3

RabbitMQ – Java Client

Jorge Simão, Ph.D.

Content

» AMQP & RabbitMQ
» Channels & Queues
» Exchanges & Bindings
» Request-Reply
» Reliability

AMQP

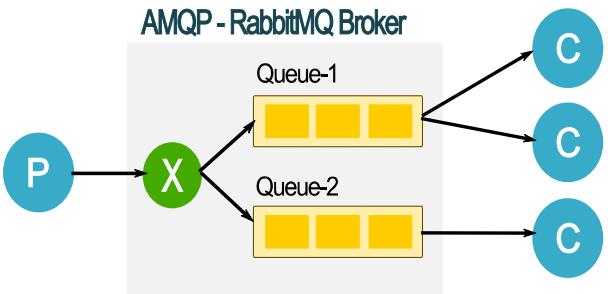
AMQP is an open-standard protocol for reliable message-based communication. AMQP is a wire-level protocol that aims to standardize all wire-level interactions between communicating endpoint and brokers. Mechanisms specified by AMQP includes:

- On-wire type encoding and message-frames
- Flow-Control
- Queuing and dispatching
- Multi-semantics reliability based on persistence, acknowledges and transactions
- Secure communication

RabbitMQ

RabbitMQ is a leading implementation of AMQP, implemented in Erlang and sponsored by Pivotal, consisting of broker (messaging-server) and client APIs for several languages, including Java, Erlang, C#, and other. RabbitMQ uses an extensible plugin-based architecture. In addition to the standard mechanisms specified in AMQP, custom extensions to AMQP are also implemented by RabbitMQ plugins.

A typical distributed system architecture based on AMQP and RabbitMQ consists of one (or more) message **Producer** application(s) and one or more message **Consumer** application(s). AMQP allows for applications to communicate directly, but in most settings applications interact via a middle-man – a message broking service.



RabbitMQ Installation

RabbitMQ is straightforward to install and getting started with. Since RabbitMQ is implemented in Erlang, the first step is to implement Erlang runtime environment. Next step, is to download RabbitMQ bundle, and follow the instruction in the RabbitMQ website for installation and setup. For Windows-OS, a installer is also available that performs all the installation&configuration steps automatically. (See list of resources at the end of this refcard.)

Once Erlang and RabbitMQ is installed, you can start a RabbitMQ server (broker) manually by running from the command **rabbitmq-server**. (In Windows-OS, you should run the command in an Administrator console).

» TIP: Starting RabbitMQ Broker Manually

```
> rabbitmq-server
RabbitMQ 3.4.1. Copyright (C) 2007-2014 GoPivotal, Inc.
...
Starting broker... completed with 0 plugins.
```

The output message produced by the server, includes the location of the log file. A useful place to look if unexpected issues occur during start-up.

In Windows, the command **rabbitmq-service install** can also be used to install the RabbitMQ server as a windows service, that is started automatically when the server machine starts. The Windows installer, register RabbitMQ as a service by default.

RabbitMQ Administration

RabbitMQ bundle also includes a very useful administration tool **rabbitmqctl** that can be used to check the status and state of the broker:

» TIP: Checking Status of RabbitMQ Broker

```
> rabbitmqctl status
Status of node 'rabbit@mypc.acme.org' ...
```

RabbitMQ – Java Client

```
[{pid,5812},
...]
```

» TIP: List Queues in RabbitMQ Broker

```
> rabbitmqctl list_queues
```

```
Listing queues ...
stocks-q 0
options-q 0
```

» TIP: List Exchanges in RabbitMQ Broker

```
>rabbitmqctl list_exchanges
```

```
Listing exchanges ...
amq.direct      direct
amq.fanout      fanout
amq.headers     headers
amq.match       headers
amq.rabbitmq.log        topic
amq.rabbitmq.trace      topic
amq.topic       topic
```

Table below summarizes some of the commands supported by **rabbitmqctl:**

Command	Description
status	Check connectivity status with broker
stop	Stop server/broker
list_queues	List declared queues
list_exchanges	List declared exchanges
list_bindings	List queue-exchange bindings

RabbitMQ Java Client

RabbitMQ includes a **Java** client library distributed as **JAR** file **rabbitmq-java-client-bin-x.y.z**. The library can be used by simply including it in the *classpath* of the Java program (or declare it as a **maven** dependency), together with a couple of dependencies.

