Cloud Message Queue

Contents

Copyright GE Digital

Chapter 1

Overview

Topics:

Overview of Cloud Message Queue

Cloud Message Queue is a service that provides the ability to exchange messages between different applications, components, and devices in a highly durable and reliable manner. It is built upon Apache ActiveMQ and is backed by Amazon MQ, which is a managed message broker service for Apache ActiveMQ. Cloud Message Queue utilizes the features of Amazon MQ to ensure its high availability and message durability.

Cloud Message Queue is designed to replace Predix Message Queue which is built upon RabbitMQ. The Predix Message Queue service will no longer be available in future and therefore, GE Digital recommends you migrate to the Cloud Message Queue service before the Predix Message Queue service is deprecated.

Some of the salient features of Cloud Message Queue are as follows:

- It supports a wide range of clients, including Java Message Service (JMS) 1.1 and .NET Message Service (NMS).
- It is compatible with a range of programming languages, including Node.js, Go, Python, Ruby, and C++.
- It supports various wire-level protocols, such as Advanced Message Queuing Protocol (AMQP), Simple (or Streaming) Text Oriented Messaging Protocol (STOMP), OpenWire, WebSocket, and MQ Telemetry Transport (MQTT).
- It provides all the standard JMS features, including point-to-point messaging, publish/subscribe messaging, request/reply, persistent and non-persistent modes, JMS transactions, and distributed (XA) transactions.
- It provides the flexibility of sending messages through both queues and topics, using a single-broker. In point-to-point messaging, the ActiveMQ broker balances the load by routing each message from the queue to one of the available consumers in a sequential manner. In pub/sub messaging, the broker delivers each message to every consumer that has subscribed to the topic.
- In addition to basic queues and topics, it supports other complex patterns, such as composite and virtual destinations. Composite destinations are used for producers to send the same message to multiple destinations. Whereas, virtual destinations are used for publishers to broadcast messages through a topic to a pool of receivers subscribing through queues.

Note: Currently, Cloud Message Queue supports the following versions of ActiveMQ:

- 5.15.0
- 5.15.6
- 5.15.8
- 5.15.9
- 5.15.10

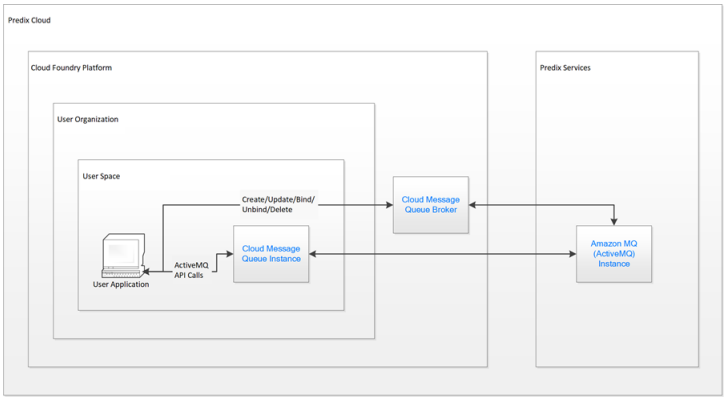
Chapter 2

Architecture

Topics:

Cloud Message Queue Architecture

The following architecture diagram provides a high-level information on how a user application interacts with an Amazon MQ (ActiveMQ) instance using the Cloud Message Queue service.



The ActiveMQ message broker sends the messages from one application to another. The application that creates a message is called the producer and the application that receives and processes that message is called the consumer of the message. The message broker routes each message through one of the following destinations:

- **Queue**: Used in a point-to-point messaging domain where the message awaits its delivery to a single consumer.
- **Topic**: Used in a publish/subscribe or pub/sub messaging domain where the message is delivered to multiple consumers that have subscribed to the topic.

Chapter 3

Getting Started with Cloud Message Queue

Topics:

Create Cloud Message Queue Service Instance

About This Task

You can create the Cloud Message Queue service instance with default or customized Amazon MQ configurations. For more information on Amazon MQ configurations, and their elements and attributes, refer to Amazon MQ documentation.

Procedure

1. In the Cloud Foundry (CF) Command Line Interface, run the following command to access the list of Cloud Message Queue plans that are available in the CF marketplace for subscription:

   ```
   cf m -s cloud-message-queue
   ```

2. Depending on whether you want to use the default or customized Amazon MQ configurations, perform one of the following tasks:

   - Create the instance with the default configurations.
   - Create the instance with customized configurations.

Create Service Instance with Default Configurations

Procedure

1. Access KBA 000036364, and then download the configuration file corresponding to the version of your ActiveMQ broker specified in the following table:

ActiveMQ Version	Configuration File
5.15.0	defaultConfigForEngineVersion5-15-0.zip
5.15.6	defaultConfigForEngineVersion5-15-6.zip
5.15.8	defaultConfigForEngineVersion5-15-8.zip
5.15.9	defaultConfigForEngineVersion5-15-9.zip
5.15.10	defaultConfigForEngineVersion5-15-10.zip

   Note: By default, the KahaDB persistence adapter is enabled in all the configurations. If you want to access the broker or destination metrics within your application, you must enable `<statisticsBrokerPlugin/>`.

2. Extract the content of the downloaded ZIP file.
3. In the Cloud Foundry (CF) Command Line Interface, run the following command to create the service instance:

   ```
   cf create-service cloud-message-queue <subscription_plan_name>
   <service_instance_name>
   ```

   Note:

   - The angle brackets (< >) in the command indicate placeholder text. You must replace the placeholder text with an appropriate value before running the command.
   - By default, Cloud Message Queue supports ActiveMQ V5.15.9 and the default configuration of ActiveMQ V5.15.9 will be used to create the instance.

