMDP {
   public void
handleMessage(TextMessage msg) {
      try {
         String xml = msg.getText( );
         //process xml trade order...
      } catch (Exception e) {
         e.printStackTrace( );
      }
 }
```

**Listing RIC-6**

Notice in Listing RIC-6 the use of the method signature handleMessage(TextMessage msg). When a message is received in the queue, Spring will invoke a method with that signature, passing the message along to the method. Notice that the message object is already cast to its correct type. If there were two message objects in the queue, say TextMessage and StreamMessage, then you would have to define two methods in the MDP: one for the TextMessage and one for the StreamMessage.

This MDP is good, but not good enough. The thing I don't like about it is the fact that it still references the JMS API. To solve this problem you can leverage the message conversion features of Spring messaging, which are described in the next section.

### Receiving Java Objects using Default Handler Methods

The above MDP is somewhat annoying because the class still references the JMS API. Wouldn't it be nice if you could avoid having any sort of messaging code in the POJO and still be able to receive and process messages? With MDPs you can.

Like the JmsTemplate example in Listing RIC-2, you can have Spring convert the JMS message object into a corresponding Java object instead. Spring comes with a default message converter called a SimpleMessageConverter, which will convert an incoming TextMessage to a String object, an ObjectMessage to a Java Object, a BytesMessage to a byte[], and a MapMessage into a java.util.Map object. By converting the JMS message object into a corresponding Java Object in the MDP, you can remove a majority (if not all) of the messaging logic in your code.

The trick to enabling the automatic message conversion is to simply remove the null value in the messageConverter property of the MessageListenerAdapter bean, or more specifically don't set the property at all. When you don't specify a value for the messageConverter property, Spring will by default use the SimpleMessageConverter object to convert the incoming JMS message object into its corresponding Java Object type:

```
<bean id="messageListener1"
      class="org.springframework.jms.listener.
adapter.MessageListenerAdapter">
      <constructor-arg>
          <bean class="com.trading.
TradingMDP"/>
      </constructor-arg>
</bean>
```

**Listing RIC-7**

**MDP Message Listener Adapter with message conversion**

Listing RIC-8 MDP with message conversion using the default listener method

With message conversion, rather than looking for a method signature matching the JMS message object, Spring will look for a method signature matching the corresponding Java Object. Thus, to process a TextMessage from the queue you would create a method signature containing handleMessage(String s) as shown in Listing RIC-8 below:

```
package com.trading;
public class TradingMDP {
   public void handleMessage(String
xml) {
      try {
         //process xml trade order...
      } catch (Exception e) {
         e.printStackTrace( );
      }
   }
}
```

***Listing RIC-8***

There are a couple of things to notice in the code shown in Listing RIC-8. First of all, since you are passing in a String object containing the message payload (in this case XML containing the stock trade order), you no longer need to extract the message payload using the getText( ) method as you did in Listing RIC-6. The other thing to notice is that you no longer need to import javax.jms.TextMessage. As a matter of fact, if you look closely at the code, you will see that there is no sign of JMS or messaging whatsoever.

This code is much better than the previous example in that it no longer contains a reference to the JMS API. This allows you to focus solely on the business logic required to process the message. However, while this code is better than the prior example, it can still be improved. The thing that still annoys me about the code in Listing RIC-8 is that it requires the use of a method named handleMessage( ) (not very descriptive in my opinion). This can be fixed quite easily, as is shown in the next section.

## *Receiving Java Objects using Custom Handler Methods*

The code in Listing RIC-8 is a big improvement over the code shown in Listing RIC-6. You no longer need to code any JMS logic, allowing you to focus on application business logic. However, the code in Listing RIC-8 is still not perfect in that it is still using the default message handler name (handleMessage( )). The handleMessage( ) method is only the default handler method name for the message listener adapter. You can set the method name to anything you want. Since the method in Listing RIC-8 is processing a trade order, perhaps a more descriptive name for the method would be processTrade( ).

To have Spring invoke a custom method name rather than the default handleMessage( ) name when a message is received, simply set the value of the defaultListenerMethod property on the MessageListenerAdapter to the name of the method in your MDP (in this case processTrade):

```
<bean id="messageListener1"
      class="org.springframework.
jms.listener.adapter.
MessageListenerAdapter">
      <constructor-arg>
          <bean class="com.trading.
TradingMDP"/>
      </constructor-arg>
      <property
name="defaultListenerMethod"
value="processTrade"/>
</bean>
```

***Listing RIC-9***

***Message Listener Adapter using custom handler method***

Listing RIC-10 MDP with message conversion using a custom handler method

Now, your POJO does not have to define the generic handleMessage( ) method, but rather a more descriptive one as shown in Listing RIC-10 (next page):

