JoramMQ 1.18
Rest API
User Manual

Content

🛠 **Warning :** The versions of Jetty and Jersey used by Joram and JoramMQ have evolved. Jetty is now in version 9.4.49 (Sept, 2022), and Jersey in version 2.37 (Sept, 2022). These evolutions require changes in the list of bundles needed in the OSGi configuration.

1 JMS Rest API

The JoramMQ JMS Rest API allows you to leverage the features of JoramMQ/JMS over a simple Rest/HTTP interface. Messages are produced and consumed through simple HTTP messages, this allows any web capable device to publish or consume messages using a regular HTTP Post or Get request.

1.1 Installation and configuration

1.1.1 Installation

The JoramMQ JMS Rest interface is installed as an OSGi bundle, it depends on the HTTP service and JAX-RS service implemented by the Jetty and Jersey bundles. You need to modify the default configuration (file 'conf/felix.properties')cas follows: (adding the lines colored in red)

```
felix.auto.start.1= \
 file:./bundle/monolog.jar \
 …
 file:./bundle/ow2-jms-2.0-spec.jar \
 file:./bundle/jakarta.jms-api.jar \
 file:./bundle/ow2-jta-1.1-spec.jar \
 …
 file:./bundle/org.osgi.compendium.jar \
 file:./bundle/gson.jar \
 file:./bundle/org.apache.felix.http.jetty.jar \
 file:./bundle/jndi-client.jar \
 file:./bundle/joram-client-jms.jar \
 file:./bundle/joram-jakarta-jms.jar \
 file:./bundle/joram-tools-rest-jms.jar
```

Then you need to stop the server, delete Felix bundle cache[1] 'data/felix/', and restart the server.

[1] *'data/felix' directory for JoramMQ distribution and 'felix-cache' for Joram.*

🛠 **Warning :** Now, the Rest/JMS connector allows to use Jakarta/JMS administered objects. So it needs Jakarta/JMS API bundle and Jakarta/JMS client bundle (see felix.properties configuration file above).

Note: With Joram the configuration is slightly different, an example configuration is delivered with the samples.

1.1.2 Configuration

The JoramMQ JMS Rest implementation does have some configuration options. These are configured via the Felix properties file in the conf directory.

Jetty

The main Jetty properties are listed in the table below (see the Felix HTTP service documentation: https://github.com/apache/felix-dev/tree/master/http#configuration-properties).

Property	Description
org.apache.felix.http.enable	Flag to enable the use of HTTP. The default is true.
org.osgi.service.http.port	The port used for servlets and resources available via HTTP. The default is 8080. See port settings below for additional information. A negative port number has the same effect as setting org.apache.felix.http.enable to false.
org.apache.felix.https.enable	Flag to enable the use of HTTPS. The default is false.
org.osgi.service.http.port.secure	The port used for servlets and resources available via HTTPS. The default is 8443. See port settings below for additional information. A negative port number has the same effect as setting org.apache.felix.https.enable to false.
org.apache.felix.https.keystore	The name of the file containing the keystore.
org.apache.felix.https.keystore.password	The password for the keystore.

You may also have to specify other variables related to the use of Jetty and especially in a SSL context. Below is the format of the configuration file and the default values for each property.

```
################################################################
# Felix HTTP Service
################################################################
org.apache.felix.http.enable=true
org.osgi.service.http.port=8989
#org.apache.felix.https.enable=true
#org.osgi.service.http.port.secure=8443
#org.apache.felix.https.keystore=./conf/keystore
#org.apache.felix.https.keystore.password=jorampass
```

Security

By default all users are allowed to connect to the JoramMQ JMS Rest component, by defining the properties below you can restrain its access:

- Authentication:
  - rest.jms.user: defines the name of the allowed user.
  - rest.jms.password: defines the password of the allowed user.

By setting the rest.jms.ipallowed property you can restrict access from a list of IP addresses. You can use this function by defining a list of IP addresses to which the access will be allowed. Access from IP's which are not in the list will be denied. By default all IP addresses are allowed.

A list is built as a comma-separated list of IP addresses, “a.b.c.d” or “a.b.c.d/m” where m specifies the number of significant bits in the address. For examples:

- rest.jms.ipallowed=192.168.1.0/24,tiga.scalagent.com,aurore.scalagent.com

There is no filtering for access from the server itself (loopback or local IP addresses).

🛠 **Warning :** Currently this mechanism only works with IPv4 address.

JMS Rest interface

The configuration properties of the JoramMQ Rest interface are listed in the table below.

Property	Description
rest.service.name	Name of the service.
rest.jms.connectionFactory	The JNDI name of the JMS ConnectionFactory used to connect to the JMS Provider.
rest.jndi.factory.initial	The value of this property should be the fully qualified class name of the factory class that will create an initial context.
rest.jndi.factory.host	The value of this property specify the hostname of the JNDI service provider the JMS Rest interface will use.
rest.jndi.factory.port	The value of this property specify the listening port of the JNDI service provider the JMS Rest interface will use.
rest.idle.timeout	Each producer or consumer resource needs to be explicitly closed, this parameter allows to set the default idle time in seconds in which the producer/consumer context will be closed if idle. A value less than or equal to zero means no time out, by default 0.
rest.cleaner.period	This parameter determines the interval in seconds the task that checks for timed-out sessions will run at. A value less than or equal to zero means that the task is inactive, by default 15 seconds.

Below is the format of the configuration file and the default values for each property.

```
rest.jms.connectionFactory = cf
rest.jndi.factory.initial = fr.dyade.aaa.jndi2.client.NamingContextFactory
rest.jndi.factory.host = localhost
rest.jndi.factory.port = 17500
rest.idle.timeout = 0
rest.cleaner.period = 60
```

1.1.3 JoramMQ XML administration script

When started the joram-client-jms.jar bundle execute an XML administration script if the joram.adminXML property is defined.

Below is the format of the Felix configuration file:

```
joram.adminXML = ./joramAdmin.xml
```

1.2 JMS Rest Interface basics

The JMS Rest interface publishes a variety of REST resources to perform various tasks on queues or topics. Only the top-level queue and topic URI schemes are published to the outside world. You must discover all over resources to interact with by looking for and traversing links. You'll find published links within custom response headers and embedded in published XML representations. Let's look at how this works.

1.2.1 Destination resources

To begin to interact with a queue or topic you must do a HEAD request on the following relative URI patterns:

- /{service}/jndi/{name} if the destination is registered in the configured JNDI service. If the destination is not registered an error is returned.