» TIP: Java +RabbitMQ Program Compilation

```
> javac -cp rabbitmq-java-client-bin-x.y.z.jar MyProducer.java MyConsumer.java
```

» TIP[Linux&Mac]: Java +RabbitMQ Program Execution

```
> export CP=.:commons-io-1.2.jar:commons-cli-1.1.jar:\
rabbitmq-java-client-bin-x.y.z.jar (terminal window 1&2)
> java -cp $CP MyConsumer.java (terminal window 1)
> java -cp $CP MyProducer.java (terminal window 2)
```

» TIP[Windows]: Java +RabbitMQ Program Execution

```
> set CP=.;commons-io-1.2.jar;commons-cli-1.1.jar;rabbitmq-java-client-bin-x.y.z.jar (terminal window 1&2)
> java -cp %CP% MyConsumer.java (terminal window 1)
> java -cp %CP% MyProducer.java (terminal window 2)
```

Opening a Connection to a Broker

To open a **Connection** to a RabbitMQ broker a **ConnectionFactory** should be first configured (or looked-up). The configuration includes all the properties need to specify where and how to connect to the broker.

» Example: Connecting to Localhost Broker

```
import com.rabbitmq.client.Channel;

ConnectionFactory factory = new ConnectionFactory();
factory.setHost("localhost");
Connection connection = factory.newConnection();
Channel channel = connection.createChannel();

//...use channel to send&recv messages...

channel.close();
connection.close();
```

» Example: Connecting with URI

```
ConnectionFactory factory = new ConnectionFactory();
try {
    factory.setUri("amqp://user:pass@host:port/vhost");
} catch (KeyManagementException |
    NoSuchAlgorithmException | URISyntaxException e) {
    ...
}
Connection connection = factory.newConnection();
...
```

» Example: Connecting to Alternative Brokers

```
Address[] addrs = new Address[]{
    new Address("master-rabbit", DEFAULT_AMQP_PORT),
    new Address("failover-rabbit", DEFAULT_AMQP_PORT};
ConnectionFactory factory = new ConnectionFactory();
Connection connection = factory.newConnection(addrs);
...
```

Once a **Connection** is open a **Channel** should be created to perform message sending and receive operation. Queue, exchange, and binding declaration operations are also provided by the Channel API.

Queues, Exchanges, and Bindings

In AMQP, messages are routed to **Queue**s for reception, and sent to **Exchange**s. **Binding**s specify the rules by which Exchanges are connected to Queues.

Queues need to be declared before a message is received from it or before it is bound to an Exchange. A Queue may have a name, used as routing key – although, this is not strictly required. Queue and exchanges are declared with several properties. Declaring the same queue or exchange with the same properties is allowed – it works as an idempotent operation (e.g. both the send and receiver can declare a queue). On the other hand, it is an error to declare the same queue or exchange with different properties.

» Example: Declaring a Named Queue

```
boolean durable = true;
boolean exclusive = false;
boolean autoDelete = false;
channel.queueDeclare(“trading-q”, durable, exclusive,
autoDelete, null);
```

» Example: Declaring Durable Exchange

```
channel.exchangeDeclare(“stocks-x”,  "fanout", true
/*durable*/);
```

» Example: Declaring and Binding Auto-Named Queue

```
channel.exchangeDeclare(“trading-x”,  "direct");
String queue = channel.queueDeclare().getQueue();
channel.queueBind(queue, “trading-x”, “” /*routingKey*/);
```

Message Publishing

Messages are sent to RabbitMQ broker by calling **Channel.basicPublish()**, using a exchange name and a (optional) routing key to specify where the message should be forward.

» Example: Message Publishing to Exchange

```
byte[] body = "SYM RabbitMQ, 1.23".getBytes();
channel.basicPublish(“stocks-x”, “” /*routingKey*/, null,
body);
```

The exchange with empty name is predefined as *default exchange*. All Queue are bound automatically to this default exchange, with a routing key that matches the name of the queue. This means that a producer can send messages to specific queue by using the default/empty exchange and the queue name as routing key.

» Example: Publishing to Queue (w/ default Exchange)

```
byte[] body = "SYM RabbitMQ".getBytes();
channel.basicPublish(“”, “stocks-q”, null, body);
```

Message Properties

In RabbitMQ API, there is no abstraction that completely defines a message. A message is made out of a body modelled as a plain byte array, and list of meta-annotations or properties modelled with type **AMQP.BasicProperties**. The utility class **MessageProperties** defines several common property combination values.

