RabbitMQ JMS Client

Introduction

RabbitMQ JMS Client is a client library for Pivotal RabbitMQ. RabbitMQ is not a JMS provider but includes a plugin needed to support the JMS Queue and Topic messaging models. JMS Client for RabbitMQ implements the JMS 1.1 specification on top of the RabbitMQ Java client, thus allowing new and existing JMS applications to connect to RabbitMQ.

The plugin and the JMS client are meant to work and be used together.

See the RabbitMQ Java libraries support page for the support timeline of the RabbitMQ JMS Client library.

Components

To fully leverage JMS with RabbitMQ, you need the following components:

- the JMS client library and its dependent libraries.
- RabbitMQ JMS topic selector plugin that is included with RabbitMQ starting with version 3.6.3. To support message selectors for JMS topics, the RabbitMQ Topic Selector plugin must be installed on the RabbitMQ server. Message selectors allow a JMS application to filter messages using an expression based on SQL syntax. Message selectors for Queues are not currently supported.

JMS and AMQP 0-9-1

JMS is the standard messaging API for the JEE platform. It is available in commercial and open source implementations. Each implementation includes a JMS provider, a JMS client library, and additional, implementation-specific components for administering the messaging system. The JMS provider can be a standalone implementation of the messaging service, or a bridge to a non-JMS messaging system.

The JMS client API is standardized, so JMS applications are portable between vendors' implementations. However, the underlying messaging implementation is unspecified, so there is no interoperability between JMS implementations. Java applications that want to share messaging must all use the same JMS implementation unless bridging technology exists. Furthermore, non-Java applications cannot access JMS without a vendor-specific JMS client library to enable interoperability.

AMQP 0-9-1 is a messaging protocol, rather than an API like JMS. Any client that implements the protocol can access a broker that supports AMQP 0-9-1. Protocol-level interoperability allows AMQP 0-9-1 clients written in any programming language and running on any operating system to participate in the messaging system with no need to bridge incompatible vendor implementations.

Because JMS Client for RabbitMQ is implemented using the RabbitMQ Java client, it is compliant with both the JMS API and the AMQP 0-9-1 protocol.

You can download the JMS 1.1 specification and API documentation from the Oracle Technology Network Web site.

Limitations

Some JMS 1.1 features are unsupported in the RabbitMQ JMS Client:

- The JMS Client does not support server sessions.
- XA transaction support interfaces are not implemented.
- Topic selectors are supported with the RabbitMQ JMS topic selector plugin. Queue selectors are not yet implemented.
- SSL and socket options for RabbitMQ connections are supported, but only using the (default) SSL connection protocols that the RabbitMQ client provides.
- The JMS `NoLocal` subscription feature, which prevents delivery of messages published from a subscriber's own connection, is not supported with RabbitMQ. You can call a method that includes the `NoLocal` argument, but it is ignored.

See the JMS API compliance documentation for a detailed list of supported JMS APIs.

Installing and Configuring

Enabling the Topic Selector Plug-in

The topic selector plugin is included with RabbitMQ. Like any RabbitMQ plugin, you need to enable the plugin in order to use it.

Enable the plugin using the `rabbitmq-plugins` command:

```
rabbitmq-plugins enable rabbitmq_jms_topic_exchange
```

You don't need to restart the broker to activate the plugin.

You need to enable this plugin only if you plan to use topic selectors in your JMS client applications.

Enabling the JMS client in a Java container

To enable the JMS Client in a Java container (e.g. Java EE application server, web container), you must install the JMS client JAR files and its dependencies in the container and then define JMS resources in the container's naming system so that JMS clients can look them up. The methods for accomplishing these tasks are container-specific, please refer to the vendors' documentation.

For standalone applications, you need to add the JMS client JAR files and its dependencies to the application classpath. The JMS resources can be defined programmatically or through a dependency injection framework like Spring.