The base of the URI is the base URL of the service as defined in the Installation and configuration section of this document, by default 'joram'.

```
HEAD /joram/jndi/myqueue HTTP/1.1
Host: localhost

--- Response ---
HTTP/1.1 200 Ok
lookup: http://localhost:8989/joram/jndi/myqueue
create-consumer: http://localhost:8989/joram/jndi/myqueue/create-consumer
create-consumer-transacted: http://localhost:8989/joram/jndi/myqueue/create-
consumer?session-mode=0
create-consumer-dups-ok: http://localhost:8989/joram/jndi/myqueue/create-consumer?
session-mode=3
create-consumer-client-ack: http://localhost:8989/joram/jndi/myqueue/create-
consumer?session-mode=2
create-producer: http://localhost:8989/joram/jndi/myqueue/create-producer
```

```
create-producer-dups-ok: http://localhost:8989/joram/jndi/myqueue/create-producer?session-mode=3
create-producer-transacted: http://localhost:8989/joram/jndi/myqueue/create-producer?session-mode=0
```

Note: You can use the "curl" utility to test this easily. Simply execute a command like this:

```
curl --head http://localhost:8989/joram/jndi/myqueue
```

The Head response contains a number of custom response headers that are URLs to additional Rest resources that allow you to interact with the destination in different ways. It is important not to rely on the scheme of the URLs returned within these headers as they are an implementation detail. Treat them as opaque and query for them each and every time you initially interact (at boot time) with the server. If you treat all URLs as opaque then you will be isolated from implementation or configuration changes as the Joram Rest interface evolves over time.

Destination resource response header

Below is a list of response headers you should expect when interacting with a Destination resource.

- Lookup: This is the URL used to access the related destination.
- create-consumer: this is the URL to create a simple message consumer for the related destination. The semantics of this link is described in Consumer resources.
- create-consumer-transacted: this is the URL to create a transacted message consumer for the related destination. The semantics of this link is described in Consumer resources.
- create-consumer-dups-ok: this is the URL to create a message consumer allowing message duplication. The semantics of this link is described in Consumer resources.
- create-consumer-client-ack: this is the URL to create a message consumer with client acknowledge QoS. The semantics of this link is described in Consumer resources.
- create-producer: this is the URL to create a simple message producer for the related destination. The semantics of this link is described in Producer resources.
- create-producer-dups-ok: this is the URL to create a simple message producer for the related destination. The semantics of this link is described in Producer resources.
- create-producer-transacted: this is the URL to create a simple message producer for the related destination. The semantics of this link is described in Producer resources.

1.2.2 Producer resources

In the previous section you saw that a destination resource publishes multiple custom headers that are links to other Rest resources. You create producer resources by doing a simple POST to the URL published by the create-producer header. If you want to use a transacted mode and/or modify the default parameters, then you must use the form parameters described below:

- session-mode: sets the QoS mode for the related JMSContext's session, by default AUTO_ACKNOWLEDGE.
- client-id: sets the client identifier for the related JMSContext's connection. If you do not provide this parameter, a client identifier will be automatically generated by the server.
- name: this is the name of the producer. If you do not provide this parameter, a name will be automatically generated by the server.

- idle-timeout: normally each producer resource need to be explicitly closed, this parameter allows to set the idle time in seconds in which the producer context will be closed if idle. If less than or equal to 0, the producer never closes.
- persistent: default QoS of the produced messages. To create persistent messages when you post your messages you can set this parameter to true, by default false.
- priority: default priority of the messages you post through this producer. This query parameter is an integer value between 0 and 9.
- time-to-live: default “time to live” of the messages you pst through this producer. The timetolive query parameter is a time in milliseconds you want the message active.
- delivery-delay: Sets the minimum length of time in milliseconds that must elapse after a message is sent before the JMS provider may deliver the message to a consumer.
- correlation-id: Specifies that messages sent using this JMSProducer will have their JMSCorrelationID header value set to the specified correlation ID.
- *user*: Specifies the user name for the JMS connection. This parameter is now deprecated, you should use a form parameter instead.
- *password*: Specifies the password for the JMS connection. This parameter is now deprecated, you should use a form parameter instead.

🛠 **Warning :** Be careful, now this post request needs a content type set to “application/x-www-form-urlencoded”.

```
POST /joram/jndi/myqueue/create-producer HTTP/1.1
Host: localhost
Content-Type:  application/x-www-form-urlencoded
Content-Length: 33

user=anonymous&password=anonymous

--- Response ---
HTTP/1.1 201 Created
close-context: http://localhost:8989/joram/jms/prod1
send-message: http://localhost:8989/joram/context/prod1
send-next-message: http://localhost:8989/joram/context/prod1/1
```

Note: You can use the "curl" utility to test this easily. Simply execute a command like this:

```
curl -X POST -v 'http://localhost:8989/joram/jndi/myqueue/create-producer' -H 'con
tent-Type: application/x-www-form-urlencoded' --data 'user=anonymous&password=anon
ymous'
```

The Post response itself contains a number of custom response headers that are URLs to additional Rest resources that allow you to send messages to the destination.

Producer resource response header

Below is a list of response headers you should expect when interacting with a Producer resource.

- close-context: this is the URL to close the producer context.

- send-message: the URL to send a new message to the destination. The semantics of this link are described in Producing messages.
- send-next-message: the URL template you can use to send a message to the destination, this URL is uniquely generated for each message and used for duplicate detection. The semantics of this link are described in Producing messages.

Messages are sent to a destination by sending a simple HTTP message to the URL published by the send-message header. The HTTP message contains whatever content you want to publish to the JMS destination. Here's an example scenario:

```
POST /joram/context/prod1/1 HTTP/1.1
Host: localhost
Content-Type: text/plain

Message test 1

--- Response ---
HTTP/1.1 201 Created
send-message: http://localhost:8989/joram/context/prod1
send-next-message: http://localhost:8989/joram/context/prod1/2
```

Note: You can use the "curl" utility to test this easily. Simply execute a command like this:

```
curl -XPost –data “Message test 1” http://localhost:8989/joram/context/prod1/1 --
header "Content-Type:text/plain"
```

The Post response itself contains a number of custom response headers that are URLs that allow you to send more messages to the destination.

1.2.3 Consumer resources

In the section 1.2.1 you saw that a destination resource publishes multiple custom headers that are links to other Rest resources. Doing an empty POST to one of these URLs will create a consumer resource that follows an auto-acknowledge protocol and, if you are interacting with a topic, creates a temporarily subscription to the topic. If you want to use the acknowledgement protocol and/or create a durable subscription (topics only), then you must use the form parameters described below:

- session-mode: sets the QoS mode for the related JMSContext's session, by default AUTO_ACKNOWLEDGE.
- client-id: sets the client identifier for the related JMSContext's connection. If you do not provide this parameter, a client identifier will be automatically generated by the server.
- name: this is the name of the consumer. If you do not provide this parameter, a name will be automatically generated by the server.
- idle-timeout: normally each consumer resource need to be explicitly closed, this parameter allows to set the idle time in seconds in which the consumer context will be closed if idle. If less than or equal to 0, the producer never closes.