» Example: Publish Durable Message

```
channel.basicPublish(“stocks-x”, routingKey,
    MessageProperties.PERSISTENT_TEXT_PLAIN, body);
```

Several properties are pre-defined, while others are application defined headers. The class **AMQP.BasicProperties.Builder**, implementing the *builder design-pattern* with a fluent API, is commonly used created an instance of **AMQP.BasicProperties**.

» Example: Publish Message with Properties Set

```
AMQP.BasicProperties props =
    new AMQP.BasicProperties.Builder()
        .contentType("text/plain")
        .deliveryMode(2 /*durable*/)
        .priority(1)
        .expiration("2000")
        .build();
channel.basicPublish(“stocks-x”, “” /*routingKey*/, props,
msg.getBytes());
```

Table below summarizes the properties of messages:

Property w/ Example	Description
appId(“stock-monitor”)	Application ID
clusterId(“emea-cluster”)	Cluster ID
contentEncoding(“utf-8”);	Character Encoding
contentType(“application/xml”)	Content MIME Type
correlationId(UUID.randomUUID() .toString())	Request-Response Correlation ID

deliveryMode(2)	Durability: 1 - non-durable; 2 - durable
expiration("1000")	Min. time broker keeps message before discarding if not consumed
messageId("1234567")	Message ID
priority(10)	Delivery Priority
replyTo("stocks-q")	Routing-key to send response message
timestamp(new Date())	Message creation time
type("?SYM")	Message category
userId("admin")	User/Principal ID

Map<String,Object> map = new HashMap<>();
map.put("name", value);
builder.headers(map);
	Application defined headers

Message Reception

In AMQP and RabbitMQ, message can be received from queue synchronously *(pull mode)*, asynchronously *(push mode)*. Synchronous reception is done by invoking method **Channel.basicGet()** to get a single individual message.

» Example: Synchronous Reception

```
boolean autoAck = false;
GetResponse msg = channel.basicGet("trading-q", autoAck);
 AMQP.BasicProperties props = msg.getProps();
byte[] body = msg.getBody();
long deliveryTag = msg.getEnvelope().getDeliveryTag();
...
```

Asynchronous reception is done by registering an implementation of interface **Consumer**, with method **Channel.basicConsume()**. Interface **Consumer** defines several callback methods related with message delivery. A simple approach to create a **Consumer** is to extend the class **DefaultConsumer**, and override only the required methods – such as **handleDelivery()** for processing received message.

» Example: Asynchronous Processing with Consumer

```
channel.basicConsume("trading-q", true /*autoAck*/,
"myConsumerTag", new DefaultConsumer(channel) {
    @Override
    public void handleDelivery(String consumerTag,
        Envelope envelope,
        AMQP.BasicProperties properties,
        byte[] body) throws IOException {

            String routingKey = envelope.getRoutingKey();
            String contentType = properties.contentType;
            long deliveryTag = envelope.getDeliveryTag();

            //process message…
    }
});
```

A useful implementation of **Consumer** is provided out-of-the-box by the Java client API – the **QueueingConsumer**, that implements application level queuing of messages.

» Example: QueueingConsumer

```
QueueingConsumer consumer =
        new QueueingConsumer(channel);
channel.basicConsume("trading-q", true, consumer);
try {
    while (true) {
        QueueingConsumer.Delivery delivery =
                consumer.nextDelivery();
        String msg = new String(delivery.getBody());
        //...
    }
} catch (ShutdownSignalException e) {}
```

Exchanges Types

AMQP and Rabbit MQ supports several types of exchanges with different routing rules. Table below summarizes the exchanges types.

Exchange Type	Description
direct	Route to bound queue(s) based on routing key
fanout	Route to all bound queues(ignores routing key)
topic	Route to bound queues with matching routing key
headers	Route to bound queue with matching header values

Topic exchanges can be use to distributed messages based on pattern matching with the routing key specified by message publisher. The pattern syntax assumes topic names are structured as tokens or words separated by dot, such as: **comp.java.rabbitmq**. The following wild card are supported:

Wildcard	Description
* (star)	Match exactly one word
# (hash)	Match zero or more words

Table below shows some example topic pattern:

Pattern	Description
comp.java.#	All topics and sub-topics related to Java
comp.amqp.*	All topics on AMQP
..rabbitmq	Sub-sub-topics on RabbitMQ

Snippet below show how to bind a queue and publish to a topic exchange.

