Business Requirement and Design Specification

CRM Message Queue Guide

Prepared For

Version 1.3

Delivery Date:
<03/29/2019>

Revision History

Author

Name	Role	Date of Preparation	Signature
Rahul Kawle	Lead BA – I		RK
Reviewer(s)			
Name	**Role**	**Date of Review**	**Signature**
Approving Authority			
Name	**Role**	**Date of Approval**	**Signature**
Revision History			
Version Number	**Date of Revision**	**Author**	**Details of Changes**
1.0	March 26, 2019	Rahul Kawle	Initial Draft
1.1	March 27, 2019	Rahul Kawle	Added swim lane diagram
1.2	March 28, 2019	Rahul Kawle	Added details related to CRM, Exchange, RabbitMQ and the generic workflow
1.3	March 29, 2019	Rahul Kawle	Added RabbitMQ details and config changes for CRM. Minor updates to all the sections.

Contents

About

This document details the proposed design for using the Message Queue system to pull the data from the Message Queue and update the required objects in third party applications based on the changes pushed by CRM Workflow triggers into the Queue. Please note that the CRM Workflow engine and the Message Queue system are tightly knit together.

Intended Audience

The intended audience for this document shall be the Client/Vendor partners and their IT teams including the CRM Application Development, Product Development and Integration development teams.

Overview

The CRM Workflow Engine and the Message Queue Systems are tightly integrated to provide the end users with a powerful end-to-end solution. The workflow engines have a defined trigger points such that as and when any add/modify/delete event occurs, the workflow engine pushes the event into the defined Queue. The Client/Vendor partner can also update the data in CRM via Exchange Proxy requests and response.

CRM

In use by industry leaders for more than two decades, CRM solutions for practice, and agency management are trusted across the industry to help improve marketing efforts, provide valuable reporting capabilities that drive better business decisions, and create a customer-centric environment that helps increase sales revenues.

Exchange System

Exchange enables users from different CRM systems to connect and make use of the integrations provided by Integrator based services. This in effect means ease of deployment as the integrations are made available to multiple CRM system users from one place. Exchange controls access to the integrations and it monitors the traffic from service providers and users. It also ensures secured data flow.

Exchange Service

The client contacts sales team for the integration needs. sales team shares the information with product management team and here the product management team then works with the client and identifies all the use cases. Once all the information is available, the product management team contacts the SI Support team.

The SI Support team also creates a Exchange Integration Service based on the information provided for the client. For the usage of Message Queue system, the Exchange has the ability to create integration services with Type = 'MessageQueue'.

CRM User on SDC Office on UAT server

The SI Support team creates a SO app user on our Software Development Community office on our User Acceptance Testing server which can used by the Client/Vendor partners for development and testing.

Exchange Adapter

Once the service is created, the SI Support team also creates an adapter that helps the Integration service to communicate between CRM and Exchange.

SI Support team also provides a proxy request post HTML file to the Client/Vendor partners. This file contains sample proxy request along with the adapter credentials and registration key to be used for a specific user.

RabbitMQ

RabbitMQ is a messaging broker. It accepts published messages, routes them and, if there were queues to route to, stores them for consumption or immediately delivers to consumers, if any. Consumers consume from queues. In order to consume messages there has to be a queue. When a new consumer is added, assuming there are already messages ready in the queue, deliveries will start immediately. The target queue can be empty at the time of consumer registration. In that case first deliveries will happen when new messages are queued. An attempt to consume from a non-existent queue will result in a channel-level exception with the code of 404 Not Found and render the channel it was attempted on to be closed.

Format of Data in Message Queue

```
{
  "appUserId": 2,
  "appUserOfficeId": 918,
  "objectId": 1840,
  "objectOfficeId": 918,
```

```
  "tableId": 98,
  "eventID": 1,
  "createdOn": "Mar 26, 2019 9:52:52 AM",
  "sxUrl": "",
  "postUrl": "http://service.leads360.com/ClientService.asmx",
  "officeName": "SDC_UAT_Client/Vendor Partner",
  "userName": "Admin",
  "modifiedColsMap": {}
}
```


Overview

RabbitMQ Java client uses com.rabbitmq.client as its top-level package. The key classes and interfaces are:

- Channel: represents an AMQP 0-9-1 channel, and provides most of the operations (protocol methods).
- Connection: represents an AMQP 0-9-1 connection
- ConnectionFactory: constructs Connection instances
- Consumer: represents a message consumer
- DefaultConsumer: commonly used base class for consumers
- BasicProperties: message properties (metadata)
- BasicProperties.Builder: builder for BasicProperties

Protocol operations are available through the Channel interface. Connection is used to open channels, register connection lifecycle event handlers, and close connections that are no longer needed. Connections are instantiated through ConnectionFactory, which is how you configure various connection settings, such as the vhost or username.