Create Service Instance with Customized Configurations

About This Task

You can create the Cloud Message Queue service instance with customized values of the following parameters:

- ActiveMQ configuration XML
- ActiveMQ engine version
- Description string

Important: The angle brackets (< >) in this topic indicate placeholder text. If such placeholder text is part of a command, replace the placeholder text with an appropriate value before running the command.

Procedure

1. Create an XML file with the customized settings of your ActiveMQ broker.

   Note: For more information on customizing the settings of your ActiveMQ broker, refer to Amazon MQ documentation.
2. In the Base64 utility, run the following command to generate a Base64 encoded output of the content of the XML file:

   ```
   cat <config_file_name>.xml | base64 -w 0
   ```
3. Perform one of the following steps to create the service instance:
   - In the Cloud Foundry (CF) Command Line Interface, run the following command:

     ```
     cf create-service cloud-message-queue <subscription_plan_name>
     <service_instance_name> -c '{"EngineVersion":
     "<ActiveMQ_broker_engine_version>",
     "Description":"<description_string>",
     "Data":"<Base64_encoded_content_of_.xml_file>"}'
     ```
   - Copy the Base64 encoded output of the content of the XML file to a `<custom-config>.json` file along with the version of your ActiveMQ broker and description string, and then in the CF Command Line Interface, run the following command:

     ```
     cf create-service cloud-message-queue <subscription_plan_name>
     <service_instance_name> -c <path_of_<custom-config>.json>
     ```

   Note: By default, Cloud Message Queue supports ActiveMQ V5.15.9. If you do not specify the version of your ActiveMQ engine broker in the command, Cloud Message Queue will use ActiveMQ V5.15.9 to create the instance.

Update Cloud Message Queue Service Instance

About This Task

You can update the Cloud Message Queue service instance by updating the values of the following parameters:

- ActiveMQ configuration XML
- Description string

Important:

- Any configuration changes to the service instance restarts the instance immediately and may take up to 25 minutes to be functional.
- The angle brackets (< >) in this topic indicate placeholder text. If such placeholder text is part of a command, replace the placeholder text with an appropriate value before running the command.

Procedure

1. Create an XML file with the updated settings of your ActiveMQ broker.

   Note: For more information on updating the settings of your ActiveMQ broker, refer to Amazon MQ documentation.
2. In the Base64 utility, run the following command to generate a Base64 encoded output of the content of the XML file:

   ```
   cat <config_file_name>.xml | base64 -w 0
   ```
3. Perform one of the following steps to update the service instance:
   - In the Cloud Foundry (CF) Command Line Interface, run the following command:

     ```
     cf update-service <service_instance_name> -c
     '{"Description":"<updated_description_string>",
     "Data":"<Base64_encoded_content_of_.xml_file>"}'
     ```
   - Copy the Base64 encoded output of the content of the XML file to a `<custom-update-config>.json` file along with the updated description string, and then in the CF Command Line Interface, run the following command:

     ```
     cf update-service <service_instance_name> -c <path_of_<custom-
     update-config>.json>
     ```

Bind the Cloud Foundry Application to the Cloud Message Queue Service Instance

About This Task

Important: The angle brackets (< >) in this topic indicate placeholder text. If such placeholder text is part of a command, replace the placeholder text with an appropriate value before running the command.

Procedure

1. Access the Cloud Foundry (CF) Command Line Interface.
2. Run the following command to bind the CF application to the Cloud Message Queue service instance:

   ```
   cf bind-service <application_name> <service_instance_name>
   ```
3. Run the following command to access the list of all environment variables of the CF application:

   ```
   cf env <application_name>
   ```
4. In the list of environment variables, verify whether the Amazon MQ configurations are correctly appearing for the `VCAP_SERVICES` variable.

   Note:
   - The Amazon MQ configurations appear in the `cloud-message-queue` section.

- The user names and passwords of the ActiveMQ broker appear in the `protocols` section and they are same for all the protocols supported by Cloud Message Queue. The same credentials will also be used to authenticate the broker during logging into the Web Active Console using HTTPS.
- The details of the hosts appear in the `hosts` section. There are two hosts to support the ActiveMQ broker in both active and standby modes. The names of the hosts are same for all the protocols supported by Cloud Message Queue.

Example:

```
"VCAP_SERVICES": {
  "cloud-message-queue": [
   {
    "credentials": {
     "configuration": {
      "engine_version": "<ActiveMQ_version>"
    },
    "protocols": {
      "amqp": {
         ...
         "hosts": [
           "<broker_id>-1.mq.us-west-2.amazonaws.com"
          ],
         "password": "broker_password",
         "port": "5671",
         "username": "broker_username"
       },
      "mqtt": {
         ...
         "hosts": [
           "<broker_id>-1.mq.us-west-2.amazonaws.com"
          ],
        "password": "broker_password",
        "port": "8883",
        "username": "broker_username"
      },
      "stomp": {
         ...
         "hosts": [
           "<broker_id>-1.mq.us-west-2.amazonaws.com"
          ],
        "password": "broker_password",
        "port": "61614",
        "username": "broker_username"
     },
    "wss": {
         ...
         "hosts": [
           "<broker_id>-1.mq.us-west-2.amazonaws.com"
          ],
        "password": "broker_password",
        "port": "61619",
        "username": "broker_username"
    }
```

Unbind the Cloud Foundry Application from the Cloud Message Queue Service Instance

Procedure

1. Access the Cloud Foundry (CF) Command Line Interface.
2. Run the following command to unbind the CF application from the Cloud Message Queue service instance:

```
cf unbind-service <application_name> <service_instance_name>
```

Important: The angle brackets (< >) in the command indicate placeholder text. You must replace it with an appropriate value before running the command.

Delete Cloud Message Queue Service Instance

Procedure

1. Access the Cloud Foundry (CF) Command Line Interface.
2. Run the following command to delete the Cloud Message Queue service instance:

```
cf delete-service <service_instance_name>
```

Important: The angle brackets (< >) in the command indicate placeholder text. You must replace it with an appropriate value before running the command.