» Example: Bind and Consume from Topic Exchange

```
channel.exchangeDeclare(“news-x”,  "topic");
String queue = channel.queueDeclare().getQueue();
channel.queueBind(queue, “news-x”, “comp.java.#”);
```

» Example: Publishing to a Topic Exchange

```
channel.basicPublish(“news-x”, “comp.java.rabbitmq”, null,
“RabbitMQ Java client updated...”.getBytes());
```

RPC: Request-Reply Interaction

Message-based communication is at the basic level uni-directional. However, sometimes is desirable to perform two-way *request-reply* interaction between endpoint similarly to what happens in Client-Server interaction - e.g. using a remote invocation framework (RPC) or web-service.

The main complication of request-reply is to make sure that the response message is correlated with the request message, even in the presence of concurrent requests made by different Producers. This is achieved by using a dedicate response queue, and set the value of the message property **replyTo** to this response queue. The property **correlationID** is also set on the response message to match the **correlationID** (or message ID) of the request message.

» Example: Request-Reply Interaction (Client)

```
replyQ = channel.queueDeclare().getQueue();

String corrId = java.util.UUID.randomUUID().toString();
BasicProperties props = new BasicProperties.Builder()
                        .replyTo(replyQ)
                        .correlationId(corrI)
                        .build();

channel.basicPublish("", "stocks-q", props, msg.getBytes());

QueueingConsumer c = new QueueingConsumer(channel);
channel.basicConsume(replyQ, true, c);
QueueingConsumer.Delivery d;
do { d = c.nextDelivery(); }
while (!d.getProperties().getCorrelationId().equals(corrId));
String reply = new String(delivery.getBody());
```

» Example: Request-Reply Interaction (Server)

```
QueueingConsumer c = new QueueingConsumer(channel);
channel.basicConsume("stocks-q", true, c);
while (true) {
    QueueingConsumer.Delivery d = c.nextDelivery();
    String request = new String(d.getBody());
    String reply = “Reply:” + request;
    BasicProperties props = d.getProperties()
    channel.basicPublish("", props.getReplyTo(),
        new BasicProperties.Builder()
            .correlationId(props.getCorrelationId()).build(),
        reply.getBytes());
}
```

Because request-reply interaction pattern is commonly used, RabbitMQ client lib provides the class **RpcClient** to simplify the implementation of the client side.

» Example: Request-Reply with Built-in RPC

```
RpcClient rpc = new RpcClient(channel, “stocks-x”, “”
/*routingKey*/);
String msg = “SYM RabbitMQ”;
byte[] reply = primitiveCall(msg.getBytes());
System.out.println(“RabbitMQ:” + new String(reply));
```

Ackwnowledges

In the examples above, RabbitMQ Channel was acknowledging messages automatically. Consumer may also choose to acknowledge message manually by calling **Channel.basicAck()**. A boolean parameter specified is multiple messages (batch) should be acknowledged.

» Example: Explicit Acknowledge of Messages

```
channel.basicConsume(“trading-q”, false /*autoAck*/,
"myConsumerTag", new DefaultConsumer(channel) {
    @Override
```


```
    public void handleDelivery(String consumerTag,
        Envelope envelope,
        AMQP.BasicProperties properties,
        byte[] body) throws IOException {

            //process message…

            channel.basicAck(envelope.getDeliveryTag(),
                false /*multiple*/);
        }
    });
```

Messages can also be explicitly rejected by calling **Channel.basicReject()**. For rejecting multiple messages, the method **Channel.basicNack()** should be used.

» Example: Rejecting an Individual Message

```
channel.basicConsume("trading-q", false /*autoAck*/,
"myConsumerTag", new DefaultConsumer(channel) {
    @Override
    public void handleDelivery(String consumerTag,
        Envelope envelope, AMQP.BasicProperties props,
        byte[] body) throws IOException {
        if (!isValid(body)) {
            channel.basicReject(envelope.getDeliveryTag(),
                    false /*requeue*/);
        }
    }
});
```

» Example: Rejecting Multiple Message

```
    public void handleDelivery(String consumerTag,
        Envelope envelope, AMQP.BasicProperties props,
        byte[] body) throws IOException {
        if (!isValid(body)) {
            channel.basicNack(envelope.getDeliveryTag(),
                true /*multiple*/, true /*requeue*/);
        }
    }
```

Transactions

Multiple messaging operation can be grouped in an atomic *unit-of-work* using transactions. Method **Channel.txSelect()** is used to setup a Channel in transactional mode. Method **Channel.txCommit()** commits a transaction, and method **Channel.txRollback()** rollbacks a transaction (usually called when an **Exception** occurs).