Defining the JMS Connection Factory

To define the JMS `ConnectionFactory` in JNDI, e.g. in Tomcat:

```
<Resource name="jms/ConnectionFactory"
            type="javax.jms.ConnectionFactory"
         factory="com.rabbitmq.jms.admin.RMQObjectFactory"
        username="guest"
        password="guest"
     virtualHost="/"
            host="localhost"
            port="5672"/>
```

To define the JMS `ConnectionFactory` in JNDI, e.g. in WildFly (as of JMS Client 1.7.0):

```
<object-factory name="java:global/jms/ConnectionFactory"
                   module="org.jboss.genericjms.provider"
                   class="com.rabbitmq.jms.admin.RMQObjectFactory">
     <environment>
         <property name="className" value="javax.jms.ConnectionFactory"/>
         <property name="username" value="guest"/>
         <property name="password" value="guest"/>
         <property name="virtualHost" value="/"/>
         <property name="host" value="localhost"/>
         <property name="port" value="5672"/>
     </environment>
</object-factory>
```

Here is the equivalent Spring bean example (Java configuration):

```
@Bean
public ConnectionFactory jmsConnectionFactory() {
  RMQConnectionFactory connectionFactory = new RMQConnectionFactory();
  connectionFactory.setUsername("guest");
  connectionFactory.setPassword("guest");
  connectionFactory.setVirtualHost("/");
  connectionFactory.setHost("localhost");
  connectionFactory.setPort(5672);
  return connectionFactory;
}
```

And here is the Spring XML configuration:

```
<bean id="jmsConnectionFactory" class="com.rabbitmq.jms.admin.RMQConnectionFactory" >
  <property name="username" value="guest" />
  <property name="password" value="guest" />
  <property name="virtualHost" value="/" />
  <property name="host" value="localhost" />
  <property name="port" value="5672" />
</bean>
```

The following table lists all of the attributes/properties that are available.

Attribute/Property	JNDI only?	Description
`name`	Yes	Name in JNDI.
`type`	Yes	Name of the JMS interface the object implements, usually `javax.jms.ConnectionFactory`. Other choices are `javax.jms.QueueConnectionFactory` and `javax.jms.TopicConnectionFactory`. You can also use the name of the (common) implementation class, `com.rabbitmq.jms.admin.RMQConnectionFactory`.
`factory`	Yes	JMS Client for RabbitMQ `ObjectFactory` class, always `com.rabbitmq.jms.admin.RMQObjectFactory`.
`username`	No	Name to use to authenticate a connection with the RabbitMQ broker. The default is "guest".

Attribute/Property	JNDI only?	Description
`password`	No	Password to use to authenticate a connection with the RabbitMQ broker. The default is "guest".
`virtualHost`	No	RabbitMQ virtual host within which the application will operate. The default is "/".
`host`	No	Host on which RabbitMQ is running. The default is "localhost".
`port`	No	RabbitMQ port used for connections. The default is "5672" unless this is a TLS connection, in which case the default is "5671".
`ssl`	No	Whether to use an SSL connection to RabbitMQ. The default is "false". See the `useSslProtocol` methods for more information.
`uri`	No	The AMQP 0-9-1 URI string used to establish a RabbitMQ connection. The value can encode the `host`, `port`, `username`, `password` and `virtualHost` in a single string. Both 'amqp' and 'amqps' schemes are accepted. Note: this property sets other properties and the set order is unspecified.
`onMessageTimeoutMs`	No	How long to wait for `MessageListener#onMessage()` to return, in milliseconds. Default is 2000 ms.
`preferProducerMessageProperty`	No	Whether `MessageProducer` properties (delivery mode, priority, TTL) take precedence over respective `Message` properties or not. Default is true (which is compliant to the JMS specification).
`requeueOnMessageListenerException`	No	Whether requeuing messages on a `RuntimeException` in the `MessageListener` or not. Default is false.

JMS and AMQP 0-9-1 Destination Interoperability

An interoperability feature allows you to define JMS 'amqp' destinations that read and/or write to non-JMS RabbitMQ resources. ***Note this feature does not support JMS topics***.

A single 'amqp' destination can be defined for both sending and consuming.

Sending JMS Messages to an AMQP Exchange

A JMS destination can be defined so that a JMS application can send `Message`s to a predefined RabbitMQ 'destination' (exchange/routing key) using the JMS API in the normal way. The messages are written "in the clear," which means that any AMQP 0-9-1 client can read them without having to understand the internal format of Java JMS messages. ***Only `BytesMessage`s and `TextMessage`s can be written in this way***.

