Stomp

Connectivity > Protocols > Stomp

ActiveMQ Classic supports the Stomp protocol and the Stomp - JMS mapping. This makes it easy to write a client in pure Ruby, Perl, Python or PHP for working with ActiveMQ Classic.

Please see the Stomp site for more details

Spec Compliance

ActiveMQ Classic v5.6 implements the Stomp v1.1 spec except for allowing spaces at the beginning or end of message header keys, they are preserved in the header values however. In future releases this will not be the case, clients should be updated and user code checked to ensure that spaces in the headers are there intentionally and not as a accident or a client "feature".

Enabling the ActiveMQ Classic Broker for Stomp

To enable STOMP protocol support in the broker add a transport connector definition whose URI scheme is `stomp`.

Example:

```
<transportConnectors>
   <transportConnector name="stomp" uri="stomp://localhost:61613"/>
</transportConnectors>
```

To see a full example, try this XML. If you save that XML as `foo.xml` then you can run stomp via the command line as

```
activemq xbean:foo.xml
```

For more help see Run Broker.

The Stomp Wire Format

Stomp uses a text based wire format that can be configured with the following options. All options can be configured on a Brokers transport bind URI.

Parameter Name	Default Value	Description
`maxDataLength`	`104857600`	Maximum size of the message body (content) that can be sent.
`maxFrameSize`	`MAX_LONG`	**From ActiveMQ Classic 5.12.0**: maximum frame size that can be sent. A Stomp frame includes a command, optional headers, and an optional body. Can help help prevent OOM DOS attacks

Example:

```
<transportConnector name="stomp+ssl" uri="stomp+ssl://localhost:61612?wireFormat.maxFrameSize=1000000"/>
```

Use the Correct Prefix!

Wire format options must have the prefix `wireFormat.` to take effect, e.g., `wireFormat.`maxDataLength`=100000`. Options missing this prefix will be ignored.

Security

From ActiveMQ Classic 5.1: Stomp fully supports ActiveMQ Classic's security mechanism. This means that the `CONNECT` command will return an `ERROR` STOMP frame on unsuccessful authentication. Also, the authorization policies will be applied when you try to access (read/write) certain destinations. If you use synchronous operations (by using receipts), you can expect an `ERROR` frame in case

SSL

For additional security, you can use Stomp over SSL as described in the following section.

Enabling Stomp over NIO

From ActiveMQ Classic 5.3: for better scalability and performance the Stomp protocol can be configured to be run over the NIO transport. The NIO transport will use far fewer threads than the corresponding TCP connector. This can help when support for a large number of queues is required. To use NIO change the URI scheme of the transport connector to `stomp+nio`.

Example:

```
<transportConnector name="stomp+nio" uri="stomp+nio://localhost:61612"/>
```

Enabling Stomp over SSL

To configure ActiveMQ Classic to use Stomp over an SSL connection change the URI scheme to `stomp+ssl`.

Example:

```
<transportConnector name="stomp+ssl" uri="stomp+ssl://localhost:61612"/>
```

For more details on using SSL with ActiveMQ Classic see the following article (How do I use SSL). An example of using Stomp over SSL on the client side can be found in the PHP Stomp client example.

Heart-Beat Grace Period

The STOMP protocol (version 1.1 or greater) defines the concept of heart beats as a method by which a client and broker can determine the health of the underlying TCP connection between them. ActiveMQ Classic supports STOMP heart beating provided the client is using version 1.1 (or greater) of the protocol.

Before ActiveMQ Classic 5.9.0: enforcement of the 'read' heart-beat timeout (that is, a heart-beat sent from the client to the broker) was strict. In other words, the broker was intolerant of late arriving read heart-beats from the client. This resulted in the broker concluding that the client was no longer present causing it to close its side of the client's connection when the client failed to honor it's configured heart-beat settings.

From ActiveMQ Classic 5.9.0: the timeout enforcement for read heart-beats is now configurable via a new transport option `transport.hbGracePeriodMultiplier`:

```
<transportConnectors>
   <transportConnector name="stomp" uri="stomp://localhost:61613?transport.hbGracePeriodMultiplier=1.5"/>
</transportConnectors>
```

This multiplier is used to calculate the effective read heart-beat timeout the broker will enforce for each client's connection. The multiplier is applied to the read-timeout interval the client specifies in its `CONNECT` frame:

```
<client specified read heart-beat interval> * <grace periodmultiplier> == <broker enforced read heart-beat
timeout interval>
```

For backward compatibility, if the grace period multiplier is not configured the default enforcement mode remains strict, e.g., `transport.hbGracePeriodMultiplier=1.0`. Attempts to configure the grace period multiplier to a value less than, or equal to `1.0` will be silently ignored.

STOMP clients that wish to be tolerant of late arriving heart-beats from the broker must implement their own solution for doing so.

