Using IBM Cloud Functions for serverless processing of MQ messages

Matt_Roberts
Published on 18/10/2017

Serverless computing is a runtime programming model in which application developers write business logic without the worry of managing the server infrastructure on which their code runs, and while only paying for the compute resources used while the application is actively processing requests. These twin benefits of reduced complexity and lower costs mean that "serverless" (or "function-as-a-service") is a rapidly growing domain, as evidenced by the proliferation of frameworks such as AWS Lambda, Azure Functions and IBM Cloud Functions (the latter built from the recently open-sourced Apache OpenWhisk).

The value of any new programming model is determined by the range of application scenarios that it can support, and since most companies have existing services and infrastructure on which their businesses are built it is critical that serverless computing can be integrated with other components of the IT landscape.

In this blog post, we will demonstrate how to integrate IBM Cloud Functions with IBM MQ to process messages arriving on an MQ queue – extending the serverless model to interact with this widely-used enterprise middleware platform.

Cloud Functions architectural patterns

There are three architectural patterns for creating a feed of events in IBM Cloud Functions (ICF). These are called 'Hooks', 'Polling', and 'Connections'.

Since IBM MQ does not provide an out-of-the-box WebHook capability for notifying us when a message arrives (Hooks), and since we want to avoid having to manage a long-running application (Connections), this blog will show the Polling pattern, in which we register an Alarm that periodically triggers our ICF "Action" to check whether there are any messages ready to be processed.

These are the steps that we will follow in this tutorial:

1. Download and configure the necessary tools and sample code
2. Build and test the IBM Cloud Functions "Action" that will process our messages
3. Upload the Action to Bluemix and configure an Alarm and Rule so that it is invoked
4. Put some sample messages to the queue to see our Action in action!

1. Download and configure the necessary tools and sample code

Cloud Functions CLI

To get started with IBM Cloud Functions, we will install the Cloud Functions CLI plugin and invoke a sample action to confirm that it has been configured correctly. If you already have the Bluemix CLI installed, you can install and test the Cloud Functions CLI:

```
# Install the Cloud Functions CLI plugin
bx plugin install Cloud-Functions -r Bluemix

# Log in to Bluemix (use the "--sso" option if you have a federated ID)
# Replace "YourOrgName" and "YourSpaceName" with the appropriate details of your account
bx login -a api.eu-gb.bluemix.net -o YourOrgName -s YourSpaceName

# Invoke this pre-configured action to test that your environment is set up correctly.
# You should see your message "hello" returned back to you in the response
bx wsk action invoke /whisk.system/utils/echo -p message hello --result

{
  "message": "hello"
}
```

Sample code for ICF Action

For the purpose of this tutorial, we have provided a sample project that implements the ICF Action. You can download the project from GitHub:

```
# Download the sample project from GitHub into a directory called "tutorial"
git clone https://github.com/ibm-messaging/mq-ibmcloud.git tutorial
```

We will walk through the content of the sample code when we build and test the Action in the next section.

The sample project is configured to build using Gradle, so if you don’t have it installed already then you should install Gradle using the appropriate instructions for your platform – for example, on a Mac:

```
brew update && brew install gradle
```

IBM MQ JMS client libraries

Since the Action we are using needs to talk to an IBM MQ queue manager we must download the necessary client libraries and make them available to be used as part of the sample project. The sample project is written in Java, so we will download the IBM MQ JMS client libraries:

- Go to IBM FixCentral and search for “WebSphere MQ”, version 9.0, platform “All”
- Select the “Text” search option, enter “Java” in the text field and click Continue
- Select and download the “IBM MQ JMS and Java redistributable client”, for example “9.0.0.2-IBM-MQC-Redist-Java “
- Once downloaded, extract the zip file into a temporary directory

- Copy the jar files from the root of the extracted directory into the /javaAction/lib directory of the sample project (eg “com.ibm.mq.allclient.jar”, “jms.jar”)
    - Note that Bouncy Castle jars (matching the pattern “bcp*.jar”) are typically not required unless your application intends to use the MQ Advanced Message Security (AMS) feature.

Internet accessible queue manager

The sample code used in this tutorial requires a queue manager that is publicly visible over the internet so that the code executing in IBM Cloud Functions can connect to it. If you don’t have a queue manager that is accessible over the public Internet, you can launch one by following the instructions in the blog post on Running the MQ docker image on the Kubernetes service in Bluemix.