- durable: Only available on topics, this specifies whether you want a durable subscription or not. A value of true or false can be given, by default false.
- selector: This is an optional JMS selector string.
- timeout: default timeout for consumer's queries.
- shared: true for shared.
- sub-name: the name used to identify this subscription.
- user: specifies the user name for the JMS connection.
- password: specifies the password for the JMS connection.

🛠 **Warning :** Be careful, now this post request needs a content type set to “application/x-www-form-urlencoded”.

```
POST /joram/jndi/myqueue/create-consumer HTTP/1.1
Host: localhost
Content-Type:  application/x-www-form-urlencoded
Content-Length: 33

user=anonymous&password=anonymous

--- Response ---
HTTP/1.1 201 Created
close-context: http://localhost:8989/joram/jms/cons1
receive-message: http://localhost:8989/joram/context/cons1
receive-next-message: http://localhost:8989/joram/context/cons1/1
```

Note: You can use the "curl" utility to test this easily. Simply execute a command like this:

```
curl -X POST -v 'http://localhost:8989/joram/jndi/myqueue/create-consumer' -H 'con
tent-Type: application/x-www-form-urlencoded' --data 'user=anonymous&password=anon
ymous'
```

The Post response itself contains a number of custom response headers that are URLs to additional Rest resources that allow you to receive messages from the destination.

Consumer resource response header

Below is a list of response headers you should expect when interacting with a Consumer resource.

- close-context: this is the URL to close the consumer context.
- receive-message: the URL to receive a new message from the destination. The semantics of this link are described in Consuming messages.
- receive-next-message: the URL template you can use to receive a message from the destination, this URL is uniquely generated for each message and used to prevent the loss of messages. The semantics of this link are described in Consuming messages.

Messages are received from a destination by sending a simple HTTP message to the URL published by the receive-message header. Here's an example scenario:

```
GET /joram/context/cons1/1 HTTP/1.1
Host: localhost

--- Response ---
HTTP/1.1 200 Ok
Content-Type: text/plain

acknowledge: http://localhost:8989/joram/context/cons1/acknowledge
acknowledge-message: http://localhost:8989/joram/context/cons1/1
receive-message: http://localhost:8989/joram/context/cons1
receive-next-message: http://localhost:8989/joram/context/cons1/2

Message test 1
```

Note: You can use the "curl" utility to test this easily. Simply execute a command like this:

```
curl -XGet http://localhost:8989/joram/context/cons1/1
```

The Post response itself contains a number of custom response headers that are URLs that allow you to consume more messages from the destination.

1.2.4 API discovery

The Joram JMS/Rest API could be easily discovered through a simple browser accessing the base URL of the service: “http:/host:port/{service}”. This web page delivers you three link allowing to discover the service resources and their uses.

1.3 Producing messages

This section discusses the protocol for sending messages to a destination.

1.3.1 Synchronization warning

Be careful, a producer is a single-threaded context producing messages, according to the JMS specification a producer can not be operated by more than one thread at a time.

1.3.2 Message Type and QoS

By default the JoramMQ JMS API allows to send JMS TextMessage (see section 1.3.6).

Persistent messages

By default, posted messages are not persistent and will not be persisted in Joram server. You can create persistent messages by modifying the default configuration as expressed in section 1.2.2 so that all messages sent are persistent.

Alternatively, you can set a URL query parameter called delivery-mode[2] when you post your messages using the URLs returned in the send-message or send-next-message.

```
POST /joram/context/prod1/send-message?delivery-mode=1
Host: localhost
Content-Type: application/xml

<body>
   <field1>value1</field1>
   <field2>value2</field2>
</body>
```

Time to live and Priority

You can set the time to live, and/or the priority of the message in the destination by setting an additional query parameter. The time-to-live query parameter is a time in milliseconds you want the message active. The priority is another query parameter with an integer value between 0 and 9 expressing the priority of the message.

```
POST /joram/context/prod1/send-message?time-to-live=15000&priority=2
Host: localhost
Content-Type: application/xml

<body>
   <field1>value1</field1>
   <field2>value2</field2>
</body>
```

1.3.3 Duplicate detection

You might have network problems when posting new messages to a destination. You may do a Post and never receive a response. Unfortunately, you don't know whether or not the server received the message and so reposting the message might cause a duplicate message to be posted to the destination. By default, the Joram Rest interface is configured to avoid duplicate messages. The initial Post to the create-producer URL returns a send-next-message URL to POST to. This URL will be uniquely generated for each message and used for duplicate generation, so if you don't get response to the Post you can post anew.

If you lose the send-next-message unique URL, then just go back to producer resource to get the send-next-message URL again.

1.3.4 Transacted mode

You can use a local transaction to group message sends (See JMS API). If any one of the operations fails, the transaction can be rolled back, and the operations can be attempted again from the beginning. If all the operations succeed, the transaction can be committed.

The create-producer-transacted URL allows the creation of a producer in a transacted session.

```
POST: http://localhost:8989/joram/jndi/myqueue/create-producer
```

[2] *Values of delivery mode are defined in JMS API: interface javax.jms.DeliveryMode.*

```
links:
     close-context: http://localhost:8989/joram/jms/prodTransacted
     commit: http://localhost:8989/joram/context/prodTransacted/commit
     rollback: http://localhost:8989/joram/context/prodTransacted/rollback
     send-message: http://localhost:8989/joram/context/prodTransacted
     send-next-message: http://localhost:8989/joram/context/prodTransacted/1
```

The API provides commit and rollback URLs (see example above) that can be invoked through a HEAD. A transaction commit means that all produced messages are sent, a transaction rollback means that all produced messages are destroyed.

1.3.5 Simple example

Create a producer on the destination registered as “myqueue” in JNDI.

```
POST: http://localhost:8989/joram/jndi/myqueue/create-producer
links:
    send-message: http://localhost:8989/joram/context/prod1
    send-message-next: http://localhost:8989/joram/context/prod1/1
    close-context: http://localhost:8989/joram/jms/prod1
```

Send a message with send-next-message link.

```
POST: http://localhost:8989/joram/context/prod1/1
links:
    send-message: http://localhost:8989/joram/context/prod1
    send-next-message: http://localhost:8989/joram/context/prod1/2
```

Send a second message with send-next-message link.

```
POST: http://localhost:8989/joram/context/prod1/2
links:
    send-message: http://localhost:8989/joram/context/prod1
    send-next-message: http://localhost:8989/joram/context/prod1/3
```

Delete the producer.

```
DELETE: http://localhost:8989/joram/jms/prod1
links:
    jndi: http://localhost:8989/joram/jndi
```

1.3.6 Advanced usage

By default the JoramMQ JMS Rest API handles JMS TextMessage without any additional property.

