_JMS

This topic explains how to use the JMS transport with SOAtest, as well as supporting tools and provisioning action tools in Virtualize.

It includes the following sections:

JMS Prerequisites

If you are using SOAtest or Virtualize tools as a JMS client, we recommend that SOAtest or Virtualize consult a JNDI provider to make connections to the JMS middleware. For this to happen, a JNDI Provider needs to be set up and all necessary jar files (i.e. ones containing the Initial Context) need to be added to the SOAtest or Virtualize classpath. (For more information on how to add jars to the classpath, see System Properties Settings). You will also need to supply the names of the connection factory, destination, and reply-to queue that the JNDI provider will look up.

If your JMS setup does not have a JNDI provider that SOAtest or Virtualize can query for an instance of a ConnectionFactory, follow the instructions in JMS Messaging without JNDI. Alternatively, you can set up a simple file system JNDI provider. The .jars and documentation for such a provider are available from the Oracle Java site. Setting up a file system provider is quite easy and the documentation is included in the download. Once the provider is ready, use the piece of simple Java code as described in Configuration for Popular JMS Providers to create an instance of the ConnectionFactory to connect to the JMS server using the host and port as arguments. The same should be done for Topics and Queues used by this product.

Configuring JMS Options

After selecting **JMS** from the Transport drop-down menu within the **Transport** tab of an appropriate tool, the following options display in the left pane:

Connection Settings

Connection Settings contains **Settings** and **Properties** tabs for **JNDI Initial Context**.

The **Properties** tab is optional and allows you to specify additional properties to be passed to the JNDI javax.naming.InitalContext constructor (in addition to the Provider URL and Initial Context factory properties that are specified in the Settings tab). Property values, which can be added by clicking **Add** and completing the Add JMS Property dialog, can be set to a fixed value, a parameterized value, a scripted value, or a unique value (an automatically-generated random unique value—no two tool invocations will use the same value).

The **Settings** tab contains the following:

- If you created a Shared Property for JMS Connections, a drop-down menu will be available from which you can choose **Use Local Settings** or **Use Shared Property**.
    - If you select **Use Shared Property**, a second drop-down menu displays from which you select the desired global JMS settings that the tool will use. For more information on global JMS settings, see Adding Global Test Suite Properties.
    - If you select **Use Local Settings**, or if no shared property is specified, you can configure the rest of the options for Connection Settings.
- **Provider URL:** Specifies the value of the property named javax.naming.Context.PROVIDER_URL passed to the JNDI javax.naming.InitialContext constructor.
- **Initial Context:** Specifies a fully qualified class name string, passed to the JNDI javax.naming.InitialContext constructor as a string value for the property named javax.naming.Context.INITIAL_CONTEXT_FACTORY.
- **Connection Factory:** Specifies the JNDI name for the factory This is passed to the lookup() method in javax.naming.InitialContext to create a javax.jms.QueueConnectionFactory or a javax.jms.TopicConnectionFactory instance.

In addition to the Settings tab, the Connection Settings also include:

- **Queue Connection Authentication:** Allows users to provide a username and password to create a queue connection. Select the **Perform Authentication** check box and enter the **Username** and **Password** to authenticate the request. If the correct username and password are not used, the request will not be authenticated.The username and password provided here is passed to the createQueueConnection() method in the javax.jms.QueueConnectionFactory class in order to get an instance of javax.jms.QueueConnection.

- **Keep-Alive Connection:** Select to notify the test whether to share or close the current connection. The shared connections are returned to the connection pool to be used across the test suite. A life cycle of a connection pool is as follows:
    - For a single test, it is destroyed at the end of the test execution.
    - For a test suite, it is destroyed at the end of the test suite execution.
    - For a load test, it is destroyed at the end of the load testing.

Queue/Topic

The Queue/Topic settings contain the following options:

- **JMS Destination:** Specifies the queue name (if point to point is used) or topic name (if publish and subscribe is used) for where the message will be sent to.
- **JMS ReplyTo:** Specifies the queue name (if point to point is used) or topic name (if publish and subscribe is used) for where to get a response message from. This can be a temporary queue if **Temporary** is selected instead of **Form**.