When messages are sent to an 'amqp' Destination, JMS message properties are mapped onto AMQP 0-9-1 headers and properties as appropriate. For example, the `JMSPriority` property converts to the `priority` property for the AMQP 0-9-1 message. (It is also set as a header with the name "JMSPriority".) User-defined properties are set as named message header values, provided they are `boolean`, numeric or `String` types.

Consuming Messages From an AMQP Queue

Similarly, a JMS destination can be defined that reads messages from a predefined RabbitMQ queue. A JMS application can then read these messages using the JMS API. RabbitMQ JMS Client packs them up into JMS Messages automatically. Messages read in this way are, by default, `BytesMessage`s, but individual messages can be marked `TextMessage` (by adding an AMQP message header called "JMSType" whose value is "TextMessage"), which will interpret the byte-array payload as a UTF8 encoded String and return them as `TextMessage`s.

When reading from an 'amqp' Destination, values are mapped back to JMS message properties, except that any explicit JMS property set as a message header overrides the natural AMQP 0-9-1 header value, unless this would misrepresent the message. For example, `JMSDeliveryMode` cannot be overridden in this way.

JMS 'amqp' `RMQDestination` Constructor

The `com.rabbitmq.jms.admin` package contains the `RMQDestination` class, which implements `Destination` in the JMS interface. This is extended with a new constructor:

```
public RMQDestination(String destinationName, String amqpExchangeName,
                      String amqpRoutingKey, String amqpQueueName);
```

This constructor creates a destination for JMS for RabbitMQ mapped onto an AMQP 0-9-1 resource. The parameters are the following:

- `destinationName` - the name of the queue destination
- `amqpExchangeName` - the exchange name for the mapped resource
- `amqpRoutingKey` - the routing key for the mapped resource
- `amqpQueueName` - the queue name of the mapped resource (to listen messages from)

Applications that declare destinations in this way can use them directly, or store them in a JNDI provider for JMS applications to retrieve. Such destinations are non-temporary, queue destinations.

JMS AMQP 0-9-1 Destination Definitions

The `RMQDestination` object has the following new instance fields:

- `amqp` – *boolean*, indicates if this is an AMQP 0-9-1 destination (if **true**); the default is **false**.
- `amqpExchangeName` – *String*, the RabbitMQ exchange name to use when sending messages to this destination, if `amqp` is **true**; the default is **null**.
- `amqpRoutingKey` – *String*, the AMQP 0-9-1 routing key to use when sending messages to this destination, if `amqp` is **true**; the default is **null**.
- `amqpQueueName` – *String*, the RabbitMQ queue name to use when reading messages from this destination, if `amqp` is **true**; the default is **null**.

There are getters and setters for these fields, which means that a JNDI `<Resource/>` definition or an XML Spring bean definition can use them, for example JNDI with Tomcat:

```
<Resource  name="jms/Queue"
           type="javax.jms.Queue"
        factory="com.rabbitmq.jms.admin.RMQObjectFactory"
destinationName="myQueue"
           amqp="true"
  amqpQueueName="rabbitQueueName"
/>
```

This is the equivalent with WildFly (as of JMS Client 1.7.0):

```
<bindings>
    <object-factory name="java:global/jms/Queue"
                    module="foo.bar"
                    class="com.rabbitmq.jms.admin.RMQObjectFactory">
        <environment>
            <property name="className" value="javax.jms.Queue"/>
            <property name="destinationName" value="myQueue"/>
            <property name="amqp" value="true"/>
            <property name="amqpQueueName" value="rabbitQueueName"/>
        </environment>
    </object-factory>
</bindings>
```

This is the equivalent Spring bean example (Java configuration):

```
    @Bean
    public Destination jmsDestination() {
        RMQDestination jmsDestination = new RMQDestination();
        jmsDestination.setDestinationName("myQueue");
        jmsDestination.setAmqp(true);
        jmsDestination.setAmqpQueueName("rabbitQueueName");
        return jmsDestination;
    }
```

And here is the Spring XML configuration:

```
    <bean id="jmsDestination" class="com.rabbitmq.jms.admin.RMQDestination" >
     <property name="destinationName" value="myQueue" />
     <property name="amqp"             value="true" />
     <property name="amqpQueueName"    value="rabbitQueueName" />
    </bean>
```