2. Build and test the IBM Cloud Functions “Action” that will process our messages

To create an ICF Action in Java, we must build a self-contained jar file with a class that exposes a particular method signature as described in the docs page for Creating Java actions. The sample project has been set up to do this for you – test that the build process works correctly by using the “gradle build” command to execute the build, and then we will look into the details of how the sample project works:

```
cd tutorial/cloudfunctions/javaAction/
gradle build

:compileJava
:processResources UP-TO-DATE
:classes
:jar
:assemble
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test UP-TO-DATE
:check UP-TO-DATE
:build

BUILD SUCCESSFUL

Total time: 2.112 secs
```

The core part of the sample project is a Java class file called ProcessMQMessages.java which you will find in the “/src/main/java” directory. At line 62 of the class you’ll see the special “main method” that allows this class to be executed as an ICF/OpenWhisk Action – note that it has a slightly different signature to the traditional J2SE “main method”, which takes a String[] as a parameter, whereas in this case the parameter is a JsonObject.

```
public static JsonObject main(JsonObject args) {
```

If you review the Java code contained in that method, you'll see the following main steps:

- Create a JMS connection to the queue manager
- Create a JMS session and a MessageConsumer object to allow us to read messages from the queue
- Inside a while loop, read a message from the queue and apply the necessary business logic to process the message
    - The business processing is applied in the "processMessage" method – in the sample project you'll see a TODO marker at line 208 indicating you should add your custom logic. For the purposes of this demonstration all the code does is print a message saying that it has successfully processed the message.
- Repeat the previous step until there are no further messages to be read from the queue at this time
- Return a response object with a property indicating how many messages have been processed.

Note on building a runnable IBM MQ client jar:

IBM Cloud Functions (and equivalently OpenWhisk) requires that a single executable jar file (including a named class that implements the "main" method above) is provided for Java Actions. To connect to IBM MQ, that executable jar file must contain the IBM MQ client libraries that we applied in the previous section. However, Java doesn't support nesting jars within jars (see the note here), and so the build process above creates a "fat jar" in which the dependency jars are unzipped and repacked as individual classes into the executable jar.

This approach means that we can provide all the necessary logic to IBM Cloud Functions within a single executable jar, and has not caused any functional problems during the work carried out for this tutorial. However this is not a configuration that is formally supported by IBM, and so consideration should be given before using this approach in a production scenario. A formally supported way of achieving the same end goal with IBM Cloud Functions is to package the application logic as a Docker Action which allows you to run in a standard operating system environment and so can use the MQ client jars without alteration, however this is a more heavyweight approach to achieving the end goal. We're interested to hear your feedback on this area.

Passing default parameters to the Action

To configure the Action to meet the needs of our specific scenario, we must provide it a series of parameters that define things like the hostname, the port of the queue manager that it should connect to, the queue manager name and, the name of the queue that it should consume from. We could embed the necessary values as hardcoded strings in our Java code, but that means we would have to build a different copy of the jar for every queue manager/queue that we want to connect to. Instead it is preferable to pass in the configuration parameters at runtime.

OpenWhisk (and thus IBM Cloud Functions) allows parameters to be passed to an Action by specifying one or more "–param" arguments when you invoke the Action. The Action will then be made available inside the "args" parameter of the main method shown above. We'll talk about how these parameters are applied in Bluemix in the next section, but for now we want to start by testing the processing logic locally:

Testing the processing locally

In order to minimize the cycle time between us making a code change and receiving feedback about whether it has been successful, it is desirable to be able to execute our Action logic locally before we think about uploading it to Bluemix. To achieve this goal, the sample code also includes a J2SE "main method" towards the bottom of the file that we can use to execute the logic standalone.

In this standalone mode the necessary parameters are defined by reading in a file called configuration.json which allows the sample code to set up a call to the OpenWhisk Action method in the same way it will be invoked when deployed to Bluemix;

```
# Update the configuration.json file in the sample project to provide the
# required details such as queue manager hostname / port, queue name etc.
```

Once you have updated the configuration.json file, you can test the application locally by running the following commands:

```
# Test the connectivity to the queue manager by invoking the Action locally using
# this command
java -jar ./build/libs/ProcessMQMessages.jar ProcessMQMessages

Simulating execution of Action
{"messagesProcessed":0}
End of simulation.

# The sample code also includes an option by which you can ask it to put some messages
# on the queue at the beginning of its processing so that they can be consumed by the
# mainline processing of the class:
java -jar ./build/libs/ProcessMQMessages.jar ProcessMQMessages 3

Simulating execution of Action
Sent 3 test messages.
Successfully processed message ID:414d5120424d49582e514d3120202020824ab2595eb0d325
text=SampleMessage1: Mon Oct 02 21:14:08 BST 2017
Successfully processed message ID:414d5120424d49582e514d3120202020824ab2595fb0d325
text=SampleMessage2: Mon Oct 02 21:14:08 BST 2017
Successfully processed message ID:414d5120424d49582e514d3120202020824ab25960b0d325
text=SampleMessage3: Mon Oct 02 21:14:08 BST 2017
{"messagesProcessed":3}
End of simulation.
```