Messaging Model

Messaging Model options specify how messages are sent between applications. Select either **Point to Point** or **Publish and Subscribe**.

Messaging Expiration

The Messaging Expiration option specifies the message expiration time (in milliseconds).

Message Exchange Pattern

Message Exchange Pattern options specify whether or not SOAtest or Virtualize receives a response. If **Get Response** is selected, SOAtest or Virtualize sends a message and receives a response. If **Get Response** is not selected, SOAtest or Virtualize sends a one-way message and does not receive a response.

If **Get Response** is selected, you can also enable **Create message consumer on the JMSReplyTo destination before sending the message**. If the response is expected to become available very quickly on the JMSReplyTo topic, this option should be enabled to ensure that SOAtest or Virtualize has subscribed to the reply topic before the response message is published.

This option is cannot be mixed with **Match response JMSCorrelationID with the request JMSMEssageID** because the JMS specification requires vendors to generate the JMSMessageID after the message is sent. As a result, there is no way to create the consumer on the response destination with that correlation (selector) set until after the message has been set and the JMSMessageID becomes available.

Message Type

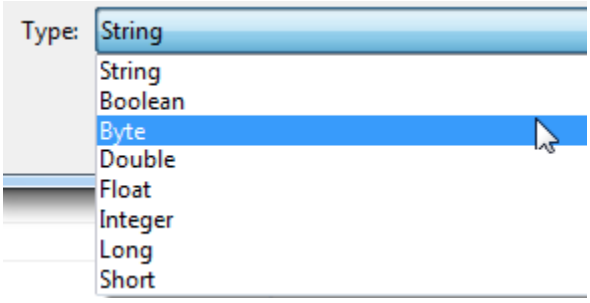
Message Type options allow you to select the message type from the drop-down menu. A JMS Message is a Java object that contains the data being transferred between JMS clients. The following Message Types are available:

- **TextMessage:** Used to send a message containing a java.lang.String. It is useful if you want only to send a simple text document. If you are doing SOAP over JMS, the String you enter into the TextMessage should be the SOAP Envelope.
- **BytesMessage:** Used to send a message containing a stream of uninterpreted bytes. The receiver of the message interprets the bytes as it sees fit. If you are doing SOAP over JMS, the bytes you enter into the ByteMessage should compose the SOAP Envelope. Data is sent in its most basic representation. It may also be useful when the JMS node is only interested in forwarding/routing data so the contents of the data aren't important to them. It is one of the most commonly used along with TextMessage.
    - When the encoding in the Parasoft preferences is UTF-8 (the default), you can use the **Method** options to control how a string is extracted from response BytesMessages. The two methods in the drop-down menu correspond to the two methods available in BytesMessage JMS API (see the Oracle Java documentation for details).
    - If a different encoding is selected in the preferences, SOAtest or Virtualize will always invoke BytesMessage.readBytes() on the response messages in order to account for different character encodings.
- **StreamMessage:** Used to send a stream of primitive values. If you are doing SOAP over JMS, the Stream you enter into the StreamMessage should of the SOAP Envelope.
- **ObjectMessage:** Used to send a Java Serializable. You should use the Scripting input view to return a Java Serializable. This object will be taken and stuck into the ObjectMessage during runtime.
- **MapMessage:** Used to send a set of name-value pairs. The values can only be java primitives or their respective wrappers, Strings, or byte arrays. One intricacy of MapMessage objects is that even though you inserted a value as a String (using setString()), if the value can be coerced into an integer we can call getInt() and get a value of type integer.

Request Message Properties

The Request Message Properties are optional and allows for any miscellaneous property values to be set into the javax.jms.Message object before it gets sent to a queue or published to a topic. These include predefined properties that get set to the outgoing requests message using one of the corresponding "set" methods in javax.jms.Message, or custom properties (for example, properties set with setStringProperty, setBooleanProperty, setByteProperty, etc.)