Following is a *complete* list of the attributes/properties that are available:

Attribute/Property Name	**JNDI Only?**	**Description**
`name`	Yes	Name in JNDI.
`type`	Yes	Name of the JMS interface the object implements, usually `javax.jms.Queue`. Other choices are `javax.jms.Topic` and `javax.jms.Destination`. You can also use the name of the (common) implementation class, `com.rabbitmq.jms.admin.RMQDestination`.
`factory`	Yes	JMS Client for RabbitMQ `ObjectFactory` class, always `com.rabbitmq.jms.admin.RMQObjectFactory`.
`amqp`	No	"**true**" means this is an 'amqp' destination. Default "**false**".
`amqpExchangeName`	No	Name of the RabbitMQ exchange to publish messages to when an 'amqp' destination. This exchange must exist when messages are published.
`amqpRoutingKey`	No	The routing key to use when publishing messages when an 'amqp' destination.
`amqpQueueName`	No	Name of the RabbitMQ queue to receive messages from when an 'amqp' destination. This queue must exist when messages are received.
`destinationName`	No	Name of the JMS destination.

Configuring Logging for the JMS Client

The JMS Client logs messages using SLF4J (Simple Logging Façade for Java). SLF4J delegates to a logging framework, such as Apache Logback. If no other logging framework is enabled, SLF4J defaults to a built-in, no-op, logger. See the SLF4J documentation for a list of the logging frameworks SLF4J supports.

The target logging framework is configured at deployment time by adding an SLF4J binding for the framework to the classpath. For example, the Logback SLF4J binding is in the `logback-classic-{version}.jar` file. To direct JMS client log messages to Logback, for example, add the following JARs to the classpath:

- slf4j-api-1.7.25.jar
- logback-core-1.2.3.jar
- logback-classic-1.2.3.jar

We highly recommend to use a dependency management tool like Maven or Gradle to manage dependencies.

The SLF4J API is backwards compatible, so you can use use any version of SLF4J. Version 1.7.5 or higher is recommended. The SLF4J API and bindings, however, must be from the same SLF4J version.

No additional SLF4J configuration is required, once the API and binding JAR files are in the classpath. However, the target framework may have configuration files or command-line options. Refer to the documentation for the target logging framework for configuration details.

Publisher Confirms

Publisher confirms are a RabbitMQ extension to implement reliable publishing. This feature builds on top of the AMQP protocol, but the JMS client provides an API to use it. This allows to benefit from a reliability feature without diverging too much from the JMS API.

Publisher confirms are disabled by default. They can be enabled by setting a `ConfirmListener` on the `RMQConnectionFactory`:

```
RMQConnectionFactory connectionFactory = new RMQConnectionFactory();
connectionFactory.setConfirmListener(context -> {
    context.getMessage(); // the message that is confirmed/nack-ed
    context.isAck(); // whether the message is confirmed or nack-ed
});
```

Note the `ConfirmListener` is not a good place to execute long-running tasks. Those should be executed in a dedicated thread, using e.g. an `ExecutorService`.

Typical operations in a `ConfirmListener` are logging or message re-publishing (in case of nacks). The publish confirms tutorial provides more guidance. It aims for the AMQP Java client, but principles remain the same for the JMS client.

Support for Request/Reply (a.k.a. RPC)

It is possible to use JMS for synchronous request/reply use cases. This pattern is commonly known as *Remote Procedure Call* or *RPC*.

With JMS API

An RPC client can be implemented in pure JMS like the following:

```
Message request = ... // create the request message
// set up reply-to queue and start listening on it
Destination replyQueue = session.createTemporaryQueue();
message.setJMSReplyTo(replyQueue);
MessageConsumer responseConsumer = session.createConsumer(replyQueue);
BlockingQueue<Message> queue = new ArrayBlockingQueue<>(1);
responseConsumer.setMessageListener(msg -> queue.add(msg));
// send request message
MessageProducer producer = session.createProducer("request.queue");
producer.send(request);
// wait response for 5 seconds
Message response = queue.poll(5, TimeUnit.SECONDS);
```

```
// close the response consumer
responseConsumer.close();
```

It's also possible to create a single reply-to destination instead of a temporary destination for each request. This is more efficient but requires to properly correlate the response with the request, by using e.g. a correlation ID header. RabbitMQ's direct reply-to is another alternative (see below).