Chapter 4

Accessing Cloud Message Queue Web Console

Topics:

About Accessing Cloud Message Queue Web Console

The Cloud Message Queue service allows you to access the ActiveMQ Web Console to manage the ActiveMQ brokers.

The Nginx web server is deployed as a Cloud Foundry (CF) application and acts as a reverse proxy for the HTTPS protocol to provide access to the web console for the broker instance.

You can access the Web Console for the following types of ActiveMQ brokers:

- Single-instance broker
- Active/standby broker

Access the Web Console for a Single-Instance Broker

About This Task

Important: In this task, angle brackets (< >) indicate placeholder text. If such placeholder text is part of a command, you must replace it with an appropriate value before running the command.

Procedure

1. Access KBA 000036362, and then download the `nginx_config_single_broker.zip` file.
2. Extract the files from the `nginx_config_single_broker.zip` file to the directory that contains the ZIP file.
3. Open the `nginx_config_single_broker` folder, and then open the `nginx.conf` file using a text editor (for example, Notepad).
4. Locate the following code in the `nginx.conf` file, and then replace the text `HOST` with the name of the host that supports the single-instance broker.

   ```
   location / { proxy_pass https://HOST:8162; proxy_connect_timeout 30s; }
   ```

   Note:

   - The default value of the port is 8162.
   - To get the name of the host for the single-instance broker, refer to the Bind the Cloud Foundry Application to the Cloud Message Queue Service Instance topic.
5. In the directory that contains the `nginx_config_single_broker.zip` file, run the following command to deploy the Nginx Cloud Foundry (CF) application:

   ```
   cf push <APP_NAME> -b https://github.com/cloudfoundry/nginx-
   buildpack.git#v1.0.18 -s cflinuxfs2 -m 128M --random-route
   ```
6. Run the following command to get the route of the deployed CF application:

   ```
   cf app <APP_NAME>
   ```
7. Locate the name of the route in the `routes` parameter of the command output, and then create a URL using the route as displayed in the following example:
   https://<Route Name>
8. Using a web browser, open the URL that you created.
   A login prompt appears.
9. Enter the user name and password to access the Web Console, and then select **Save**.

Note: To obtain the username and password to access the Web Console, refer to the Bind the Cloud Foundry Application to the Cloud Message Queue Service Instance topic.

The Web Console appears.

Access the Web Console for an Active or Standby Broker

About This Task

Important: In this task, angle brackets (< >) indicate placeholder text. If such placeholder text is part of a command, you must replace it with an appropriate value before running the command.

Procedure

1. Access KBA 000036363, and then download the `nginx_config_activestandby_broker.zip` file.
2. Extract the files from the `nginx_config_activestandby_broker.zip` file to the directory that contains the ZIP file.
3. Open the `nginx_config_activestandby_broker.zip` folder, and then open the `nginx.conf` file using a text editor (for example, Notepad).
4. Locate the following code in the `nginx.conf` file, and then replace the text `HOST_1` and `HOST_2` with the names of the hosts that support the ActiveMQ broker in both active and standby modes.

   ```
   upstream backend { server HOST_1:8162 fail_timeout=60s; server HOST_2:8162 fail_timeout=60s; }
   ```

   Note:

   - The default value of the port is 8162.
   - To get the names of the hosts for the active or standby broker, refer to the Bind the Cloud Foundry Application to the Cloud Message Queue Service Instance topic.
5. In the directory that contains the `nginx_config_activestandby_broker.zip` file, run the following command to deploy the Nginx Cloud Foundry (CF) application:

   ```
   cf push <APP_NAME> -b https://github.com/cloudfoundry/nginx-buildpack.git#v1.0.18 -s cflinuxfs2 -m 128M --random-route
   ```

6. Run the following command to get the route of the deployed CF application:

   ```
   cf app <APP_NAME>
   ```

7. Locate the name of the route in the `routes` parameter of the command output, and then create a URL using the route as displayed in the following example:
   https://<Route Name>
8. Using a web browser, open the URL that you created.
   A login prompt appears.
9. Enter the user name and password to access the Web Console, and then select **Save**.

   Note: To obtain the username and password to access the Web Console, refer to the Bind the Cloud Foundry Application to the Cloud Message Queue Service Instance topic.

   The Web Console appears.

Chapter 5

Using Cloud Message Queue Service

Topics:

About Using Cloud Message Queue Service

Using the Cloud Message Queue service, you can connect your application with the ActiveMQ broker and create a message queue to send or receive messages. The subsequent topics provide code examples for working with Cloud Message Queue in the following programming languages:

- Java
- GO
- Ruby

Tip:

- The transport endpoint for ActiveMQ Advanced Message Queuing Protocol (AMQP) starts with `amqp+ssl`. When creating an AMQP client, you must replace `amqp+ssl` with `amqps` in your application code.
- If you choose to subscribe to a High Availability (HA) plan, you must use the Failover Transport protocol. For more information on the Failover Transport protocol, refer to the ActiveMQ documentation.

Connect to ActiveMQ Broker using Java Message Service (JMS)

Before You Begin

- Ensure that Java is installed and configured in your system.
- In the project directory, modify the `pom.xml` file as follows to add the `activemq-client.jar` and `activemq-pool.jar` packages to your Java class path:

```
<dependencies>
    <dependency>
        <groupId>org.apache.activemq</groupId>
        <artifactId>activemq-client</artifactId>
        <version>5.15.8</version>
    </dependency>
    <dependency>
        <groupId>org.apache.activemq</groupId>
        <artifactId>activemq-pool</artifactId>
        <version>5.15.8</version>
    </dependency>
</dependencies>
```

About This Task

Cloud Message Queue supports JMS, which allows you to create message queues, and send or receive messages within a distributed communication system that is loosely coupled, reliable, and asynchronous. This topic describes how to connect your application with the message broker using JMS.