Using a JSON content the API allows to specify the message type and additional properties, currently TextMessage, MapMessage and BytesMessages are supported. The related JSON document should be an object with the following property:

- *type*: the JMS class name of the desired message (“TextMessage”, “BytesMessage” or “MapMessage”).
- *header: an object containing one or several JMS properties: DeliveryMode, DeliveryTime, Priority, Expiration and CorrelationID.*

- *properties*: an object containing all desired properties. By default, values are string, number (double) or boolean, however the real type of each property can be fixed using the following syntax for a pair:
  - name, [value, type]
- *body*: an object corresponding to the content of the message, a string for a TextMessage, an array for a BytesMessage and a collection of key / value pairs for a MapMessage.

Example

Below the JSON content describing a BytesMessage with 2 properties, a String p1 and an integer p2.

```
{
  "type":"BytesMessage",
  "header":{
    "CorrelationID":" … "
  }
  "properties":{
    "p1":["value1","java.lang.String"],
    "p2":["2","java.lang.Integer"]
  }
  "body":[109,121,32,116,101,115,116,32,109,101,115,115,97,103,101]
}
```

1.4 Consuming messages

This section discusses the protocol for receiving messages to a destination.

1.4.1 Synchronization warning

A consumer is a single-threaded context consuming messages, according to the JMS specification a consumer can not be operated by more than one thread at a time.

1.4.2 Message Type

By default the JoramMQ Rest/JMS API allows to receive JMS TextMessage (see section 1.4.10).

1.4.3 Acknowledgement mode

By default the consumer is created in AUTO_ACKNOWLEDGE mode. With this mode, the context's session automatically acknowledges the receipt of a message when the session has successfully returned from a call to receive.

The create-consumer-dups-ok URL allows the creation of a consumer in a DUPS_OK_ACKNOWLEDGE mode. This acknowledgment mode instructs the session to lazily acknowledge the delivery of messages.

The create-consumer-client-ack URL allows the creation of a consumer in CLIENT_ACKNOWLEDGE mode. With this mode, the client must acknowledge a consumed message by doing a post to the corresponding acknowledge-message URL. The acknowledge URL permits to acknowledge all messages previously delivered to the consumer.

1.4.4 Transacted mode

You can use a local transaction to group message receives (See JMS API). If any one of the operations fails, the transaction can be rolled back, and the operations can be attempted again from the beginning. If all the operations succeed, the transaction can be committed.

The create-consumer-transacted URL allows the creation of a consumer in a transacted session.

```
POST: http://localhost:8989/joram/jndi/myqueue/create-consumer
links:
     close-context: http://localhost:8989/joram/jms/consTransacted
     commit: http://localhost:8989/joram/context/consTransacted/commit
     rollback: http://localhost:8989/joram/context/consTransacted/rollback
     receive-message: http://localhost:8989/joram/context/consTransacted
     receive-next-message: http://localhost:8989/joram/context/consTransacted/1
```

The API provides commit and rollback URLs (see example above) that can be invoked through a HEAD. A transaction commit means that all consumed messages are acknowledged, a transaction rollback means that all consumed messages are recovered and redelivered unless they have expired.

1.4.5 Durable subscription

By default, when the destination is a topic the consumer is created on a temporary subscription. To create a durable subscription you can set a URL query parameter called durable to true when you post to the create-consumer URL.

```
POST /joram/jndi/mytopic/create-consumer?durable=true&name=sub1
Host: localhost

--- Response ---
HTTP/1.1 201 Created
close-context: http://localhost:8989/joram/jms/sub1
receive-message: http://localhost:8989/joram/context/sub1
receive-next-message: http://localhost:8989/joram/context/sub1/1
```

1.4.6 Selector

You can create a message consumer using a message selector[3] by setting a query parameter named selector. A message selector allows the client to restrict the messages delivered to the consumer to those that match the selector. Selectors are defined using SQL 92 syntax and typically apply to message headers and properties, but can not use the message content.

The example below create a durable subscriber on the topic named “mytopic” with a message selector about index and p1 properties: “(index > 10) OR (index < 4) AND (p1='value1')”

```
POST /joram/jndi/mytopic/create-consumer?durable=true&selector=%28index+
%3E+10%29+OR+%28index+%3C+4%29+AND+%28p1%3D%27value1%27%29&name=sub1
Host: localhost
```

[3] *only messages with properties matching the message selector expression are delivered.*

```
--- Response ---
HTTP/1.1 201 Created
close-context: http://localhost:8989/joram/jms/sub1
receive-message: http://localhost:8989/joram/context/sub1
receive-next-message: http://localhost:8989/joram/context/sub1/1
```

1.4.7 Receiving messages

After you have created a consumer resource, you are ready to start pulling messages from the broker. The response to the consumer creation contains receive-message and receive-next-message response headers. Get to the URL contained within this header to consume the next message in the queue or topic subscription. A successful Get causes the broker to extract a message from the queue or topic subscription, acknowledge it[4], and return it to the consuming client. If there are no messages in the queue or topic subscription a 204 (No Content) HTTP code is returned.

For each request you can specify an optional query parameter called timeout. This parameter is the time in milliseconds the request blocks until a message arrives. A timeout of -1 never expires, and the call blocks indefinitely. With a timeout of zero the request returns the next message if one is immediately available.

Note: For both successful and unsuccessful posts to the receive-message and/or receive-next-message URL, the response will contain new receive-message and receive-next-message headers. You must always use these new URLs to consume new messages.

1.4.8 Recovering from network failures

If you experience a network failure and there is no response to your post to a receive-next-message URL, just re-do your Post. A Post to a receive-next-message URL is idempotent, meaning that it will return the same result until you acknowledge the corresponding message.

The consumer resource caches the non-acknowledged messages so if there is a network failure you can do a re-post and the cached message will be returned. This is the reason why the protocol always requires you to use the next new receive-next-message URL returned with each response.

1.4.9 Simple example

Create a consumer on the destination registered as “myqueue” in JNDI, using the create-consumer-client-ack link.

```
POST: http://localhost:8989/joram/jndi/myqueue/create-consumer&session-mode=2
--- Response ---
links:
    close-context: http://localhost:8989/joram/jms/cons1
    receive-message: http://localhost:8989/joram/context/cons1
    receive-next-message: http://localhost:8989/joram/context/cons1/1
```

Receive a first message using receive-next-message link.

```
GET: http://localhost:8989/joram/context/cons1/1
--- Response ---
links:
    acknowledge: http://localhost:8989/joram/context/cons1
```

[4] *In auto-acknowledge mode.*

```
    acknowledge-message: http://localhost:8989/joram/context/cons1/1
    receive-message: http://localhost:8989/joram/context/cons1
    receive-next-message: http://localhost:8989/joram/context/cons1/2
```

Receive a second message using receive-next-message link.

```
GET: http://localhost:8989/joram/context/cons1/2
--- Response ---
links:
    acknowledge: http://localhost:8989/joram/context/cons1
    acknowledge-message: http://localhost:8989/joram/context/cons1/2
    receive-message: http://localhost:8989/joram/context/cons1
    receive-next-message: http://localhost:8989/joram/context/cons1/3
```

Acknowledges the first message using the corresponding acknowledge-message link.

```
DELETE: http://localhost:8989/joram/context/cons1/1
--- Response ---
links:
    receive-message: http://localhost:8989/joram/context/cons1
    receive-next-message: http://localhost:8989/joram/context/cons1/3
```

Acknowledges the second message using the corresponding acknowledge-message link.

```
DELETE: http://localhost:8989/joram/context/cons1/2
--- Response ---
links:
    receive-message: http://localhost:8989/joram/context/cons1
    receive-next-message: http://localhost:8989/joram/context/cons1/3
```

Delete the consumer.

```
DELETE: http://localhost:8989/joram/jms/cons1
--- Response ---
links:
    jndi: http://localhost:8989/joram/jndi
    jms: http://localhost:8989/joram/jms
```

1.4.10 Advanced usage

By default the JoramMQ JMS Rest API handles JMS TextMessage without any additional property.