Error handling and poison messages

As with all messaging-based applications, it is important to consider how you will handle the case where a failure occurs in processing the message. In some scenarios you may be able to discard that message – for example, if the sender re-issues that request after failing to receive a response in an expected time period. However, it is often desirable to include a retry capability that handles situations where the processing of the message may fail because of a transient issue in the network connectivity or a downstream system.

To avoid the Action itself having to wait to retry the message processing (which significantly extends the processing duration you are charged for), the sample project consumes the message from the queue under a local transaction, and includes logic to roll back that transaction in the event of a failure in the processing. This means the message will remain on the queue and the Action will attempt to process the message again when the alarm fires around 1 minute later.

The above logic ensures that processing of a given message is resilient to short-lived errors in the surrounding infrastructure or downstream systems, but what if the failure is caused by something in the content of the message itself which will not be resolved by any number of retries – a "poison message"? Ideally the business processing logic would be able to detect this and handle the message appropriately (possibly by discarding it immediately), but we want to protect ourselves from the infinite re-delivery of failing messages, and so the sample project also includes logic that checks the number of times that a message has been re-delivered and triggers a different codepath when that scenario is detected.

If you are interested in simulating the behaviour of the Action in this scenario, you can do so by putting a message onto the queue (using the MQ Web Console, MQ Explorer or an application of your choice) that contains the text POISON MESSAGE!, and then use the Cloud Functions Activity Log view to see the details of how the message is handled.

3. Upload the Action to Bluemix, and configure the Alarm and Rule so that it is invoked

Now that we have successfully tested the Action locally, we are ready to upload it to the cloud to be tested in the real deployment environment. Make sure your command prompt is still authenticated to Bluemix (as you did when you installed the CLI plugin at the beginning of this tutorial), then create the Action in Bluemix using the following command:

```
# Log in to Bluemix (use the "--sso" option if you have a federated ID)
# Replace "YourOrgName" and "YourSpaceName" with the appropriate details of your account
bx login -a api.eu-gb.bluemix.net -o YourOrgName -s YourSpaceName

# Create the Action inside Bluemix
# (this command will take a few seconds to complete while the jar is uploaded)
wsk action create processMessages ./build/libs/ProcessMQMessages.jar \
  --main ProcessMQMessages

# Note that if you want to re-upload the Action later you can do so using the
# "wsk action update" command with the same arguments as above.
```

Apply default parameters to the Action in Bluemix

When we tested our Action locally, we set up the necessary configuration parameters by loading them from a file on the local filesystem. However, that approach isn't possible in the cloud because there is no filesystem for the Action to read from. Instead we can apply the default parameters to the Action itself using the following command, which uses the same properties file to provide the parameter values:

```
wsk action update processMessages -P configuration.json
```

Test the Action in Bluemix

Now that we have deployed the Action and configured its default parameters, we can test the invocation of our processing logic in the real runtime environment using the following commands:

```
# Invoke the action without any parameters to see how it will behave
# when it gets triggered for real
wsk action invoke --result processMessages

{
  "messagesProcessed": 0
}

# If you like, you can still use the "numTestMessages" parameter to put some sample
# messages on the queue as part of the Action to provide a simple test that it does
# actually process messages
wsk action invoke --result processMessages --param numTestMessages 5

{
  "messagesProcessed": 5
}
```

