The JMS Control Developer's Guide

Table of contents

1. Overview: Messaging Systems and JMS

A JMS control makes it easy for your application to communicate with messaging systems. To better understand how to use a JMS control, it helps to understand messaging systems and how JMS control interact with them.

1.1. Understanding Messaging Systems

Messaging systems provide communication between software components. A client of a messaging system can send messages to, and receive messages from, any other client. Each client connects to a messaging server that provides facilities for sending and receiving messages. Codehaus's ActiveMQ which is a component of the Apache Geronimo project, is an example of a messaging server.

Messaging systems provide distributed communication that is asynchronous. This means that a component sends a message to a destination and a message recipient can retrieve messages from a destination, but the sender and receiver do not communicate directly. The sender only knows that a destination exists to which it can send messages, and the receiver also knows there is a destination from which it can receive messages. As long as they agree what message format and what destination to use, the messaging system manages the actual message delivery.

Messaging systems also provide reliability for message delivery. The specific level of reliability is typically configurable on a per-destination or per-client basis, but messaging systems are capable of guaranteeing that a message will be delivered, and that it will be delivered to each intended recipient exactly once.

JMS supports two basic styles of message-based communications: point-to-point and publish-and-subscribe. Each is described in greater detail below.

1.2. Using JMS Queues for Point-to-Point Messaging

Point-to-point messaging is accomplished with JMS queues, which are specific named resources configured in a JMS server. A JMS client, of which the JMS control is an example, sends messages to a queue or receives messages from a queue.

Point-to-point messages have a single consumer. Multiple receivers can listen for messages on the same queue, but once any receiver retrieves a particular message from the queue that message is consumed and is no longer available to other potential consumers.

The messaging system continues to resend a particular message until a predetermined number of retries have been attempted. Once the message is received, a message consumer acknowledges receipt.

1.3. Using JMS Topics for Publish-and-Subscribe Messaging

Publish-and-subscribe messaging is accomplished with JMS topics. A topic is a specific named resource configured in a JMS server.

A JMS client, of which the JMS control is an example, publishes messages to a topic, or subscribes to a topic. Published messages have multiple potential subscribers. All current subscribers to a topic receive all messages published to that topic after the subscription becomes active.

1.4. Connection Factories and Transactions

Before a JMS client can send or receive messages to a queue or topic, it must obtain a connection to the messaging system, via a connection factory. A connection factory is a resource that is configured by the message server administrator. The names of connection factories are stored in a JNDI directory, where clients wishing to make a connection can look them up.

Unless otherwise specified the default initial context is used. This may be overridden by settng the `jndiContextFactory` and `jndiProviderUrl` properties, either programically using the `setJndiContextFactory()` and `setJndiProviderUrl()` setters or via the corresponding `@Destination` attributes.

2. JMS Control Annotations

2.1. JMS Control Class-level Annotations

The `JMSControl.Destination` annotation defines the destination of the message, the message type and connection related attributes. The attributes defined for this annotation are:

Attribute	Value	Required	Description
sendJndiName	String	Yes	JNDI name of the queue or topic.
sendCorrelationProperty	String	No	The correlation property to be used for message sent. Default is empty, which signifies that the JMS correlation header is to be used.

connectionFactoryJndiN	String	Yes	JNDI name of the connection factory. Required
transacted	boolean	No	True if en-queuing is under transactional semantics of the enclosing container. Default is true.
acknowledgeMode	enum AcknowledgeMode	No	The acknowledgement strategy, one of Auto, Client, DupsOk. Default is Auto.
sendType	JMSControl.Destination	No	Values are Auto, Queue and Topic. If Auto, then the type is determined by the destination named by the sendJndiName attribute. Default is Auto.
jndiContextFactory	String	No	The class name of the jndi-context-factory. Default is none.
jndiProviderURL	String	No	The provider URL for JNDI. Default is none.

2.2. JMS Control Method Annotations

Methods added to a JMS control that send messages may be annotated with the following annotations:

Annotation	Value	Description
JMSControl.Message	JMSControl.MessageType (enum)	Enum values are: Auto, Text, Bytes, Object, Map and JMSMessage
JMSControl.Priority	int	A JMS priority (0-9). Defaults to provider's default priority.
JMSControl.Expiration	long	A JMS expiration in milliseconds. Default's to provider's default expiration.