```
package com.trading;v
public class TradingMDP {
   public void processTrade(String
xml) {
      try {
         //process xml trade order...
      } catch (Exception e) {
         e.printStackTrace( );
      }
   }
}
```

***Listing RIC-10***

Now this is what I call a MDP! Take a close look at the code in Listing RIC-10. This code is actually an asynchronous message listener that receives a trade order in XML format from a message queue. However, there is no reference to the JMS API, and as a matter of fact no reference to messaging or Spring whatsoever. This illustrates the true power of MDPs. The code in Listing RIC-10, although written as an asynchronous message listener, can be used (and tested) outside of the context of messaging and JMS. This means that the same exact source code can be used in the context of a messaging infrastructure as well as other input channels (such as Web Services, Web-based requests, or even VIOP).

## MDP Limitations

Looking at the source code in Listing RIC-10, you might be convinced that MDPs are the greatest thing since sliced bread. However, there are a few limitations and considerations to take into account before converting all of your code to MDPs with message conversion and custom handler methods.

The first consideration to take into account when using MDPs with message conversion and custom handler methods (as shown in Listing RIC-10) is that without proper documentation, you can't tell that the code is actually an asynchronous message listener. A developer, not knowing this fact, may alter the source code making it incompatible with the messaging framework. Although the code doesn't show it, the class is still an asynchronous message listener and for the most part should be treated as such.

The second limitation with MDPs will become immediately apparent with the following scenario. Using the code in Listing RIC-10, assume that, along with the XML trade order, you also need to pass some additional information in the message properties section of the message header (such as security credentials) as shown in Listing RIC-11:

```
public void sendMessage( ) throws Exception {
   StringBuffer xml = new StringBuffer("\n");
   xml.append("<trade>"              + "\n");
   xml.append("  <side>BUY</side>"     + "\n");
   xml.append("  <symbol>AAPL</symbol>" + "\n");
   xml.append("  <shares>100</shares>"  + "\n");
   xml.append("</trade>"             + "\n");

   jmsTemplate.send(new MessageCreator( ) {
      public Message createMessage(Session session)
throws JMSException {
         TextMessage msg = session.
createTextMessage(xml.toString( ));
         msg.setLongProperty("traderId", 99);
         return msg;
      }
   });
}
```

***Listing RIC-11***

***Sending additional header info.***

Listing 12: MDP without message conversion using a custom handler method

Now assume for auditing purposes you need to extract the traderId property from the message header and write a record out to the audit table when receiving the trade order message in the MDP. Oops – looking at the code in Listing RIC-10, this isn't possible with a MDP with message conversion because only the

message payload is passed to the method (not the header information). The MDP does not have access to the original message and cannot access the message header. This is a serious limitation in that you cannot perform request/reply processing (you would need access to the JMSReplyTo header property) and cannot process additional opaque information contained in the message properties area of the header.

Of course, this can easily be fixed by moving back to the MDP version that processes a JMS message object. You simply need to set the messageConverter property on the MessageListenerAdapter to null (as shown in Listing 4) and change the method signature in the source code, which is shown in Listing RIC-12 below:

```
package com.trading;
import javax.jms.TextMessage;

public class TradingMDP {
   public void processTrade(TextMessage msg) {
      try {
         String xml = msg.getText( );
         //process xml trade order...
      } catch (Exception e) {
         e.printStackTrace( );
      }
   }
}
```

**Listing RIC-12**

## Conclusion

*In this article I introduced the JmsTemplate for sending messages and described the stages of a simple MDP from using the JMS-specific handleMessage(TextMessage msg) method to the message-agnostic processTrade(String xml) method used in the stock trade example. The Spring messaging framework significantly reduces the amount of code needed to send and receive messages, allowing you to focus on the application business logic rather than the JMS infrastructure. Although there are some limitations and considerations to take into account when using MDPs, they are nevertheless a powerful tool for processing JMS messages.*

## About The Author

*Mark Richards is a Director and Senior Technical Architect at Collaborative Consulting, LLC (http://www.collaborative.com). He is the author of the 2nd edition of Java Message Service (O'Reilly, 2009) and of Java Transaction Design Strategies (C4Media Publishing, 2006). He is also a contributing author of several other books, including 97 Things Every Software Architect Should Know (O'Reilly, 2009), NFJS Anthology Volume 1 (Pragmatic Bookshelf, 2006) and NFJS Anthology Volume 2 (Pragmatic Bookshelf, 2007). Mark is a regular speaker for the No Fluff Just Stuff Symposium Series and speaks at other conferences and user groups throughout the world.*