Property values, which can be added by clicking **Add** and completing the Add JMS Property dialog, can be set to a fixed value, a parameterized value, a scripted value, or a unique value (an automatically-generated random unique value—no two tool invocations will use the same value). Be sure to specify the type for each added value.



Response Message Correlation

The Response Message Correlation settings contain the following options:

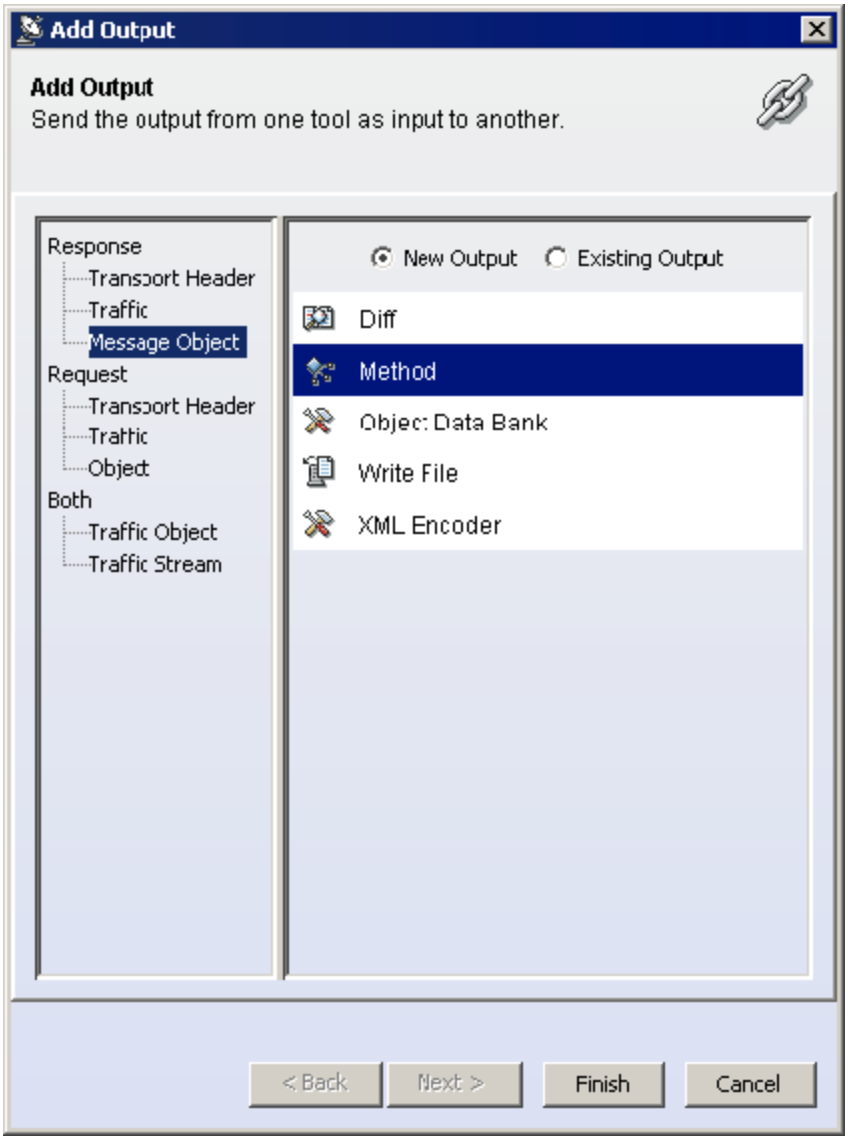
- **Match response JMSCorrelationID with request JMSMessageID:** If selected, the term JMSCorrelationID = '[msgId]' will be appended to the selector expression, where msgId is dynamically generated from the outgoing (request) javax.jms.Message (using the getJMSMessageID() method). Effectively, this results in the tool blocking until a message with the specified correlation id becomes available in the queue (or topic) and it will only retrieve that particular message, rather than retrieving any message in the queue (or topic). The tool will timeout after the timeout amount elapses and if there is no message that watches the selector criteria.
- **Match response JMSCorrelationID with request JMSCorrelationID:** If selected, the term JMSCorrelationID = '[correlationId]' will be appended to the selector expression, where correlationId is retrieved from JMSCorrelationID property in the Message Properties section. The option becomes enabled only if such property is added to the Message Properties section. Effectively, this results in the tool blocking until a message with the specified correlation id becomes available in the queue (or topic) and it will only retrieve that particular message, rather than retrieving any message in the queue (or topic). The tool will timeout after the timeout amount elapses and if there is no message that watches the selector criteria.
- **Additional Selector Expression Terms:** (Optional) Enter a value to act as a message filter. For example, by entering `username==John`, only messages that contain "John" as a username will be delivered. If this field is left blank, then any messages can be received from the queue. This expression is passed to the createReceiver() method of the javax.jms.QueueSession class in point to point messaging, or the createSubscriber() method of the javax.jms.TopicSession class in publish and subscribe messaging. For tips on specifying a selector, see Using Message Selector Filters.