Procedure

1. Create a message producer to send messages.
2. Create a message consumer to receive messages.

Create Message Producer using Java Message Service (JMS)

Before You Begin

- Ensure that Java is installed and configured in your system.
- In the project directory, ensure that the `pom.xml` file is modified as follows to add the `activemq-client.jar` and `activemq-pool.jar` packages to your Java class path:

```
<dependencies>
    <dependency>
        <groupId>org.apache.activemq</groupId>
        <artifactId>activemq-client</artifactId>
        <version>5.15.8</version>
    </dependency>
    <dependency>
        <groupId>org.apache.activemq</groupId>
        <artifactId>activemq-pool</artifactId>
        <version>5.15.8</version>
    </dependency>
</dependencies>
```

About This Task

This topic describes how to create a message producer using JMS. Additionally, you can create a connection to the message broker, create a queue, and send messages to the message consumer.

Notice: This topic provides code examples to demonstate how to create a message producer using JMS.

Procedure

1. Run the following command to add the library objects to the ActiveMQ web project:

```
import javax.jms.Connection; import javax.jms.ConnectionFactory;
import javax.jms.DeliveryMode; import javax.jms.Destination; import
javax.jms.JMSException; import javax.jms.Message; import
javax.jms.MessageConsumer; import javax.jms.MessageProducer; import
javax.jms.MessageListener; import javax.jms.Session; import
javax.jms.TextMessage; import javax.naming.Context; import
org.apache.activemq.jms.pool.PooledConnectionFactory;
```

2. Create a connection factory for the message producer using the ActiveMQ endpoint.

```
Hashtable<Object, Object> env = new Hashtable<Object, Object>();
env.put(Context.INITIAL_CONTEXT_FACTORY,
"org.apache.qpid.jms.jndi.JmsInitialContextFactory");
env.put("connectionfactory.factoryLookup", endpoint);
javax.naming.Context context = new javax.naming.InitialContext(env);

// Create a connection factory.
ConnectionFactory connectionFactory = (ConnectionFactory)
context.lookup("factoryLookup");

// Create a pooled connection factory.
PooledConnectionFactory pooledConnectionFactory = new
PooledConnectionFactory();
pooledConnectionFactory.setConnectionFactory(connectionFactory);
pooledConnectionFactory.setMaxConnections(10);
```

```
// Establish a connection for the producer.
Connection producerConnection =
pooledConnectionFactory.createConnection(username, password);
producerConnection.start();
```

3. Create a session, a message queue, and a message producer to send messages to the destination application.
   The following code example will create a message queue named `QUEUE`:

```
// Create a session.
Session producerSession = producerConnection.createSession(false,
ACKNOWLEDGE_MODE);

// Create a queue named "QUEUE".
Destination producerDestination = producerSession.createQueue(QUEUE);

// Create a producer from the session to the queue.
MessageProducer producer =
producerSession.createProducer(producerDestination);
producer.setDeliveryMode(DELIVERY_MODE);
```

4. Register the message producer to send messages to the destination application.

```
long totalMillis  = 0;
while (totalMillis < RUN_MILLISECONDS) {
    // Generate a random UUID to use for the message
    String message =  UUID.randomUUID().toString();
    // byte[] messageBodyBytes = message.getBytes();
    System.out.println(String.format("Publishing message to exchange
%s: %s", QUEUE, message));
    TextMessage producerMessage =
producerSession.createTextMessage(message);
    // Send the message.
    producer.send(producerMessage);

    Thread.sleep(INTERVAL);
    totalMillis += INTERVAL;
}
```

5. Close the producer, session, and connection.

```
producer.close();
producerSession.close();
producerConnection.close();
pooledConnectionFactory.stop();
```

Create Message Consumer using Java Message Service (JMS)

Before You Begin

- Ensure that Java is installed and configured in your system.
- In the project directory, ensure that the `pom.xml` file is modified as follows to add the `activemq-client.jar` and `activemq-pool.jar` packages to your Java class path:

```
<dependencies>
    <dependency>
        <groupId>org.apache.activemq</groupId>
        <artifactId>activemq-client</artifactId>
        <version>5.15.8</version>
```

```
    </dependency>
    <dependency>
        <groupId>org.apache.activemq</groupId>
        <artifactId>activemq-pool</artifactId>
        <version>5.15.8</version>
    </dependency>
</dependencies>
```

About This Task

This topic describes how to create a message consumer using JMS. Additionally, you can create a connection to the message broker, create a queue, and receive messages from the ActiveMQ message queue.

Notice: This topic provides code examples to demonstate how to create a message consumer using JMS.

Procedure

1. Run the following command to add the library objects to the ActiveMQ web project:

```
importjavax.jms.Connection; importjavax.jms.ConnectionFactory;
importjavax.jms.DeliveryMode; importjavax.jms.Destination;
importjavax.jms.JMSException; importjavax.jms.Message;
importjavax.jms.MessageConsumer; importjavax.jms.MessageProducer;
importjavax.jms.MessageListener; importjavax.jms.Session;
importjavax.jms.TextMessage; importjavax.naming.Context;
importjavax.naming.NamingException;
```

2. Create a message listener to receive the messages asynchonously.

```
class ConsumerMessageListener implements MessageListener {
    public void onMessage(Message message) {
        TextMessage textMessage = (TextMessage) message;
        try {
            System.out.println("Message received : " +
textMessage.getText());
        } catch (JMSException e) {
            e.printStackTrace();
        }
    }
}
```