» Examples: Publishing with Transactions

```
try {
    String[] stocks = {"SYM RabbitMQ = 99.9",
                       "SYM LinuxLTD = 10.0",
                       "SYM XMQ = 50.0"};
    channel.txSelect();
    for (String stock: stocks) {
        channel.basicPublish("", "stocks-q", null,
                             stock.getBytes());
    }
    channel.txCommit();
} catch (Exception e) {
    channel.txRollback();
}
```

» Examples: Receiving with Transactions

```
try {
    channel.txSelect();
    for (int i=0; i<3; i++) {
        GetResponse msg = channel.basicGet("trading-q",
true);
        System.out.println(new String(msg.getBytes()));
    }
    channel.txCommit();
} catch (Exception e) {
    channel.txRollback();
}
...
```

Spring AMQP

Java application development can be further simplified with a more high-level API provided by **Spring AMQP**. The class **RabbitTemplate** is used to send messages and to perform synchronous reception. This class transparently manages the life-cycle of resources such as **Connection** and **Channels**. Messaging operation always involves the opening and closing of a **Connection** and a **Channel**. For increased performance caching of these resources is provided by class **CachingConnectionFactory**.

» Example: Publishing with RabbitTemplate

```
ConnectionFactory cf = new CachingConnectionFactory();
RabbitTemplate template = new RabbitTemplate(cf);
template.convertAndSend("stocks-x", "nasdaq", "SYM
RabbitMQ=99.9");
```

» Example: Synchronous Receiving with RabbitTemplate

```
Message msg = template.receiveAndConvert("stocks-q");
MessageProperties props = msg.getMessageProperties();
System.out.println(props.getContentType() + ":" +
    new String(msg.getBytes())
```


Asynchronous reception requires the additional abstraction of a **MessageListenerContainer.** Snippet below show how a Java8 lambda-expression together with a **MessageListenerAdapter** can be registered to receive and process messages.

» Example: Aynchronous Receive with ListenerContainer

```
SimpleMessageListenerContainer container = new
SimpleMessageListenerContainer(cf);
container.setMessageListener(new MessageListenerAdapter(
    (msg)->System.out.println(msg));
container.setQueueNames("stocks-q");
container.start();
```

A class specific to RabbitMQ is provided to perform administrative tasks such as declaring queues, exchanges, and bindings.

» Example: Declaring Queues and Exchanges

```
RabbitAdmin admin = new RabbitAdmin(cf);
Queue queue = new Queue("stocks-q");
admin.declareQueue(queue);
TopicExchange exchange = new TopicExchange("stocks-x");
admin.declareExchange(exchange);
admin.declareBinding(BindingBuilder.bind(queue).to(exchange).with("foo.*"));
```

Resources

- Tutorials on RabbitMQ – http://www.rabbitmq.com/getstarted.html
- Java Client API Guide – http://www.rabbitmq.com/api-guide.html
- Spring AMQP Project – http://projects.spring.io/spring-amqp
- Download RabbitMQ – http://www.rabbitmq.com/download.html
- Git repository for Spring AMQP samples - https://github.com/SpringSource/spring-amqp-samples

About the Author

Jorge Simão is a software engineer and IT Trainer, with two decades long experience on education delivery both in academia and industry. Expert in a wide range of computing topics, he his an author, trainer, and director (Education & Consulting) at EInnovator. He holds a B.Sc., M.Sc., and Ph.D. in Computer Science and Engineering.

RabbitMQ Training

Take an intensive three-day instructor-led course in+ RabbitMQ, and learn how to setup and develop applications with RabbitMQ. The course covers RabbitMQ installation and configuration, developing messaging applications with the Java APIs, including Spring RabbitMQ, and delves into more advanced topics including clustering, high availability, performance, and security. Modules are accompanied by lab exercises that provide hands-on experience. Book now an on-site class: **www.einnovator.org/course/rabbitmq**

Contacts

Training – Bookings & Inquiries
training@einnovator.org

Consultancy – Partnerships & Inquiries
consulting@einnovator.org

General Info
info@einnovator.org