Message Object Outputs for Clients Using JMS

You can add message object outputs to SOAP Clients and Messaging Clients that utilize the JMS transport. For example, an Extension tool chained to a Messaging Client that uses JMS will have access to the response JMS Message. In the ObjectMessage case, you can use `getter` and `equals()` method s to validate the response thereby creating a regression control. In addition you can chain a Diff tool to the Response Traffic and if the response is an ObjectMessage, SOAtest will convert the inserted serializable object to XML format and perform an XML diff. By doing this you can use data bank values, ignore XPath differences, etc.

To do this complete the following:

1. Right-click the SOA Client or Messaging Client node for which you would like to add the output and select **Add Output** from the shortcut menu. The Add Output wizard displays.



2. Select **Response> Message Object** in the left pane of the Add Output wizard and then choose a **New Output** or **Existing Output** and the desired tool (e.g. an Extension tool) from the right pane.
3. Click the **Finish** button. SOAtest adds the new output to the selected Client node.

Configuration for Popular JMS Providers

See JMS Provider Configuration

Responding to a Temporary JMSReplyTo Queue

To respond to a JMS message with a temporary queue set in the JMSReplyTo field:

1. Create a Call Back Tool and configure it to receive the JMS message.
2. Right-click the Call Back Tool and choose **Add Output> Incoming JMS Message> Extension tool**.
3. Write a custom script that places the incoming message (input object) into the scripting context with a predetermined key.
    - This key is defined within the SOAtest Extensibility API under SOAPUtil.JMS_MSG_KEY. To access the SOAtest Extensibility API, choose **Parasoft> Help**, then look for the book titled "Parasoft SOAtest Extensibility API".
    - Here is a Jython example:
      ```
      from soaptest.api import *

      def getJMSReplyTo(input, context):
        context.put(SOAPUtil.JMS_MSG_KEY, input)
      ```
4. Create and configure the tool that will send the JMS response message.
    - SOAPUtil.JMS_CONTEXT_QUEUE, which is another keyword defined in the SOAtest Extensibility API, should be used as the JMSDestination of this tool.

SOAtest will then use the temporary queue of the received message as the destination in place of the keyword entered as the destination.

Using Message Selector Filters

In various tools, you can specify a value to act as a message filter. This is specified in a field labelled **Message Selector** or **Additional Selector Expression Terms**.

For example, by entering `username==John`, only messages that contain "John" as a username will be delivered. If this field is left blank, then any messages can be received.