3. Create a connection factory for the message consumer using the endpoint of the ActiveMQ message broker.

```
Hashtable<Object, Object> env =newHashtable<Object, Object>();
env.put(Context.INITIAL_CONTEXT_FACTORY,
"org.apache.qpid.jms.jndi.JmsInitialContextFactory");
env.put("connectionfactory.factoryLookup", endpoint);
javax.naming.Context context =newjavax.naming.InitialContext(env);

// Create a connection factory.ConnectionFactory connectionFactory =
(ConnectionFactory) context.lookup("factoryLookup");

// Establish a connection for the consumer.Connection
consumerConnection = connectionFactory.createConnection(username,
password);
consumerConnection.start();
```

4. Create a session, a message queue, and a message consumer to receive messages from the ActiveMQ message queue.
   The following code example will create a message queue named `QUEUE`:

```
// Create a session.
Session consumerSession = consumerConnection.createSession(false,
ACKNOWLEDGE_MODE);

// Create a queue named "QUEUE".
Destination consumerDestination = consumerSession.createQueue(QUEUE);

// Create a message consumer from the session to the queue.
MessageConsumer consumer =
consumerSession.createConsumer(consumerDestination);
consumer.setMessageListener(new ConsumerMessageListener());
Thread.sleep(100000);
```

5. Close the consumer, session, and connection.

```
consumer.close();
consumerSession.close();
consumerConnection.close();
```

Connect to ActiveMQ Broker with Golang

Before You Begin

Run the following command to import the library package for AMQP 1.0 into your application.

```
import "pack.ag/amqp"
```

About This Task

The Cloud Message Queue service supports AMQP 1.0, which allows you to connect your application with the message queue broker, create message queues, and send or receive messages.

Code Examples for Connecting to ActiveMQ Broker using AMQP 1.0

- The following code example demonstrates how to create a message producer, and then send messages to the destination application using the message queue:

```
// Create session
client, err := amqp.Dial("amqps://"+host+":"+port,
    amqp.ConnSASLPlain(username, password),
)
if err != nil {
    fmt.Println(fmt.Sprintf("Dialing AMQP server:",
err))
}
defer client.Close()

// Open a session
session, err := client.NewSession()
ctx := context.Background()

queue := "test-queue"
```

```
sender, err := session.NewSender(
    amqp.LinkTargetAddress(queue),
)
if err != nil {
    fmt.Println(fmt.Sprintf("Cannot create session,
error:", err))
}

ctx, cancel := context.WithTimeout(ctx, 30*time.Second)
Sending messages to queue:

totalSeconds := 0
maxSeconds := 30
for totalSeconds < maxSeconds {
    // Generate random UUID for message body. Convert
to []byte
    randomUUID := uuid.NewV4()
    body := []byte(randomUUID.String())

    // Publish a message
    output := fmt.Sprintf("Published message to test-
queue: %s", body)
    fmt.Println(output)

    err = sender.Send(ctx,
amqp.NewMessage([]byte(body)))
    if err != nil {
        fmt.Println(fmt.Sprintf("Failed to publish
message, error:", err))
    }
    time.Sleep(5 * time.Second)
    totalSeconds += 5
}

// Close connection
sender.Close(ctx)
cancel()
```

- The following code example demonstrates how to create a message receiver, and then receive messages using the message queue:

```
// Create client
client, err := amqp.Dial("amqps://"+host+":"+port,
    amqp.ConnSASLPlain(username, password),
)
if err != nil {
    fmt.Println(fmt.Sprintf("Dialing AMQP server:",
err))
}
defer client.Close()

// Open a session
session, err := client.NewSession()
if err != nil {
    fmt.Println(fmt.Sprintf("Cannot create session,
error:", err))
}
ctx := context.Background()

queue := "test-queue"
```

```
Create receiver and receive mesages

// Create a receiver
receiver, err := session.NewReceiver(
    amqp.LinkSourceAddress(queue),
    amqp.LinkCredit(10),
)
if err != nil {
    fmt.Println(fmt.Sprintf("Creating receiver link,
error:", err))
}

defer func() {
    ctx, cancel := context.WithTimeout(ctx,
35*time.Second)
    receiver.Close(ctx)
    cancel()
}()

for {
    // Receive next message
    msg, err := receiver.Receive(ctx)
    if err != nil {
        fmt.Println(fmt.Sprintf("Reading message from
AMQP error:", err))
    }

    // Accept message
    msg.Accept()

    fmt.Println(fmt.Sprintf("Message received: %s\n",
msg.GetData()))
}
```

About Connecting the ActiveMQ Broker with Ruby

The Cloud Message Queue service supports the following communication protocols:

- Simple (or Streaming) Text Oriented Messaging Protocol (STOMP)
- Message Queuing Telemetry Transport (MQTT)

If your application is built on Ruby, you can use one of these protocols to connect to the ActiveMQ broker and send or receive messages.

Note: All other transport protocols except AMQP 1.0 are supported with Ruby. Currently, no official Ruby library release supports AMQP 1.0.

Connect to ActiveMQ Broker using Simple (or Streaming) Text Oriented Messaging Protocol (STOMP)

Before You Begin

Run the following command to import the STOMP library packages to your application:

```
require 'stomp'
```

About This Task

The Cloud Message Queue service supports STOMP, which allows you to connect your application with the message queue broker, and send or receive messages.

Notice: This topic provides code examples to demonstate how to connect your application with the message queue broker using STOMP.