Note this sample uses a `MessageListener` and a `BlockingQueue` to wait for the response. This implies a network roundtrip to register an AMQP consumer and another one to close the consumer. `MessageConsumer#receive` could have been used as well, in this case the JMS client internally polls the reply destination to get the response, which can result in several network roundtrips if the response takes some time to come. The request call will also incur a constant penalty equals to the polling interval (100 milliseconds by default).

The server part looks like the following:

```
// this is necessary when using temporary reply-to destinations
connectionFactory.setDeclareReplyToDestination(false);
...
MessageProducer replyProducer = session.createProducer(null);
MessageConsumer consumer = session.createConsumer("request.queue");
consumer.setMessageListener(message -> {
    try {
        Destination replyQueue = message.getJMSReplyTo();
        if (replyQueue != null) {
            // create response and send it
            Message response = ...
            replyProducer.send(replyQueue, response);
        }
    } catch (JMSException e) {
        // deal with exception
    }
});
```

Note the `connectionFactory.setDeclareReplyToDestination(false)` statement: it is necessary when using temporary reply-to destinations. If this flag is not set to `false` on the RPC server side, the JMS client will try to re-create the temporary reply-to destination, which will interfere with the client-side declaration.

See this test for a full RPC example.

The JMS client also supports direct reply-to, which is faster as it doesn't imply creating a temporary reply destination:

```
Message request = ...
// use direct reply-to
RMQDestination replyQueue = new RMQDestination(
    "amq.rabbitmq.reply-to", "", "amq.rabbitmq.reply-to", "amq.rabbitmq.reply-to"
);
replyQueue.setDeclared(true); // don't need to create this destination
message.setJMSReplyTo(replyQueue);
MessageConsumer responseConsumer = session.createConsumer(replyQueue);
BlockingQueue<Message> queue = new ArrayBlockingQueue<>(1);
responseConsumer.setMessageListener(msg -> queue.add(msg));
// send request message
MessageProducer producer = session.createProducer("request.queue");
producer.send(request);
```

```
// wait response for 5 seconds
Message response = queue.poll(5, TimeUnit.SECONDS);
// close the response consumer
responseConsumer.close();
```

Using direct reply-to for JMS-based RPC has the following implications:

- it uses automatically auto-acknowledgment
- the response must be a `BytesMessage` or a `TextMessage` as direct reply-to is considered an AMQP destination. Use `response.setStringProperty("JMSType", "TextMessage")` on the response message in the RPC server if you want to receive a `TextMessage` on the client side.

See this test for a full RPC example using direct reply-to.

With Spring JMS

Spring JMS is a popular way to work with JMS as it avoids most of JMS boilerplate.

The following sample shows how a client can perform RPC with the `JmsTemplate`:

```
// NB: do not create a new JmsTemplate for each request
JmsTemplate tpl = new JmsTemplate(connectionFactory);
tpl.setReceiveTimeout(5000);
Message response = tpl.sendAndReceive(
    "request.queue",
    session -> ... // create request message in MessageCreator
);
```

This is no different from any other JMS client.

The `JmsTemplate` uses a temporary reply-to destination, so the call to `connectionFactory.setDeclareReplyToDestination(false)` on the RPC server side is necessary, just like with regular JMS.

RPC with direct reply-to must be implemented with a `SessionCallback`, as the reply destination must be explicitly declared:

```
// NB: do not create a new JmsTemplate for each request
JmsTemplate tpl = new JmsTemplate(connectionFactory);
Message response = tpl.execute(session -> {
    Message request = ... // create request message
    // setup direct reply-to as reply-to destination
    RMQDestination replyQueue = new RMQDestination(
        "amq.rabbitmq.reply-to", "", "amq.rabbitmq.reply-to", "amq.rabbitmq.reply-to"
    );
    replyQueue.setDeclared(true); // no need to create this destination
    message.setJMSReplyTo(replyQueue);
    MessageConsumer responseConsumer = session.createConsumer(replyQueue);
    BlockingQueue<Message> queue = new ArrayBlockingQueue<>(1);
    responseConsumer.setMessageListener(msg -> queue.add(msg));
    // send request message
    MessageProducer producer = session.createProducer(session.createQueue("request.queue"));
    producer.send(message);
    try {
        // wait response for 5 seconds
        Message response = queue.poll(5, TimeUnit.SECONDS);
        // close the response consumer
        responseConsumer.close();
        return response;
    } catch (InterruptedException e) {
        // deal with exception
    }
});
```

See this test for a full example of RPC with Spring JMS, including using a `@JmsListener` bean for the server part.