JMSControl.Delivery	JMSControl.DeliveryMode (enum)	This attribute determines the delivery mode of the message. Defaults to the JMS provider's default delivery mode. Enum values are: NonPersistent, Persistent and Auto
JMSControl.Type	String	Specifies the JMS type.
JMSControl.CorrelationId	String	Specifies the JMS correlation id.
JMSControl.Properties	JMSControl.PropertyValue[]	One or more string/int/long valued properties to be added to the message. `PropertyValue` has the string valued attributes 'name', 'value' and class valued 'type'. The allowed values for 'type' are String.class, Integer.class and Long.class. If type is not specified, then String is assumed.

Notes for the JWSControl.MessageType enumerated value:

- If not specified or no message-type string, then the default is Auto.
- If Auto, then the type of JMS message is determined by the type of the body passed in; rules for determining these types are:
  - If the body is a String or XmlObject, a TextMessage is sent.
  - If the body is a byte[], a StreamMessage is sent.
  - If the body is a Map, a MapMessage is sent
  - If the body is a JMSMessage, a JMSMessage is sent
  - Otherwise if the body is Serializable, an ObjectMessage is sent.
  - Any other type results in a control exception.

2.3. JMS Control Method Parameter Annotations

These annotations denote which parameter is to be the body of the message and zero or more properties to be set in the message respectively. The following annotations my be used on method parameters:

Annotation	Value	Description
JMSControl.Property	String	The parameter contains the value of the property.

JMSControl.Priority	int	A JMS priority (0-9). If not specifed the method-level annotation is used; if method-level annotation has not been specified the default for the JMS provider is used.
JMSControl.Expiration	long	JMS expiration in milliseconds. If not specified the method-level annotation is used; if method-level annotation has not been specified the default for the provider is used.
JMSControl.Delivery	JMSControl.DeliveryMode	The DeliveryMode valued parameter determines the delivery mode of the message. If not specified, then the method-level annotation is used; else the default for the provider is used.
JMSControl.Type	String	The JMS type.
JMSControl.CorrelationId	String	The JMS correlation id.

3. JMS Control Methods

A JMS control always includes the following methods:

Method	Description
getSession()	Get the queue/topic session.
getDestination()	Get the queue/topic destination.
getConnection()	Get the queue/topic connection.
setHeaders(Map)	Sets the JMS headers to be assigned to the next JMS message sent. Note that these headers are set only on the next message, subsequent messages will not get these headers. Also note that if the body is a message itself, then any header set through this map will override headers set in the message. The keys should be of type HeaderType or equivalent strings. See table below for valid values.

setHeader(HeaderType,Object)	Sets a JMS header to be assigned to the next JMS message sent. Note that this header is set only on the next message, subsequent messages will not get this header. Also note that if the body is a message itself, then the header set here will override the header set in the message.
setProperties(Map)	Sets the JMS properties to be assigned to the next JMS message sent. Note that these properties are set only on the next message, subsequent messages will not get these properties. Also note that if the next message is sent through a publish method, then any property set through this map will override properties set in the message itself.
setProperty(String,Object)	Set the given JMS property to be assigned to the next JMS message sent. Note that this property is set only on the next message, subsequent messages will not get this property. Also note that if the body is a message itself, then the property set here will override the property set in the message.

The methods of the extension control-classes correspond to sending a message to a topic/queue, e.g.

```
send<message-type>(...)
```

4. Header Types

The table below defines the valid values for header types passed into setHeader() or setHeaders():

JMS Message Method	HeaderType/String	Allowed Value Types
setJMSType()	JMSType	String
setJMSCorrelationID()	JMSCorrelationID	String or byte[]
setJMSExpiration()	JMSExpiration	String valued long or Long
setJMSPriority()	Priority	String valued int or Integer

5. Creating a JMS Control

The JMS control is an extensible control. Before a JMS Control can be used in an application, a sub-interface of the `org.apache.beehive.controls.system.jms.JmsControl` interface must be created and annotated with `@ControlExtension`.

```
@ControlExtension
public interface SampleQueue
    extends JMSControl {
...
}
```

A JMS control needs to know the destination of the messages it will send. This is accomplished using a JNDI context. Unless otherwise specified the default initial context is used. This may be overridden by settng the `@Destination` annotation's `jndiContextFactory` and the `jndiProviderUrl` attributes.

The queue/topic destination is then obtained using the value of the `sendJndiName` attribute of the `@Destination` annotation. A queue/topic connection is obtained using by the `jndiConnectionFactory` attribute. In most cases the same connection factory is used for both queues and topics.

The `@Destination.sendType` attribute may be used to constrain the use of the control to either a topic or a queue. By default its value is `Auto` which allows for run-time determination of whether the `sendJndiName` names a queue or a topic. By setting it to Queue or Topic a run-time check is made to see if the connection factory and destination is of the correct type.