Connections and Channels

The core API classes are Connection and Channel, representing an AMQP 0-9-1 connection and channel, respectively. They are typically imported before used:

```
import com.rabbitmq.client.Connection;

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

Connecting to RabbitMQ

The following code connects to a RabbitMQ node using the given parameters (host name, port number, etc):

```
ConnectionFactory factory = new ConnectionFactory();

// "guest"/"guest" by default, limited to localhost connections

factory.setUsername(userName);

factory.setPassword(password);

factory.setVirtualHost(virtualHost);

factory.setHost(hostName);

factory.setPort(portNumber);


Connection conn = factory.newConnection();
```

All of these parameters have sensible defaults for a RabbitMQ node running locally. The default value for a property will be used if the property remains unassigned prior to creating a connection:

Property	Default Value
Username	“guest”
Password	“guest”
Virtual Host	“/”
Hostname	“localhost”
Port	5672 for regular connections, 5671 for connections that use TLS

Alternatively, URLs may be used:

```
ConnectionFactory factory = new ConnectionFactory();

factory.setUri("amqp://userName:password@hostName:portNumber/virtualHost");

Connection conn = factory.newConnection();
```

All of these parameters have sensible defaults for a stock RabbitMQ server running locally.

Note that user guest can only connect from localhost by default. This is to limit well-known credential use in production systems.

The Connection interface can then be used to open a channel:

```
Channel channel = conn.createChannel();
```

The channel can now be used to send and receive messages, as described in subsequent sections.

Successful and unsuccessful client connection events can be observed in server node logs.

Disconnecting from RabbitMQ

To disconnect, simply close the channel and the connection:

```
channel.close();

conn.close();
```

Note that closing the channel may be considered good practice, but isn't strictly necessary here - it will be done automatically anyway when the underlying connection is closed.

Client disconnection events can be observed in server node logs.

Using Exchanges and Queues

Client applications work with exchanges and queues, the high-level building blocks of the protocol. These must be declared before they can be used. Declaring either type of object simply ensures that one of that name exists, creating it if necessary.

Continuing the previous example, the following code declares an exchange and a server-named queue, then binds them together.

```
channel.exchangeDeclare(exchangeName, "direct", true);

String queueName = channel.queueDeclare().getQueue();

channel.queueBind(queueName, exchangeName, routingKey);
```

This will actively declare the following objects, both of which can be customised by using additional parameters. Here neither of them have any special arguments.

- a durable, non-autodelete exchange of "direct" type
- a non-durable, exclusive, autodelete queue with a generated name

The above function calls then bind the queue to the exchange with the given routing key.

Note that this would be a typical way to declare a queue when only one client wants to work with it: it doesn't need a well-known name, no other client can use it (exclusive) and will be cleaned up automatically (autodelete). If several clients want to share a queue with a well-known name, this code would be appropriate:

```
channel.exchangeDeclare(exchangeName, "direct", true);

channel.queueDeclare(queueName, true, false, false, null);

channel.queueBind(queueName, exchangeName, routingKey);
```

This will actively declare:

- a durable, non-autodelete exchange of "direct" type
- a durable, non-exclusive, non-autodelete queue with a well-known name

Many Channel API methods are overloaded. These convenient short forms of exchangeDeclare, queueDeclare and queueBind use sensible defaults. There are also longer forms with more parameters, to let you override these defaults as necessary, giving full control where needed.

This "short form, long form" pattern is used throughout the client API uses.

Deleting Entities and Purging Messages

A queue or exchange can be explicitly deleted:

```
channel.queueDelete("queue-name")
```

It is possible to delete a queue only if it is empty:

```
channel.queueDelete("queue-name", false, true)
```

or if it is not used (does not have any consumers):

```
channel.queueDelete("queue-name", true, false)
```

A queue can be purged (all of its messages deleted):

```
channel.queuePurge("queue-name")
```

Publishing Messages

To publish a message to an exchange, use Channel.basicPublish as follows:

```
byte[] messageBodyBytes = "Hello, world!".getBytes();

channel.basicPublish(exchangeName, routingKey, null, messageBodyBytes);
```

For fine control, you can use overloaded variants to specify the mandatory flag, or send messages with pre-set message properties:

```
channel.basicPublish(exchangeName, routingKey, mandatory,

            MessageProperties.PERSISTENT_TEXT_PLAIN,

            messageBodyBytes);
```

This sends a message with delivery mode 2 (persistent), priority 1 and content-type "text/plain". You can build your own message properties object, using a Builder class mentioning as many properties as you like, for example:

```
channel.basicPublish(exchangeName, routingKey,

        new AMQP.BasicProperties.Builder()

          .contentType("text/plain")

          .deliveryMode(2)

          .priority(1)

          .userId("bob")
```

```
            .build(),
            messageBodyBytes);
```

This example publishes a message with custom headers:

```
Map<String, Object> headers = new HashMap<String, Object>();
headers.put("latitude",  51.5252949);
headers.put("longitude", -0.0905493);

channel.basicPublish(exchangeName, routingKey,
          new AMQP.BasicProperties.Builder()
            .headers(headers)
            .build(),
            messageBodyBytes);
```

This example publishes a message with expiration:

```
channel.basicPublish(exchangeName, routingKey,
          new AMQP.BasicProperties.Builder()
            .expiration("60000")
            .build(),
            messageBodyBytes);
```

We have not illustrated all the possibilities here.

Note that BasicProperties is an inner class of the autogenerated holder class AMQP.

Invocations of Channel#basicPublish will eventually block if a resource-driven alarm is in effect.

Retrieving Individual Messages ("Pull API")

To explicitly retrieve messages, use Channel.basicGet. The returned value is an instance of GetResponse, from which the header information (properties) and message body can be extracted:

```
boolean autoAck = false;

GetResponse response = channel.basicGet(queueName, autoAck);

if (response == null) {

   // No message retrieved.