Procedure

1. Run the following command to define the SSL client certificate components (that are, public key and private certificate) in the Privacy Enhanced Mail (PEM) file:

```
openssl req -newkey rsa:2048 -nodes -keyout privateKey.key -x509 -
days 365 -out certificate.pem

ssl_opts = Stomp::SSLParams.new(

:key_file =>
"#{File.expand_path(File.dirname(File.dirname(__FILE__)))}/
certificate/privateKey.key",

:cert_file =>
"#{File.expand_path(File.dirname(File.dirname(__FILE__)))}/
certificate/certificate.pem",

:fsck => true,

:ciphers => ciphers_list

)
```

Note: The following list of ciphers are supported with the STOMP client:

```
ciphers_list = [

["DHE-RSA-AES256-SHA", "TLSv1/SSLv3", 256, 256],

["DHE-DSS-AES256-SHA", "TLSv1/SSLv3", 256, 256],

["AES256-SHA", "TLSv1/SSLv3", 256, 256],

["EDH-RSA-DES-CBC3-SHA", "TLSv1/SSLv3", 168, 168],

["EDH-DSS-DES-CBC3-SHA", "TLSv1/SSLv3", 168, 168],

["DES-CBC3-SHA", "TLSv1/SSLv3", 168, 168],

["DHE-RSA-AES128-SHA", "TLSv1/SSLv3", 128, 128],

["DHE-DSS-AES128-SHA", "TLSv1/SSLv3", 128, 128],

["AES128-SHA", "TLSv1/SSLv3", 128, 128],

["RC4-SHA", "TLSv1/SSLv3", 128, 128],

["RC4-MD5", "TLSv1/SSLv3", 128, 128],

["EDH-RSA-DES-CBC-SHA", "TLSv1/SSLv3", 56, 56],
```

```
["EDH-DSS-DES-CBC-SHA", "TLSv1/SSLv3", 56, 56],

["DES-CBC-SHA", "TLSv1/SSLv3", 56, 56],

["EXP-EDH-RSA-DES-CBC-SHA", "TLSv1/SSLv3", 40, 56],

["EXP-EDH-DSS-DES-CBC-SHA", "TLSv1/SSLv3", 40, 56],

["EXP-DES-CBC-SHA", "TLSv1/SSLv3", 40, 56],

["EXP-RC2-CBC-MD5", "TLSv1/SSLv3", 40, 128],

["EXP-RC4-MD5", "TLSv1/SSLv3", 40, 128]

]
```

2. Create the STOMP client and connect the client with your application.

```
hash = {

:hosts => [

   {

     :login => stomp_credentials['username'],

     :passcode => stomp_credentials['password'],

     :host => stomp_credentials['host'],

     :port => stomp_credentials['port'],

     :ssl => ssl_opts

   }

],

:reliable => false,

:start_timeout => 0,

:connect_headers => {"accept-version" => "1.1,1.2", "host" =>
"vhost", "heart-beat" => "5000,10000" }

}

STDOUT.puts "Creating client"


client = Stomp::Client.new(hash)

conn_frame = client.connection_frame()
```

3. Ensure that the application and the STOMP client are connected.

```
if conn_frame.command == Stomp::CMD_ERROR

STDOUT.puts "Unable to connect...\n"
```

```
raise conn_frame.body

end

puts "-" * 50

puts "Connect version - #{conn_frame.headers['version']}"

puts "Connect server   - #{conn_frame.headers['server']}"

puts "Session ID       - #{conn_frame.headers['session']}"

puts "Heartbeats       - #{conn_frame.headers['heart-beat']}"

puts "SSL Verify       - #{ssl_opts.verify_result}"

puts "-" * 50 , "\n\n"
```

4. Perform the tests as specified in the following examples to ensure that the messages are sent and received from the message queue.
   The following code example demonstrates how to send the message text, `Hello World!`, to the message queue:

   ```
   client.publish( "/queue/myqueue", "Hello World!")
   ```

   The following code example demonstrates how to subscribe to a destination application to receive messages from the message queue:

   ```
   begin

   client.subscribe("/queue/myqueue", {:ack => "client",
   "activemq.prefetchSize" => 10000,:browser => "false"} ) do |message|

      STDOUT.puts "Received message from the queue: #{message.body}"

   end

   rescue Interrupt => _

   client.close

   end
   ```

Connect to ActiveMQ Broker using Message Queuing Telemetry Transport (MQTT)

Before You Begin

Run the following command to import the MQTT library packages to your application:

```
require 'mqtt'
```

About This Task

The Cloud Message Queue service supports the MQTT protocol, which allows you to connect your application with the message queue broker, and send or receive messages.

Notice: This topic provides code examples to demonstate how to connect your application with the message queue broker using the MQTT protocol.

Procedure

1. Run the following command to define the SSL client certificate components (that are, public key and private certificate) in the Privacy Enhanced Mail (PEM) file:

```
openssl req -newkey rsa:2048 -nodes -keyout privateKey.key -x509 -
days 365 -out certificate.pem
```

2. Create the MQTT client and connect the client with your application.

```
client = MQTT::Client.new

client.host = mqtt_credentials['host']

client.username = mqtt_credentials['username']

client.password = mqtt_credentials['password']

client.port = mqtt_credentials['port']

client.ssl = true

client.cert_file =
"#{File.expand_path(File.dirname(File.dirname(__FILE__)))}/
certificate/certificate.pem"

client.key_file =
"#{File.expand_path(File.dirname(File.dirname(__FILE__)))}/
certificate/privateKey.key"
```

3. Perform the tests as specified in the following examples to ensure that the messages are sent and received from the message queue.
   The following code example demonstrates how to send a message to the message queue:

```
client.connect do

   client.publish(queue, message)

end
```

   The following code example demonstrates how to subscribe to a destination application to receive messages from the message queue:

```
client.connect do

client.subscribe(queue)

# wait 10 messages

10.times.each do

   _topic, message = client.get

   STDOUT.puts "Received message from the queue: #{message}"

end
```

```
end
```

Chapter 6

Migrating to Cloud Message Queue

Topics:

About Migrating to Cloud Message Queue

Cloud Message Queue is designed to replace Predix Message Queue. The Predix Message Queue service will no longer be available in future and therefore, GE Digital recommends you migrate to the Cloud Message Queue service before the Predix Message Queue service is deprecated.