Requesting a JSON content allows to receive messages of different types with the additional properties associated. Currently only TextMessage, MapMessage and BytesMessages are supported. The JSON document returned by the operation will be an object with the following property:

- *type*: the JMS class name of the returned message (“TextMessage”, “BytesMessage” or “MapMessage”).
- *header: an object containing the following JMS properties: DeliveryMode, Priority, Redelivered, Timestamp, Expiration, CorrelationID, CorrelationIDAsBytes, Destination, MessageID, ReplyTo and Type.*
- *properties*: an object containing the properties of the message. Each value is a JSON array containing the value and the real type of this value.

- *body*: an object describing the content of the message, a string for a TextMessage, an array for a BytesMessage and a collection for a MapMessage.

Example

Below the JSON content describing a returned BytesMessage with 3 properties, the integer JMSXDeliveryCount, a String p1 and an integer p2.

```
{
  "type":"BytesMessage",
  "header":{
    "MessageID":" … ",
    "CorrelationID":" … "
     …
  }
  "properties":{
    "JMSXDeliveryCount"=[1, "java.lang.Integer"],
    "p1":["value1","java.lang.String"],
    "p2":["2","java.lang.Integer"]
  }
  "body":[109,121,32,116,101,115,116,32,109,101,115,115,97,103,101]
}
```

1.5 Recovering from client or server crashes

If the server crashes the consumer resources are cleaned. If you do a Post to the send-next-message or receive-next-message URLs, the server will return an error. This is telling you that the URL you are using is no longer defined on the server.

If the client crashes there are multiple ways you can recover. If you have remembered the last send-next-message or receive-next-message link, you can just re-POST to it. If you have remembered the producer or consumer resource URL, you can do a request to obtain a new link URLs.

1.6 Use with an HTML browser

Since the protocol relies on a standard protocol (HTTP) and format (HTML/JSON), the JMS Rest interface can be even accessed from within the browser with a simple URL. The following URLs give a human readable description of the interface:

- http://host:port/{service}
- http://host:port/{service}/jndi
- http://host:port/{service}/jms
- http://host:port/{service}/context

The base of the URI is the base URL of the service as defined in the Installation and configuration section of this document, by default 'joram'.

1.7 Java sample

The code below creates a producer and a consumer on a queue registered in JNDI with the name “myqueue”. Then a simple message is sent through the producer, and read through the consumer.

1.7.1 Code

Initialisation

```
ClientConfig config = new ClientConfig();
Client client = ClientBuilder.newClient(config);

URI base = UriBuilder.fromUri("http://localhost:8989/joram/").build();
WebTarget target = client.target(base);
```

Lookup for the destination registered in JNDI and get the URIs to create producer and consumer.

```
Builder builder = target.path("jndi").path("myqueue").request();
Response response = builder.accept(MediaType.TEXT_PLAIN).head();

URI uriCreateProd = client.target(response.getLink("create-producer"))
                          .queryParam("name", "prod1")
                          .getUri();

URI uriCreateCons = client.target(response.getLink("create-consumer"))
                          .queryParam("name", "cons1")
                          .getUri();
```

Create the producer and get links to send a message and close this producer (be careful to use the right content type).

```
response = client.target(uriCreateProd)
    .request()
    .accept(MediaType.TEXT_PLAIN)
    .post(Entity.entity(null, MediaType.APPLICATION_FORM_URLENCODED));

System.out.println("== create-producer = " + response.getStatus());
print(response.getLinks());

URI uriCloseProd = response.getLink("close-context").getUri();
URI uriSendNextMsg = response.getLink("send-next-message").getUri();
```

Then send a message

```
response = client.target(uriSendNextMsg)
                 .request()
                 .accept(MediaType.TEXT_PLAIN)
                 .post(Entity.entity("Test message.", MediaType.TEXT_PLAIN));

System.out.println("== send-next-message = " + response.getStatus());
print(response.getLinks());
```

Create the consumer and get links to receive a message and close this consumer (be careful to use the right content type).

```
response = client.target(uriCreateCons)
     .request()
     .accept(MediaType.TEXT_PLAIN)
     .post(Entity.entity(null, MediaType.APPLICATION_FORM_URLENCODED));

System.out.println("== create-consumer = " + response.getStatus());
print(response.getLinks());

URI uriCloseCons = response.getLink("close-context").getUri();
URI uriReceiveNextMsg = response.getLink("receive-next-message").getUri();
```

Then receive the message.

```
response = client.target(uriReceiveNextMsg)
                 .request()
                 .accept(MediaType.TEXT_PLAIN)
                 .get();

String msg = response.readEntity(String.class);
if (response.getStatus() == Response.Status.OK.getStatusCode() && msg != null) {
  System.out.println("== receive-next-message = " + response.getStatus() +
                     ", msg = " + msg);
} else {
  System.out.println("ERROR consume msg = " + msg + ", response = " + response);
}
```

Close the producer.

```
response = client.target(uriCloseProd)
                 .request()
                 .accept(MediaType.TEXT_PLAIN)
                 .delete();

System.out.println("== close-producer = " + response.getStatus());
print(response.getLinks());
```

Close the consumer.

```
response = client.target(uriCloseCons)
                 .request()
                 .accept(MediaType.TEXT_PLAIN)
                 .delete();

System.out.println("== close-consumer = " + response.getStatus());
print(response.getLinks());
```