Implementation Details

This section provides additional implementation details for specific JMS API classes in the JMS Client.

Deviations from the specification are implemented to support common acknowledgement behaviours.

JMS Topic Support

JMS topics are implemented using an AMQP topic exchange and a dedicated AMQP queue for each JMS topic subscriber. The AMQP topic exchange is `jms.temp.topic` or `jms.durable.topic`, depending on whether the JMS topic is temporary or not, respectively. Let's take an example with a subscription to a durable `my.jms.topic` JMS topic:

- a dedicated AMQP queue is created for this subscriber, its name will follow the pattern `jms-cons-{UUID}`.
- the `jms-cons-{UUID}` AMQP queue is bound to the `jms.durable.topic` exchange with the `my.jms.topic` binding key.

If another subscriber subscribes to `my.jms.topic`, it will have its own AMQP queue and both subscribers will receive messages published to the `jms.durable.topic` exchange with the `my.jms.topic` routing key.

The example above assumes no topic selector is used when declaring the subscribers. If a topic selector is in use, a `x-jms-topic`-typed exchange will sit between the `jms.durable.topic` topic exchange and the subscriber queue. So the topology is the following when subscribing to a durable `my.jms.topic` JMS topic with a selector:

- a dedicated AMQP queue is created for this subscriber, its name will follow the pattern `jms-cons-{UUID}`.
- a `x-jms-topic`-typed exchange is bound to the subscriber AMQP queue with the `my.jms.topic` binding key and some arguments related to the selector expressions. Note this exchange is scoped to the JMS session and not only to the subscriber.
- the `x-jms-topic`-typed exchange is bound to the `jms.durable.topic` exchange with the `my.jms.topic` binding key.

Exchanges can be bound together thanks to a RabbitMQ extension. Note the topic selector plugin must be enabled for topic selectors to work.

QueueBrowser Support

Overview of queue browsers

The JMS API includes objects and methods to browse an existing queue destination, reading its messages *without* removing them from the queue. Topic destinations cannot be browsed in this manner.

A `QueueBrowser` can be created from a (queue) `Destination`, with or without a selector expression. The browser has a `getEnumeration()` method, which returns a Java `Enumeration` of `Message`s copied from the queue.

If no selector is supplied, then all messages in the queue appear in the `Enumeration`. If a selector is supplied, then only those messages that satisfy the selector appear.

Implementation

The destination queue is read when the `getEnumeration()` method is called. A *snapshot* is taken of the messages in the queue; and the selector expression, if one is supplied, is used at this time to discard messages that do not match.

The message copies may now be read using the `Enumeration` interface (`nextElement()` and `hasMoreElements()`).

The selector expression and the destination queue of the `QueueBrowser` may not be adjusted after the `QueueBrowser` is created.

An `Enumeration` cannot be "reset", but the `getEnumeration()` method may be re-issued, taking a *new* snapshot from the queue each time.

The contents of an `Enumeration` survive session and/or connection close, but a `QueueBrowser` may not be used after the session that created it has closed. `QueueBrowser.close()` has no effect.

Which messages are included

Messages that arrive, expire, are re-queued, or are removed after the `getEnumeration()` call have no effect on the contents of the `Enumeration` it produced. If the messages in the queue change *while the* `Enumeration` *is being built*, they may or may not be included. In particular, if messages from the queue are simultaneously read by another client (or session), they may or may not appear in the `Enumeration`.

Message copies do not "expire" from an `Enumeration`.

Order of messages

If other client sessions read from a queue that is being browsed, then it is possible that some messages may subsequently be received out of order.

Message order will not be disturbed if no other client sessions read the queue at the same time.