Here are some tips for working with message selector filters:

- You can parameterize Additional Selector Expression Terms values against variables, environment variables, and data source values. The syntax to reference variables and environment variables is ${myVariableName}. The syntax to reference XML Data Bank values and Data Source Values is: ${myColumnName}. For example, you could use JMSCorrelationID ='${myColumnName}.
- If you'd like to filter event messages based on content, you can chain a tool such as XML Transformer to its XML Event Output.
- The selector field links to a JMS feature called "message selectors." The specified expression is passed to the javax.jms.Session object when creating a javax.jms.MessageConsumer. The selector expression, as defined by the JMS specification, can be applied to JMS message headers and properties, but not to message body contents.
- In point to point messaging, this expression is passed to the createReceiver() method of the javax.jms.QueueSession class. In publish and subscribe messaging, it is passed to the create-Subscriber() method of the javax.jms.TopicSession class.
- The expression syntax is a subset of SQL92. For example, if the expression is `fruit = 'apple'` or `JMSCorrelationID = '123456'`, only JMS messages that have the property `fruit` defined with the value `apple` or the JMS header JMSCorrelationID set to the value `123456` will be picked up.
- For more information, refer to the Java Message Service Specification at the Oracle Java site.

JMS Messaging without JNDI

Using JNDI in order to obtain JMS connection factory and Destination instances is highly recommended from an architectural perspective because it decouples JMS consumer code from vendor-specific dependencies. In test or staging environments, a JMS system occasionally does not have a JNDI configured yet, or the JNDI does not yet include the desired connection factories. Furthermore, it might be helpful to bypass JNDI during testing in order to debug issues or isolate system performance characteristics with and without JNDI.

For these reasons, SOAtest's or Virtualize's JMS-aware tools allow for sending and receiving JMS messages using vendor connection factories directly—without going through JNDI—as long as your JMS provider permits this.

This capability supports certain JMS implementations that are designed to allow for JMS connections to be established without JNDI in the first place, and which provide connection factory classes with a constructor that takes a single string argument as the connection URL.

Supported JMS Implementations

Since this support for JMS messaging without JNDI is not based on standard JMS API, it is not guaranteed to be portable across different JMS implementations.

This capability has been tested with Sonic and TIBCO JMS. It is also supported for WebSphere MQ, with the configuration described in IBM WebSphere MQ (MQ Series).

At the time of this writing, the direct creation of connection factories for Oracle/BEA WebLogic, JBoss, or WebSphere Default JMS provider is not supported by SOAtest or Virtualize and in most of these cases, it is not documented or encouraged by these vendors.

Configuration

To configure one of the JMS messaging tools in SOAtest or Virtualize to send/receive messages without JNDI:

- Keep the connection URL in the JNDI connection settings **Provider URL** field.
- Leave the **Initial Context** field empty.
- Specify a fully-qualified MOM connection factory class name in the **Connection Factory** field. That classname should represent a class that implements javax.jms.ConnectionFactory.
    - In the cases of Sonic and TIBCO, you can follow the same patterns as JNDI provider URLs (see Progress Sonic MQ/ESB and TIBCO EMS for details).

The empty **Initial Context** field is interpreted as a signal to instantiate the connection factory object directly and without JNDI. It will attempt to do so with a constructor that takes a single string argument, passing in the connection URL that you specified in the **Provider URL** field. The connection authentication settings will still be used as usual. If no such constructor exists in your provider's connection factory API, then JNDI is needed in order to instantiate the connection factory class.

For example, in the case of Sonic JMS, the **Connection Factory** class name to be provided would be `progress.message.jclient.ConnectionFactory`. For TIBCO JMS, it would be `com.tibco.tibjms.TibjmsConnectionFactory`. On the other hand, JBoss and OpenJMS do not have a connection factory that takes a single connection URL argument. In these cases, there must be a JNDI in order to exchange JMS messages with these systems using SOAtest or Virtualize.

Regardless of the JNDI settings configuration, SOAtest and Virtualize will always attempt to resolve the destination (queue or topic) name from JNDI if JNDI exists and if the name exists in the directory. If it cannot find it via JNDI, then it will attempt to create the destination directly from the JMS Session instance—assuming that the user-provided name is the physical destination name and not a JNDI name.

Note that Sun's JNDI implementation is available as well. See Sun JMS for details.

Browsing Queue Contents

The Queue Browser allows you to see the contents of queues deployed on Websphere MQ, Websphere Application Server, Tibco EMS, Sonic MQ, ActiveMQ, and any other JMS provider. See Browsing Queues

for details.