1.7.2 Execution traces

```
== lookup "myQueue" = 201
```

```
link :
    create-consumer : http://localhost:8989/joram/jndi/queue/create-consumer
    lookup : http://localhost:8989/joram/jndi/queue
    create-producer-transacted : http://localhost:8989/joram/jndi/queue/create-
producer?session-mode=0
    create-consumer-client-ack : http://localhost:8989/joram/jndi/queue/create-
consumer?session-mode=2
    create-consumer-dups-ok : http://localhost:8989/joram/jndi/queue/create-
consumer?session-mode=3
    create-consumer-transacted : http://localhost:8989/joram/jndi/queue/create-
consumer?session-mode=0
    create-producer : http://localhost:8989/joram/jndi/queue/create-producer
    create-producer-dups-ok : http://localhost:8989/joram/jndi/queue/create-
producer?session-mode=3

== create-producer = 201
link :
    send-next-message : http://localhost:8989/joram/context/prod100/1
    send-message : http://localhost:8989/joram/context/prod100
    close-context : http://localhost:8989/joram/jms/prod100

== send-next-message = 200
link :
    send-next-message : http://localhost:8989/joram/context/prod100/2
    send-message : http://localhost:8989/joram/context/prod100

== create-consumer = 201
link :
    receive-message : http://localhost:8989/joram/context/cons102
    receive-next-message : http://localhost:8989/joram/context/cons102/1
    close-context : http://localhost:8989/joram/jms/cons102

== receive-next-message = 200, msg = Test message.
link :
    receive-message : http://localhost:8989/joram/context/cons102
    receive-next-message : http://localhost:8989/joram/context/cons102/2

== close-producer = 200
link :
    jndi : http://localhost:8989/joram/jndi
    jms : http://localhost:8989/joram/jms

== close-consumer = 200
link :
    jndi : http://localhost:8989/joram/jndi
    jms : http://localhost:8989/joram/jms
```

1.8 Java advanced samples

Initialization.

```
ClientConfig config = new ClientConfig();
Client client = ClientBuilder.newClient(config);
WebTarget target = client.target(getBaseURI());

URI request;
Response response;
```

Create the producer, and get the get links to send a message and close this producer.

```
request = target.path("jms")
                .path("queue")
                .path("myQueue1")
                .path("create-producer")
                .queryParam("name", "prod1").getUri();

response = client.target(request)
    .request()
    .accept(MediaType.TEXT_PLAIN, MediaType.APPLICATION_JSON)
    .post(Entity.entity(null, MediaType.APPLICATION_FORM_URLENCODED));

URI send = response.getLink("send-next-message").getUri();
URI close = response.getLink("close-context").getUri();
```

Builds the JSON content corresponding to a BytesMessage.

```
HashMap<String, Object> msg = new HashMap<>();

msg.put("type", "BytesMessage");

HashMap<String, Object> header = new HashMap<>();
header.put("CorrelationID", msgid);
header.put("ReplyTo", reply);
msg.put("header", header);

HashMap<String, Object> props = new HashMap<>();
props.put("p1", new String[]{"value1", String.class.getName()});
props.put("p2", new String[]{"2", Integer.class.getName()});
msg.put("properties", props);

byte[] body = "the body of my message".getBytes();
msg.put("body", body);

Gson gson = new GsonBuilder().create();
String json_msg = gson.toJson(msg);
```

Builds the JSON content corresponding to a MapMessage

```
HashMap<String, Object> msg = new HashMap<>();
```

```
    msg.put("type", "MapMessage");

    HashMap<String, Object> header = new HashMap<>();
    header.put("CorrelationID", msgid);
    header.put("ReplyTo", reply);
    msg.put("header", header);

    HashMap<String, Object> props = new HashMap<>();
    props.put("p1", new String[]{"value1", String.class.getName()});
    props.put("p2", new String[]{"2", Integer.class.getName()});
    msg.put("properties", props);

    HashMap map = new HashMap();
    map.put("key1", "my test message");
    map.put("key2", new String[]{"2", Integer.class.getName()});
    msg.put("body", map);

    Gson gson = new GsonBuilder().create();
    String json_msg = gson.toJson(msg);
```

Sends the message (either BytesMessage or MapMessage above), then close the producer.

```
    // Send next message
    response = client.target(send)
                     .request()
                     .accept(MediaType.TEXT_PLAIN)
                     .post(Entity.entity(json_msg, MediaType.APPLICATION_JSON));
```

Create the consumer, and get the get links to receive a message and close this consumer.

```
    request = target.path("jms")
                    .path("queue")
                    .path("myQueue1")
                    .path("create-consumer")
                    .queryParam("name", "cons1").getUri();

    response = client.target(request)
        .request()
        .accept(MediaType.TEXT_PLAIN)
        .post(Entity.entity(null, MediaType.APPLICATION_FORM_URLENCODED));

    URI receive = response.getLink("receive-next-message").getUri();
    URI close = response.getLink("close-context").getUri();
```

Receive a message.

```
    response = client.target(receive)
                     .request()
                     .accept(MediaType.APPLICATION_JSON)
                     .get();
```

```
    json_msg = response.readEntity(String.class);

    if ((response.getStatus() == Response.Status.OK.getStatusCode()) &&
        (json_msg != null)) {
      HashMap<String, Object> msg = gson.fromJson(json_msg, HashMap.class);
      // Reads the message depending of its JMS type.
      String type = msg.get("type");
      Map header = (Map) msg.get("header");

      Map properties = (Map) msg.get("properties");

        if ("BytesMessage".equals(type)) {
          byte[] body = gson.fromJson(msg.get("body").toString(), byte[].class)
        } else if ("MapMessage".equals(type)) {
          Map body = (Map) msg.get("body");
        } else if ("TextMessage".equals(type)) {
          String body = (String) msg.get("body");
        } else {
          throw new Exception("Error receiving message");
        }
    } else {
      throw new Exception("Error receiving message");
    }
```

Closes the producer or the consumer using the URI returned by the creation request.

```
    response = client.target(close)
                     .request()
                     .accept(MediaType.TEXT_PLAIN)
                     .delete();
```

1.9 Using Joram samples

1.9.1 Rest sample

1. Shell#1 – Go to the 'samples' directory:
   cd joram/samples/src/joram
2. Shell#1 – Compile the samples:
   ant clean compile
3. Shell#1 – Launch the Joram server with the JMS/Rest API
   ant reset rest_server
4. Shell#2 – Go to the 'samples' directory:
   cd joram/samples/src/joram
5. Shell#2 – Sends 10 messages using the JMS/Rest API:
   ant rest.producer
6. Shell#2 – Receives 10 messages using the JMS/Rest API:
   ant rest.consumer

1.9.2 Rest bridge sample

1. Shell#1 – Go to the 'samples' directory:
   cd joram/samples/src/joram
2. Shell#1 – Compile the samples:
   ant clean compile
3. Shell#1 – Launch the Joram 'foreign' server with the JMS/Rest API
   ant reset rest_server
4. Shell#2 – Go to the 'samples' directory:
   cd joram/samples/src/joram
5. Shell#2 – Launch the Joram 'bridge' server with JMS/Rest bridge activated
   ant restbridge_server
6. Shell#3 – Runs the administration. It creates 2 queues on the Joram 'bridge' server: a distribution queue that forward messages to the queue 'queue' on Joram 'foreign' server, and an acquisition queue that acquires messages from the queue 'queue' on Joram 'foreign' server.
   ant restbridge_admin
7. Shell#3 – Sends a message to the distribution queue on the Joram 'bridge' server. This queue forwards each message to the queue 'queue' on Joram 'foreign' server through the JMS/Rest API.
   ant restbridge_producer
8. Shell#3 – Receives a message from the acquisition queue on the Joram 'bridge' server. This queue acquires each message from the queue 'queue' on Joram 'foreign' server through the JMS/Rest API.
   ant restbridge_consumer

2 JoramMQ administration Rest API

The JoramMQ Administration Rest API allows you to use several administration features of JoramMQ over a simple Rest/HTTP interface. For example this allows any web capable device to create User or Destination using a regular HTTP Post or Get request.