Memory usage

When a message is read from the `Enumeration` (with `nextElement()`), then no reference to it is retained in the Java Client. This means the storage it occupies in the client is eligible for release (by garbage collection) if no other references are retained. Retaining an `Enumeration` will retain the storage for all message copies that remain in it.

If the queue has many messages -- or the messages it contains are very large -- then a `getEnumeration()` method call may consume a large amount of memory in a very short time. This remains true even if only a few messages are selected. There is currently limited protection against `OutOfMemoryError` conditions that may arise because of this. See the next section.

Setting a maximum number of messages to browse

Each connection is created with a limit on the number of messages that are examined by a `QueueBrowser`. The limit is set on the `RMQConnectionFactory` by `RMQConnectionFactory.setQueueBrowserReadMax(int)` and is passed to each `Connection` subsequently created by `ConnectionFactory.createConnection()`.

The limit is an integer that, if positive, stops the queue browser from reading more than this number of messages when building an enumeration. If it is zero or negative, it is interpreted as imposing no limit on the browser, and all of the messages on the queue are scanned.

The default limit for a factory is determined by the `rabbit.jms.queueBrowserReadMax` system property, if set, and the value is specified as `0` if this property is not set or is not an integer.

If a `RMQConnectionFactory` value is obtained from a JNDI provider, then the limit set when the factory object was created is preserved.

Release Support

Support for `QueueBrowser`s is introduced in the JMS Client 1.2.0. Prior to that release, calling `Session.createBrowser(Queue queue[, String selector])` resulted in an `UnsupportedOperationException`.

Group and individual acknowledgement

Prior to version 1.2.0 of the JMS client, in client acknowledgement mode (`Session.CLIENT_ACKNOWLEDGE`), acknowledging any message from an open session would acknowledge *every* unacknowledged message of that session, whether they were received before or after the message being acknowledged.

Currently, the behaviour of `Session.CLIENT_ACKNOWLEDGE` mode is modified so that, when calling `msg.acknowledge()`, only the message `msg` *and all* previously received *unacknowledged messages on that session* are acknowledged. Messages received *after* `msg` was received are not affected. This is a form of *group acknowledgement*, which differs slightly from the JMS 1.1 specification but is likely to be more useful, and is compatible with the vast majority of uses of the existing acknowledge function.

For even finer control, a new acknowledgement mode may be set when creating a session, called `RMQSession.CLIENT_INDIVIDUAL_ACKNOWLEDGE`.

A session created with this acknowledgement mode will mean that messages received on that session will be acknowledged individually. That is, the call `msg.acknowledge()` will acknowledge only the message `msg` and not affect any other messages of that session.

The acknowledgement mode `RMQSession.CLIENT_INDIVIDUAL_ACKNOWLEDGE` is equivalent to `Session.CLIENT_ACKNOWLEDGE` in all other respects. In particular the `getAcknowledgeMode()` method returns `Session.CLIENT_ACKNOWLEDGE` even if `RMQSession.CLIENT_INDIVIDUAL_ACKNOWLEDGE` has been set.

Arbitrary Message support

Any instance of a class that implements the `javax.jms.Message` interface can be *sent* by a JMS message producer.

All properties of the message required by `send()` are correctly interpreted except that the `JMSReplyTo` header and objects (as property values or the body of an `ObjectMessage`) that cannot be deserialized are ignored.

The implementation extracts the properties and body from the `Message` instance using interface methods and recreates it as a message of the right (`RMQMessage`) type (`BytesMessage`, `MapMessage`, `ObjectMessage`, `TextMessage`, or `StreamMessage`) before sending it. This means that there is some performance loss due to the copying; but in the normal case, when the message is an instance of `com.rabbitmq.jms.client.RMQMessage`, no copying is done.

Further Reading

To gain better understanding of AMQP 0-9-1 concepts and interoperability of the RabbitMQ JMS client with AMQP 0-9-1 clients, you may wish to read an Introduction to RabbitMQ Concepts and browse our AMQP 0-9-1 Quick Reference Guide.

Getting Help and Providing Feedback

If you have questions about the contents of this guide or any other topic related to RabbitMQ, don't hesitate to ask them on the RabbitMQ mailing list.

Help Us Improve the Docs <3

If you'd like to contribute an improvement to the site, its source is available on GitHub. Simply fork the repository and submit a pull request. Thank you!