- Please check the STOMP specification for the details on heart-beating
- The JIRA that implemented this: ActiveMQ Classic 5.x does not support the notion of a grace-period for heart beats as supported by the STOMP protocol

destination. Also note that the default separator in MOM systems is `.` (dot). Whilst `FOO.BAR` is the normal syntax to identify a queue type destination the Stomp equivalent is `/queue/FOO.BAR`

Be careful about starting destinations with `/`

If in Stomp world you use `/queue/foo/bar` then in a JMS world the queue would be called `foo/bar` not `/foo/bar`.

Persistent Messaging in STOMP

STOMP messages are non-persistent by default. To use persistent messaging add the following STOMP header to all `SEND` requests: `persistent:true`. This default is the opposite of that for JMS messages.

Working with JMS Text/Bytes Messages and Stomp

Stomp is a very simple protocol - that's part of the beauty of it! As such, it does not have knowledge of JMS messages such as `TextMessage`'s or `BytesMessage`'s. The protocol does however support a `content-length` header. To provide more robust interaction between STOMP and JMS clients, ActiveMQ Classic keys off of the inclusion of this header to determine what message type to create when sending from Stomp to JMS. The logic is simple:

Inclusion of content-length header	Resulting Message
yes	`BytesMessage`
no	`TextMessage`

This same logic can be followed when going from JMS to Stomp, as well. A Stomp client could be written to key off of the inclusion of the `content-length` header to determine what type of message structure to provide to the user.

Message Transformations

The `transformation` message header on `SEND` and `SUBSCRIBE` messages could be used to instruct ActiveMQ Classic to transform messages from text to the format of your desire. Currently, ActiveMQ Classic comes with a transformer that can transform XML/JSON text to Java objects, but you can add your own transformers as well.

Here's a quick example of how to use built-in transformer (taken from test cases)

```
        + "</pojo>";

public void testTransformationReceiveXMLObject() throws Exception {

    MessageProducer producer = session.createProducer(new ActiveMQQueue("USERS." + getQueueName()));
    ObjectMessage message = session.createObjectMessage(new SamplePojo("Dejan", "Belgrade"));
    producer.send(message);

    String frame = "CONNECT\n" + "login: system\n" + "passcode: manager\n\n" + Stomp.NULL;
    stompConnection.sendFrame(frame);

    frame = stompConnection.receiveFrame();
    assertTrue(frame.startsWith("CONNECTED"));

    frame = "SUBSCRIBE\n" + "destination:/queue/USERS." + getQueueName() + "\n" + "ack:auto" + "\n" +
"transformation:jms-object-xml\n\n" + Stomp.NULL;
    stompConnection.sendFrame(frame);

    frame = stompConnection.receiveFrame();
    assertTrue(frame.trim().endsWith(xmlObject));

    frame = "DISCONNECT\n" + "\n\n" + Stomp.NULL;
    stompConnection.sendFrame(frame);
}
```

Dependencies

ActiveMQ Classic uses XStream for its transformation needs. Since it's the optional dependency you have to add it to broker's classpath by putting the appropriate JAR into the `lib/` folder. Additionally, if you plan to use JSON transformations you have to add Jettison JSON parser to the classpath.

In order to create your own transformer, you have to do the following:

1. Build your transformer by implementing a FrameTranslator interface
2. Associate it with the appropriate header value by creating a file named as a value you want to use in the `META-INF/services/org/apache/activemq/transport/frametranslator/` folder of your JAR which will contain the value `class=_fully qualified classname of your transformer_`

For example the built-in transformer contains the following value:

```
class=org.apache.activemq.transport.stomp.XStreamFrameTranslator
```

in the `META-INF/services/org/apache/activemq/transport/frametranslator/jms-xml` file.

Debugging

In case you want to debug Stomp communication between broker and clients you should configure the Stomp connector with the `trace` parameter, like this:

```
<transportConnectors>
   <transportConnector name="stomp" uri="stomp://localhost:61613?trace=true"/>
</transportConnectors>
```

This will instruct the broker to trace all packets it sends and receives.

Furthermore, you have to enable tracing for the appropriate log. You can achieve that by adding the following to your `conf/log4j.properties`

```
log4j.logger.org.apache.activemq.transport.stomp=TRACE
```

```
log4j.appender.stomp=org.apache.log4j.RollingFileAppender
log4j.appender.stomp.file=${activemq.base}/data/stomp.log
log4j.appender.stomp.maxFileSize=1024KB
log4j.appender.stomp.maxBackupIndex=5
log4j.appender.stomp.append=true
log4j.appender.stomp.layout=org.apache.log4j.PatternLayout
log4j.appender.stomp.layout.ConversionPattern=%d \[%-15.15t\] %-5p %-30.30c{1} - %m%n

log4j.logger.org.apache.activemq.transport.stomp=TRACE, stomp
log4j.additivity.org.apache.activemq.transport.stomp=false

# Enable these two lines and disable the above two if you want the frame IO ONLY (e.g., no heart beat messages,
inactivity monitor etc).
#log4j.logger.org.apache.activemq.transport.stomp.StompIO=TRACE, stomp
#log4j.additivity.org.apache.activemq.transport.stomp.StompIO=false
```