2.1 Installation and configuration

2.1.1 Installation

The JoramMQ administration Rest interface is installed as an OSGi bundle (joram-tools-rest-admin.jar in the bundle directory), it depends on the HTTP service and JAX-RS service implemented by the Jetty and Jersey bundles. You need to modify the default configuration as follows: (adding the lines colored in red)

```
felix.auto.start.1= \
 file:./bundle/monolog.jar \
 …
 file:./bundle/ow2-jms-2.0-spec.jar \
 file:./bundle/jakarta.jms-api.jar \
 file:./bundle/ow2-jta-1.1-spec.jar \
 …
 file:./bundle/org.osgi.compendium.jar \
 file:./bundle/gson.jar \
 file:./bundle/org.apache.felix.http.jetty.jar \
 file:./bundle/jndi-client.jar \
 file:./bundle/joram-client-jms.jar \
 file:./bundle/joram-jakarta-jms.jar \
 file:./bundle/joram-tools-rest-admin.jar
```

Then you need to stop the server, delete Felix bundle cache 'data/felix/', and restart the server.

2.1.2 Configuration

The JoramMQ Administration Rest implementation does have some configuration options. These are configured via the Felix properties file in the conf directory :

- Jetty configuration (see section 1.1.2).
- Authentication: by default all users are allowed to connect to the JoramMQ administration Rest component, by defining the properties below you can restrain its access:
  - rest.admin.root: defines the name of the allowed user.
  - rest.admin.password: defines the password of the allowed user.
- By setting the rest.admin.ipallowed property you can restrict access from a list of IP addresses (see section 1.1.2).
- Credentials for JoramMQ administration user. If these properties are not defined the default credentials from ConnectionFactory[5] are used.

[5] *These defaults can be modified using JoramDfltRootLogin and JoramDfltRootPassword Java properties.*

- rest.admin.jms.user: defines the name of the administration user of JoramMQ/JMS.
- rest.admin.jms.password: defines the password of the administration user of JoramMQ/JMS.

2.2 Usage

The JoramMQ Rest administration interface publishes a variety of REST resources to perform various tasks on users, queues or topics.

Authentication

If the JoramMQ administration API is configured to require authentication (see section 2.1.2) you need to add an authorization header in the request. This property is an encoded string built from the user's name and password separated by a colon ':'.

For example to get the list of existing queues with a curl request:

```
curl --user "admin:admin" --get http://192.168.1.36:8989/joram/admin/queue
```

2.2.1 Creating a destination

You create a destination resources by doing a simple POST on the following relative URI patterns:

- /joram/admin/[queue|topic]/{name}

To modify the default parameters you must use the form parameters described below:

- server-id: unique identifier of location server. If not set the destination is created on the “local” server.
- jndi-name: the symbolic name to register the destination in JNDI context.
- jndi-bind: boolean requesting to rebind the destination in JNDI (by default true if jndi-name is defined). If true and jndi-name is not defined, *name* is used.
- free-readind: Grants the read right to all users on this destination.
- free-writing: Grants the write right to all users on this destination.
- class-name: Defines the implementation class of the destination (by default Queue or Topic class).

A set of additional properties can be send as a JSON collection, this is especially useful in the case of the creation of specialized destinations.

2.2.2 Deleting a destination

You delete a destination resources by doing a simple DELETE on the following relative URI patterns:

- /joram/admin/[queue|topic]/{name}

To modify the default parameters you must use the form parameters described below:

- server-id: unique identifier of location server, if not set the destination is searched on the “local” server.
- jndi-unbind: boolean requesting to unbind the destination in JNDI (by default true if jndi-name is defined). If true and jndi-name is not defined, *name* is used.
- jndi-name: the symbolic name of the destination in JNDI.

2.2.3 Creating User

You create a user resources by doing a simple POST on the following relative URI patterns:

- /joram/admin/user/{name}

To modify the default parameters you must use the form parameters described below:

- password: The user password (required).
- identity-class-name: Defines the implementation class of the user identity (by default the class org.objectweb.joram.shared.security.SimpleIdentity).
- server-id: unique identifier of location server. If not set the user is created on the “local” server.

A set of additional properties can be send as a JSON collection.

2.2.4 Deleting a user

You delete a user resources by doing a simple DELETE on the following relative URI patterns:

- /joram/admin/user/{name}

To modify the default parameters you must use the form parameters described below:

- server-id: unique identifier of location server, if not set the user is searched on the “local” server.
- password: The user password

2.2.5 Creating a ConnectionFactory

You can create a ConnectionFactory resources by doing a simple GET on the following relative URI patterns:

- `/joram/admin/[local|tcp]/create`

To modify the default parameters you must use the form parameters described below:

- jndi-name: registered name of the ConnectionFactory in JNDI.
- host: DNS name or IP of the server's host.
- port: port number of the listening socket.
- reliable-class: implementation class of the protocol, by default:
  - org.objectweb.joram.client.jms.tcp.ReliableTcpClient.

2.3 Use with an HTML browser

Since the protocol relies on a standard protocol (HTTP) and format (HTML/JSON), the JoramMQ administration Rest interface can be even accessed from within the browser with a simple URL. The following URL gives a human readable description of the interface:

- http://host:port/joram/admin

2.3.1 Creating a destination

You can simply create a destination resources by doing a simple Get request through your HTML browser on the following relative URI pattern:

- /joram/admin/[queue|topic]/{name}

2.3.2 Creating a user

You can simply create an user resources by doing a simple Get request through your HTML browser on the following relative URI pattern:

- /joram/admin/user/{name}

To modify the default parameters you must use the form parameters described below:

- password: The user password.

2.3.3 Destinations and users discovery

The following URLs allows to get a list of queues, topics or users:

- http://host:port/joram/admin/queue
- http://host:port/joram/admin/topic
- http://host:port/joram/admin/user

2.4 Java sample

The code below registers a TcpConnectionFactory, then it creates a queue and a user, then it deletes them.