The extension interface can include one or more methods that send messages. These methods must have at least one parameter that corresponds to the body of the message. Other annotated parameters can defined to provide property values and other information at run-time to the message. The method itself can be annotated as well.

In the example below, the OrderQueue control class has one submitOrder() method that takes an Order object as the body and a string that sets the 'DeliverBy' property in the javax.jms.ObjectMessage to be sent to the queue.orders JMS queue.

```
@ControlExtension
@JMSControl.Destination(sendJndiName="queue.orders",jndiConnectionFactory="weblogic.jws
public interface OrderQueue extends JMSControl
 {
    public class Order implements java.io.Serializable
    {
        public Order()
        {

        }
        public Order(int buyer,String[] list)
        {
            buyerId = buyer;
            itemList = list;

        }
        private int buyerId;
        private String[] itemList;
    }

    public void submitOrder(Order order,@Property(name="DeliverBy") String
deliverBy);
}
```

6. Specifying the Message Body

This section describes some of the ways in which you can specify the body of a message sent via the JMS control.

6.1. Selecting the Message Type

A JMS control can send text messages (including XML messages), byte array messages, object messages, and javax.jms.Message (JMS Message) objects. These are the types defined by the JMS messaging service specification.

When you create a JMS control, you can specify which type of message it sends and receives with the `JMSControl.Message.messageType()` annotation.

You have complete control over the send methods, as long as you are sending a message of a supported type; you can modify method signatures as you need to, including adding additional parameters to handle message headers and properties. However, you can only specify one parameter in the method for the message body.

6.2. Sending and Receiving a Simple Text Message

The simplest message body is a text message. The following example shows a simple text message sent to the messaging service via a JMS control:

```
public void sendString(String msg) throws Exception
  {
  myJMSControl.sendTextMessage(msg);
  }
```

6.3. Sending and Receiving an XML Message using XMLBeans

If you need to send a set of values in the message body, you can construct the message body using an XMLBeans object type. Apache XMLBeans technology generates a set of Java classes from an XML schema (.xsd) file. You can then use these classes to work with XML documents in your code.

If you don't already have a schema file, you can construct one by hand, or you can generate one from an XML document or fragment using a third-party authoring tool. Once you've generated the XMLBeans classes from the schema file, you can import those classes into your JMS control class. You can then modify the send method or receiving callback on the JMS control to send or receive a message of the appropriate type.

Note that XMLBeans messages are transmitted as JMS text messages. When you create a JMS control that will use an XMLBeans type for the message body, specify the type as with the `JMSControl.Message.messageType()` annotation as 'Text'.

The following is a simple JWS Control which sends an XML message:

```
import java.util.Date;

import org.apache.beehive.controls.api.bean.ControlExtension;
import org.apache.beehive.controls.system.jms.JMSControl;
import org.apache.xmlbeans.XmlObject;

@ControlExtension
@JMSControl.Destination(sendJndiName="jms.SampleQueue",jndiConnectionFactory="weblogic.
public interface SampleQueue extends JMSControl
{
   /**
    * Submit an xml object (org.apache.xmlbeans) as a text message.
    * @param document the document.
    * @param type the message JMS type.
    */
    public void submitXml(XmlObject document,@Type String type);

    /**
     * Submit an xml object (org.apache.xmlbeans) with JMS type
"xmlObject".
     * @param document the document.
     */
    @Message(MessageType.Text)
    @Type("xmlObject")
    public void submitXml(XmlObject document);

    /**
     * Submit an already constructed message
     * @param message the jms-message.
     */
    public void submitMessage(Message message);
}
```

7. Specifying Message Headers and Properties

The JMS control includes properties for setting and retrieving headers and properties on a JMS message.

7.1. Accessing Message Headers

A JMS message includes a number of header fields that contain information used to identify and route messages. You can set the message headers for an outgoing message using the JMS control by using the `JMSControl.setHeaders(Map)` method or the `JMSControl.setHeader(HeaderType, Object)` method. The supported headers for an outgoing message are:

- JMSCorrelationID
- JMSExpriation
- Priority
- JMSType

For more information on these headers, see the Sun JMS specification.

7.2. Accessing Message Properties

A JMS message can also include properties that you or the message sender can add to send additional information about the message. You can think of them as optional, custom headers. Properties can be of type boolean, byte, short, int, long, float, double, or string. They can be set when a message is sent. You can add as many properties to the message as you need to.

You can set the properties of messages sent using the JMS control by using the `JMSControl.setProperties(Map)` or the `JMSControl.setProperty(String, Object)` methods.

When the JMS control is sending a message, the JMS control adds the properties specified to the outgoing message. You can optionally specify that a parameter passed to the method that sends the message should be substituted as a property value on the message.