} else {

   AMQP.BasicProperties props = response.getProps();

   byte[] body = response.getBody();

   long deliveryTag = response.getEnvelope().getDeliveryTag();

   ...
```

and since the autoAck = false above, you must also call Channel.basicAck to acknowledge that you have successfully received the message:

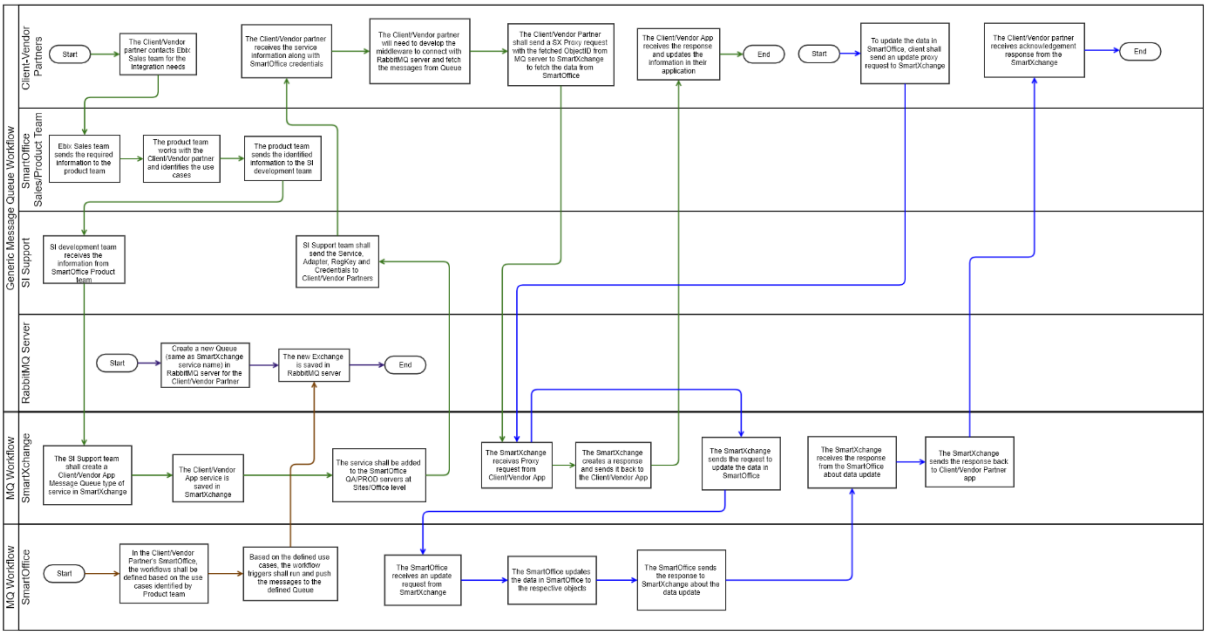
```
   ...

   channel.basicAck(method.deliveryTag, false); // acknowledge receipt of the message

}
```

The Client/Vendor Partner needs to use the Plugins to develop their code for Message Queue. More information is available @ https://www.rabbitmq.com/devtools.html

Workflow Swim lane Diagram



Workflow

- The Client/Vendor partner contacts Sales team for the Integration needs
- The Sales team provides the information to the Product Management team
- The Product Management team works with the Client/Vendor partners to gather all the information and the user cases
- The Product Management team provides all the information to the SI Support team
- The SI Support team creates following for the Client/Vendor Partner, viz.
    - A CRM user on Software Development Community office on User Acceptance Testing server
    - A Service in Exchange with Type = “MessageQueue” with Name = “Client/Vendor Partner App Name”
    - An adapter for the service to communicate with CRM
    - A Queue in RabbitMQ server having same name as the Service created in Exchange
    - The Client/Vendor partner needs to subscribe to this service to fetch the events from defined Message Queue, or else the Client/Vendor partner will not be able to fetch the events from the Queue
- The SI Support team sends all the above information to the client
- The Client/Vendor partner shall be required to develop the middleware to connect with RabbitMQ server and fetch the messages from Queue
- The Client/Vendor Partner shall send a SX Proxy request with the fetched ObjectID from MQ server to Exchange to fetch the data from CRM
- The Exchange receives proxy request from Client/Vendor App
- The Exchange fetches the data stored in the defined Queue
- The Exchange creates a response and sends it back to the Client/Vendor App
- The Client/Vendor App receives the response and updates the information in their application

RabbitMQ Server Details

URL	Will share the information separately
User	Will share the information separately
Pass	Will share the information separately
Queue Name	Will share the information separately
Virtual Host	Will share the information separately