2.4.1 Code

Initialization

```
ClientConfig config = new ClientConfig();
Client client = ClientBuilder.newClient(config);
WebTarget target = client.target(getBaseURI());
```

Create a TcpConnectionFactory and register it in JNDI

```
URI uri = target.path("tcp").path("create")
            .queryParam("jndi-name", "cf")
            .queryParam("host", "localhost")
            .queryParam("port", "17600").getUri();
response = client.target(uri).request().accept(MediaType.TEXT_PLAIN).get();
```

Create a queue and register it in JNDI

```
uri = target.path("queue").path("queue1")
            .queryParam("free-reader", "true")
            .queryParam("jndi-name", "queue1").getUri();
response = client.target(uri)
                .request()
                .accept(MediaType.TEXT_PLAIN, MediaType.APPLICATION_JSON)
                .post(Entity.entity(
                        "{ \"threshold\": \"3\" }\n",
                        MediaType.APPLICATION_JSON));
```

Create a user

```
uri = target.path("user").path("user1").queryParam("password", "pass").getUri();
```

```
response = client.target(uri)
                 .request()
                 .accept(MediaType.TEXT_PLAIN)
                 .post(null);
```

Delete the queue and unbind it from JNDI

```
uri = target.path("queue").path("queue1")
            .queryParam("jndi-name", "queue1").getUri();
response = client.target(uri).request().accept(MediaType.TEXT_PLAIN).delete();
```

Delete the user

```
uri = target.path("user").path("user1").queryParam("password", "pass").getUri();
response = client.target(uri).request().accept(MediaType.TEXT_PLAIN).delete();
```

Authentication

If the JoramMQ administration API is configured to require authentication (see section 2.2) you need to add an authorization header in the request. The code below shows a simple example:

```
String encoded = DatatypeConverter.printBase64Binary("admin:admin".getBytes());
uri = target.path( … ).getUri();
response = client.target(uri)
                 .request().header("Authorization", encoded)
                 .accept(MediaType.APPLICATION_JSON).get();
```

3 JoramMQ JMX Rest API

The JoramMQ JMX Rest API allows you to expose the JMX Mbeans of the JoramMQ server over a simple Rest/HTTP interface. For example this allows any web capable device to get Mbean's attributes using a regular HTTP request.

3.1 Installation and configuration

The JoramMQ JMX Rest interface is normally declared in the default configuration.

3.1.1 Installation

The JoramMQ JMX Rest interface is installed as an OSGi bundle (joram-tools-rest-jmx.jar in the bundle directory), it depends on the HTTP service and JAX-RS service implemented by the Jetty and Jersey bundles. If needed you can add these bundles to the list of installed bundles as follows:

```
felix.auto.start.1= \
 file:./bundle/monolog.jar \
 …
 file:./bundle/org.osgi.compendium.jar \
 file:./bundle/gson.jar \
 file:./bundle/org.apache.felix.http.jetty.jar \
 file:./bundle/joram-tools-rest-jmx.jar
```

Then you need to stop the server, delete Felix bundle cache 'data/felix/', and restart the server.

3.1.2 Configuration

The JoramMQ JMX Rest implementation does have some configuration options. These are configured via the Felix properties file in the conf directory:

- Jetty configuration (see section 1.1.2).
- Authentication: by default all users are allowed to connect to the JoramMQ JMX Rest component, by defining the properties below you can restrain its access:
  - rest.jmx.root: defines the name of the allowed user.
  - rest.jmx.password: defines the password of the allowed user.
- By setting the rest.jmx.ipallowed property you can restrict access from a list of IP addresses (see section 1.1.2).

3.2 Usage

The JoramMQ JMX Rest interface publishes a variety of REST resources to perform various operations on JMX Mbeans. For each response there is a set of links allowing to access the additional information.

Authentication

If the JoramMQ JMX API is configured to require authentication (see section 3.1.2) you need to add an authorization header in the request. This property is an encoded string built from the user's name and password separated by a colon ':'.

For example to get the number of pending messages in the destination 'queue' with a curl request:

- curl --user "admin:admin" --get http://192.168.1.36:8989/joram/jmx/domains/Joram%230/Joram%230:name=queue,type=Destination/PendingMessageCount

3.2.1 List JMX domains

You get the list of all JMX domains by doing a simple GET on the following URI pattern:

- http://host:port/joram/jmx/domains

It returns the list of JMX domains either in HTML or JSON depending of the required document type.

3.2.2 List JMX object's names of a domain

You get the list of all JMX object's name of a domain by doing a simple GET on the following URI pattern:

- http://host:port/joram/jmx/domains/{domain}

It returns the list of all JMX object's name of the domain either in HTML or JSON depending of the required document type.

3.2.3 List attributes of a JMX Mbean

You get the list of all attributes of a particular MBean by doing a simple GET on the following URI pattern:

- http://host:port/joram/jmx/domains/{domain}/{object-name}

It returns the list of all attributes (name and value) of a particular MBean either in HTML or JSON depending of the required document type.

3.2.4 Get the value of an attribute.

You get the value of a particular attribute by doing a simple GET on the following URI pattern:

- http://host:port/joram/jmx/domains/{domain}/{object-name}/{attribute}

It returns the value of the specified attribute.

3.3 Use with an HTML browser

Since the protocol relies on a standard protocol (HTTP) and format (HTML/JSON), the JoramMQ JMX Rest interface can be even accessed from within the browser with a simple URL. The following URLs give a human readable view of the interface:

- http://host:port/joram/jmx
  - Give a browasble view of JMX Mbeans.
- http://host:port/joram/jmx/domains

- Give the list of all JMX domains.

- http://host:port/joram/jmx/domains/{domain}
  - Give the list of all MBeans of the specified domain.
- http://host:port/joram/jmx/domains/{domain}/{ object-name}
  - Shows all attributes of the MBean.

4 Exposing JMX using Jolokia

The JoramMQ default configuration ships with the Jolokia http agent deployed. Jolokia is a remote JMX over HTTP bridge that exposes Mbeans. For a full guide as to how to use, you can refer to the Jolokia documentation or use HawtIO as described in the JoramMQ documentation. However a simple example to query the number of connected clients would be to use a browser and go to the URL:

```
http://localhost:7050/jolokia/read/
com.scalagent.jorammq:host=localhost,manager=ClientManager/ConnectCount
```

After authentication[6] this would give you back something like the following:

```
{"request":
{"mbean":"com.scalagent.jorammq:host=localhost,manager=ClientManager","attribute":
"ConnectCount","type":"read"},"value":0,"timestamp":1455548959,"status":200}
```

Note: You can use the "wget" utility to test this easily. Simply execute a command like this:

```
wget --user admin --password adminpass
http://192.168.1.101:7050/jolokia/read/com.scalagent.jorammq:host=localhost,manage
r=ClientManager/ConnectCount
-----
{
  "request": {
      "mbean":"com.scalagent.jorammq:host=localhost,manager=ClientManager",
      "attribute":"ConnectCount",
      "type":"read"
    },
  "value":0,
  "timestamp":1455549378,
  "status":200
}
```

[6] *By default admin/adminpass as configured in the Felix configuration file.*