Migrate from Predix Message Queue to Cloud Message Queue

Before You Begin

- Analyze and benchmark your workload to identify which Cloud Message Queue subscription plan is appropriate for your Cloud Foundry (CF) application. If you choose to subscribe to a High Availability (HA) plan, you must use the Failover Transport protocol. For more information on the Failover Transport protocol, refer to the ActiveMQ documentation.
- Identify the protocol that is most suitable for your team. AmazonMQ supports Advanced Message Queuing Protocol (AMQP), Simple (or Streaming) Text Orientated Messaging Protocol (STOMP), OpenWire, WebSocket, and MQ Telemetry Transport (MQTT). However, you can choose a protocol based on your programming language.
- Analyze whether persistent or non-persistent messaging is most suitable for your CF application.
- Analyze whether the messages will be consumed by fast or slow consumers, and then based on that information, decide on the durability of the messages.

About This Task

This topic describes the steps that you must perform to migrate from Predix Message Queue to Cloud Message Queue.

Important: In case of connection failure during the migration, utilize the design patterns for auto-recovery and to avoid a deadlock or race condition.

Procedure

1. Create the Cloud Message Queue service instance for your preferred subscription plan.
2. Ensure that all messages that were scheduled to be delivered by the RabbitMQ broker have been delivered and processed according to the logic of your CF application.
3. Migrate from RabbitMQ to ActiveMQ.
4. Deploy the CF application using the Blue-Green deployment method to reduce application downtime and risk. For more information on Blue-Green deployment, refer to the Cloud Foundry documentation.
5. Run your test cases to verify whether the CF application is correctly deployed, and then switch the route mapping.

Create Cloud Message Queue Service Instance

About This Task

You must create a new service instance for the Cloud Message Queue plan that you want to subscribe.

Important: In this topic, angle brackets (< >) indicate placeholder text. If such placeholder text is part of a command, you must replace it with an appropriate value before running the command.

Procedure

1. Access the Cloud Foundry (CF) Command Line Interface.
2. Run the following command to verify whether Cloud Message Queue is available in the CF marketplace of your organization:

```
cf m
```

3. Run the following command to access the list of Cloud Message Queue plans that are available in the CF marketplace for subscription:

```
cf m -s cloud-message-queue
```

4. Run the following command to create the service instance:

```
cf create-service cloud-message-queue <subscription_plan_name>
<instance_name>
```

5. Wait for 15-20 minutes, and then run the following command to check the status of the service instance:

```
cf service <instance_name>
```

Migrate from RabbitMQ to ActiveMQ

About This Task

Predix Message Queue uses RabbitMQ and Cloud Message Queue uses ActiveMQ as the underlying engine. Therefore, migrating from RabbitMQ to ActiveMQ is an important step while migrating from Predix Message Queue to Cloud Message Queue.

The ActiveMQ supports Advanced Message Queuing Protocol (AMQP) v1.0, whereas RabbitMQ supports AMQP v0.9.1. There are major differences between these protocols and because of these differences, you must make some modifications in the code to migrate from RabbitMQ to ActiveMQ.

Refer to the following documentation for more information on the modifications that you must make in the code:

- Amazon Web Services (AWS) documentation
- Amazon MQ Workshop documentation

Important: Ignore the procedure of creating the message broker that is provided in the AWS and Amazon MQ Workshop documentation. The procedure is not valid for Cloud Message Queue.

Chapter 7

Security

Topics:

About Cloud Message Queue Security

The Cloud Message Queue service supports basic authentication using a unique user name and password for each service instance. After binding the service to the Cloud Foundry (CF) application, you can obtain the user name and password from the `VCAP_SERVICES` environment variable.

Additionally, at the Infrastructure as a Service (IaaS) level, all Cloud Message Queue instances are provisioned within private subnets and are not accessible directly over the internet. You can only access them through applications on the CF platform.

Chapter 8

Backup

Topics:

About Backing up Messages

The Cloud Message Queue service instances use Amazon Elastic File System (Amazon EFS) for storing messages. Amazon MQ redundantly stores each message in multiple Availability Zones. This acts as a secured backup of messages and prevents any loss of information.

Note: Cloud Message Queue does not support any additional backup process.

Chapter 9

Retrieving Broker and Destination Metrics

Topics:

About Retrieving Metrics for the Message Queue Broker and Destination Applications

As a Cloud Foundry (CF) user, you can use the Statistics Plugin integrated with ActiveMQ to query and retrieve statistical data for the message queue broker, message queue, and destination applications. The statistical data contains various metrics, such as size of the messages, average time of the messages in the queue, the usage of application memory, number of producers and consumers, and so on. For more information on the Statistics Plugin, refer to the Apache ActiveMQ documentation.

Retrieve Metrics for Broker and Destination Applications

Before You Begin

- In the ActiveMQ XML configuration file, add the following broker plugin using the custom configuration:

```
<statisticsBrokerPlugin/>
```

Note: It is recommended to add the plugin when creating or updating the Cloud Message Queue service instance. For more information on how to apply custom configurations when creating or updating the service instances, refer to the Create Service Instance with Customized Configurations on page 7 or Update Cloud Message Queue Service Instance on page 7 topics, respectively.