After this, all your Stomp packets will be logged to the `data/stomp.log`

Java API

From ActiveMQ Classic 5.2: there is a simple Java Stomp API distributed with ActiveMQ Classic. Note that this API is provided purely for testing purposes and you should always consider using standard JMS API from Java instead of this one. The following code snippet provides a simple example of using this API:

```
StompConnection connection = new StompConnection();
connection.open("localhost", 61613);

connection.connect("system", "manager");
StompFrame connect = connection.receive();

if(!connect.getAction().equals(Stomp.Responses.CONNECTED)) {
        throw new Exception ("Not connected");
}

connection.begin("tx1");
connection.send("/queue/test", "message1", "tx1", null);
connection.send("/queue/test", "message2", "tx1", null);
connection.commit("tx1");

connection.subscribe("/queue/test", Subscribe.AckModeValues.CLIENT);

connection.begin("tx2");

StompFrame message = connection.receive();
System.out.println(message.getBody());
connection.ack(message, "tx2");

message = connection.receive();
System.out.println(message.getBody());
connection.ack(message, "tx2");

connection.commit("tx2");

connection.disconnect();
```

This example is part of the standard ActiveMQ Classic distribution. You can run it from the `./example` folder with:

```
ant stomp
```

Stomp Extensions for JMS Message Semantics

Note that STOMP is designed to be as simple as possible - so any scripting language/platform can message any other with minimal effort. STOMP allows pluggable headers on each request such as sending & receiving messages. ActiveMQ Classic has several extensions to the Stomp protocol, so that JMS semantics can be supported by Stomp clients. An OpenWire JMS producer can send messages to a Stomp consumer, and a Stomp producer can send messages to an OpenWire JMS consumer. And Stomp to Stomp configurations, can use the richer JMS message control.

`correlation-id`	`JMSCorrelationID`	Good consumers will add this header to any responses they send.
`expires`	`JMSExpiration`	Expiration time of the message.
`JMSXGroupID`	`JMSXGroupID`	Specifies the Message Groups.
`JMSXGroupSeq`	`JMSXGroupSeq`	Optional header that specifies the sequence number in the Message Groups.
`persistent`	`JMSDeliveryMode`	Whether or not the message is persistent.
`priority`	`JMSPriority`	Priority on the message.
`reply-to`	`JMSReplyTo`	Destination you should send replies to.
`type`	`JMSType`	Type of the message.

ActiveMQ Classic Extensions to STOMP

You can add custom headers to STOMP commands to configure the ActiveMQ Classic protocol. Here are some examples:

Verb	Header	Type	Description
`CONNECT`	`client-id`	`string`	Specifies the JMS clientID which is used in combination with the `activemq.subcriptionName` to denote a durable subscriber.
`SUBSCRIBE`	`activemq.dispatchAsync`	`boolean`	Should messages be dispatched synchronously or asynchronously from the producer thread for non-durable topics in the broker? For fast consumers set this to `false`. For slow consumers set it to `true` so that dispatching will not block fast consumers.
`SUBSCRIBE`	`activemq.exclusive`	`boolean`	I would like to be an Exclusive Consumer on the queue.
`SUBSCRIBE`	`activemq.maximumPendingMessageLimit`	`int`	For Slow Consumer Handling on non-durable topics by dropping old messages - we can set a maximum-pending limit, such that once a slow consumer backs up to this high water mark we begin to discard old messages.
`SUBSCRIBE`	`activemq.noLocal`	`boolean`	Specifies whether or not locally sent messages should be ignored for subscriptions. Set to `true` to filter out locally sent messages.
`SUBSCRIBE`	`activemq.prefetchSize`	`int`	Specifies the maximum number of pending messages that will be dispatched to the client. Once this maximum is reached no more messages are dispatched until the client acknowledges a message. Set to a low value > **1** for fair distribution of messages across consumers when processing messages can be slow. **Note**: if your STOMP client is implemented using a dynamic scripting language like Ruby, say, then this parameter ***must*** be set to `1` as there is no notion of a client-side message size to be sized. STOMP does not support a value of `0`.
`SUBSCRIBE`	`activemq.priority`	`byte`	Sets the priority of the consumer so that dispatching can be weighted in priority order.
`SUBSCRIBE`	`activemq.retroactive`	`boolean`	For non-durable topics make this subscription retroactive.

SUBSCRIBE	activemq.subscriptionName	string	same activemq.client-id on the connection and activemq.subcriptionName on the subscribe prior to v5.7.0. **Note**: the spelling subcriptionName NOT subscriptionName. This is not intuitive, but it is how it is implemented in ActiveMQ Classic 4.x. For the 5.0 release of ActiveMQ Classic, both subcriptionName and subscriptionName will be supported (subcriptionName was removed as of v5.6.0).
SUBSCRIBE	selector	string	Specifies a JMS Selector using SQL 92 syntax as specified in the JMS 1.1 specification. This allows a filter to be applied to each message as part of the subscription.