Example: Retrieving Metrics for the Message Broker and Destination Applications

The following code example retrieves metrics for the message broker and destination applications:

```
import java.util.Enumeration;
import javax.jms.Connection;
import javax.jms.MapMessage;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageProducer;
import javax.jms.Queue;
import javax.jms.QueueBrowser;
import javax.jms.Session;
import org.apache.activemq.ActiveMQConnectionFactory;
public class StatisticsPluginData {
 private static String AMQ_ENDPOINT = "ssl://
hostname:61617";
 private static String ACTIVE_MQ_USERNAME = "user";
    private static String ACTIVE_MQ_PASSWORD = "pwd";
    private String DESTINATION_NAME = "testqueue";
    private String DESTINATION_TYPE = "queue";
    private int msgCount = 2;

 public StatisticsPluginData(String destinationName,
String destinationType, String amqEndpoint, String
username, String password, int msgCount) {
  super();
  this.DESTINATION_NAME = destinationName;
  this.DESTINATION_TYPE = destinationType;
```

```
  StatisticsPluginData.AMQ_ENDPOINT = amqEndpoint;
  StatisticsPluginData.ACTIVE_MQ_PASSWORD = password;
  StatisticsPluginData.ACTIVE_MQ_USERNAME = username;
  this.msgCount = msgCount;
 }

 private static ActiveMQConnectionFactory
createActiveMQConnectionFactory() {
        // Create a connection factory.
        final ActiveMQConnectionFactory
connectionFactory =
                new
ActiveMQConnectionFactory(AMQ_ENDPOINT);
        // Pass the username and password.

connectionFactory.setUserName(ACTIVE_MQ_USERNAME);

connectionFactory.setPassword(ACTIVE_MQ_PASSWORD);
        return connectionFactory;
    }

 public void getQueueStatistics()
 {
  Session session = null;
  Connection connection = null;
  ActiveMQConnectionFactory connectionFactory = null;
  Queue statisticsDestination = null;
  Queue replyTo;
  MessageConsumer consumer = null;
  MessageProducer producer = null;
  Message msg;
  String statisticsQueue;
  MapMessage statisticsMsg;
  Enumeration qEnum;
  String statistic;

  try
  {
   connectionFactory = createActiveMQConnectionFactory();
   connection = connectionFactory.createConnection();
   connection.start();

   session = connection.createSession(false,
Session.AUTO_ACKNOWLEDGE);
   replyTo = session.createTemporaryQueue();

   consumer = session.createConsumer(replyTo);
   producer = session.createProducer(null);

   statisticsQueue = "ActiveMQ.Statistics.Destination."
+ this.DESTINATION_NAME;

   statisticsDestination =
session.createQueue(statisticsQueue);
   msg = session.createMessage();
   msg.setJMSReplyTo(replyTo);
   producer.send(statisticsDestination,msg);

   statisticsMsg = (MapMessage)consumer.receive();
   System.out.println("Printing
Statistics.........................");
```

```
   for(qEnum = statisticsMsg.getMapNames();
qEnum.hasMoreElements();)
   {
    statistic = qEnum.nextElement().toString();
    System.out.println(statistic + " -- " +
statisticsMsg.getObject(statistic));
   }

  }
  catch(Exception e)
  {
   System.out.println("Error msg - QueueStatistics
Failure: ..........."+e.getMessage());
   e.printStackTrace();
  }
 }
}
```

Chapter 10

Reference

Topics:

Performance Benchmarking

The following table lists the results of the performance benchmarking that is performed on the Cloud Message Queue service for each subscription plan using the Java client and OpenWire protocol.

Plan	Description	Requires Subscription?	Model	Maximum No. of Connections	Maximum No. of Messages/Sec
Dedicated-1-Q1	Single node, 1GB RAM, 1vCPU. Economical option for lightest workloads in development	Yes	mq.t2.micro	100	23,000
Dedicated-1-Q10	Single node, 8GB RAM, 2vCPU. Economical option for light workloads in development	Yes	mq.m5.large	1,000	50,000
Dedicated-1-Q20	Single node, 16GB RAM, 4vCPU. Ideal for medium workloads in development	Yes	mq.m5.xlarge	1,000	90,000
Dedicated-1-Q30	Single node, 32GB RAM, 8vCPU. Ideal for heavy workloads in development	Yes	mq.m5.2xlarge	1,000	150,000
Dedicated-2HA-Q1	Two node, highly available, 1GB RAM, 1vCPU. Economical option for lightest workloads in development	Yes	mq.t2.micro	100	23,000
Dedicated-2HA-Q10	Two node, highly available, 8GB RAM, 2vCPU. Economical option for light workloads in production environment	Yes	mq.m5.large	1,000	50,000

Plan	Description	Requires Subscription?	Model	Maximum No. of Connections	Maximum No. of Messages/Sec
Dedicated-2HA-Q20	Two node, highly available, 16GB RAM, 4vCPU. Ideal for medium workloads in production environment	Yes	mq.m5.xlarge	1,000	90,000
Dedicated-2HA-Q30	Two node, highly available, 32GB RAM, 8vCPU. Ideal for heavy workloads in production environment	Yes	mq.m5.2xlarge	1,000	150,000

Chapter 11

Release Notes

Topics:

Third Quarter of 2020

V1.0

This topic provides a list of product changes released for this service in this version.

Table 1: Enhancements and New Features

The following enhancements and new features have been added.

Description	Tracking ID
A new messaging service, Cloud Message Queue, has been introduced that provides the ability to exchange messages between different applications, components, and devices in a highly durable and reliable manner. The Cloud Message Queue service offers the following features and capabilities: • **Platform**: Backed by Amazon MQ, which is a managed message broker service for Apache ActiveMQ • **User defined configuration**: Allows the users to have full control on the broker configuration (for example, engine version, destination-specific attributes, enabling statistics, broker plug-ins, and so on). The users can pass the desired configuration while creating and updating instances. • **Subscription plans**: Provides eight subscription plans that support a range of instance models from mq.t2.micro to mq.m5.2xlarge in both dedicated and High Availability (HA) configurations • **Supported protocols**: Supports various wire-level protocols, such as Advanced Message Queuing Protocol (AMQP) 1.0, Simple (or Streaming) Text Oriented Messaging Protocol (STOMP), OpenWire, WebSocket, and MQ Telemetry Transport (MQTT) • **Supported programming languages**: Supports a wide range of programming languages, including Node.js, Go, Python, Ruby, and C++ • **Supported clients**: Supports a wide range of clients, including Java Message Service (JMS) 1.1 and .NET Message Service (NMS)	F52595