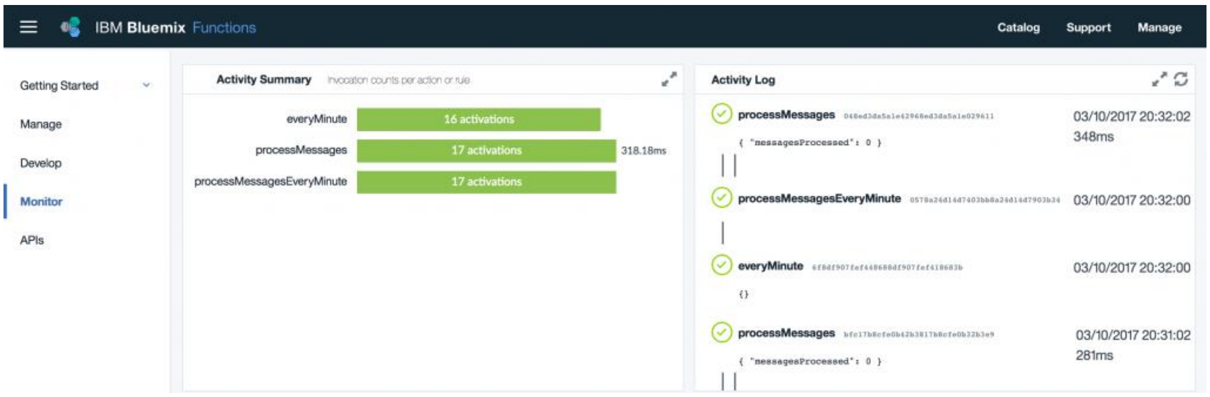
Set up an Alarm to trigger our Action regularly

We now have an Action that will process all the messages from the queue when we call it, but we need to put a trigger in place so that the action gets invoked regularly. Without that trigger our processing logic would never get executed. To do this in ICF we create an Alarm trigger – which fires at a repeating time interval that we specify – and then define a Rule that associates the Alarm (trigger) with our Action:

```
# Create an Alarm trigger that fires once every minute
wsk trigger create everyMinute \
  --feed /whisk.system/alarms/alarm \
  --param cron "*/1 * * * *" \
  --param trigger_payload "{}"

# Create a Rule that associates the trigger with the Action so that the action
# gets invoked every time the trigger fires.
wsk rule create processMessagesEveryMinute everyMinute processMessages
```

With the Action deployed – and the Trigger and Rule in place to execute it – we can now navigate to Monitor tab of the Cloud Functions user interface. After making sure that we have the correct Organization, Region and Space selected in the top right corner of the screen, we will be able to see evidence in the Activity Log of our processing logic being invoked, and we'll find that there are no messages waiting to be processed:

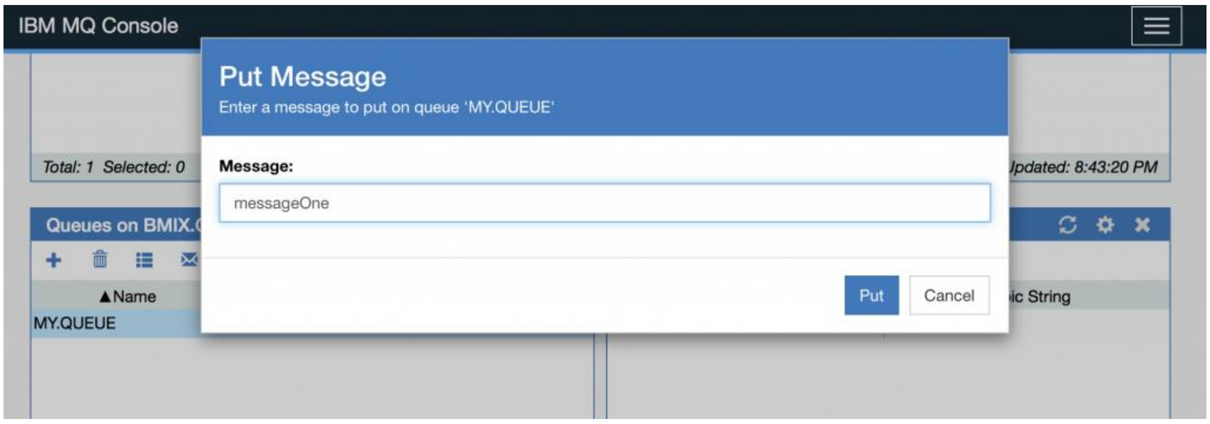


```
processMessages  048ed3da5a1e42968ed3da5a1e029611          03/10/2017 20:32:02
{ "messagesProcessed": 0 }                                 348ms
processMessagesEveryMinute  0578a24d14d7403bb8a24d14d7903b34  03/10/2017 20:32:00
everyMinute  6f8df907fef448688df907fef418683b              03/10/2017 20:32:00
{}
processMessages  bfc17b8cfe0b42b3817b8cfe0b32b3e9          03/10/2017 20:31:02
{ "messagesProcessed": 0 }                                 281ms
```

Cloud Functions – Monitor tab, including Activity Log

4. Put some sample messages to the queue to see our Action in action!

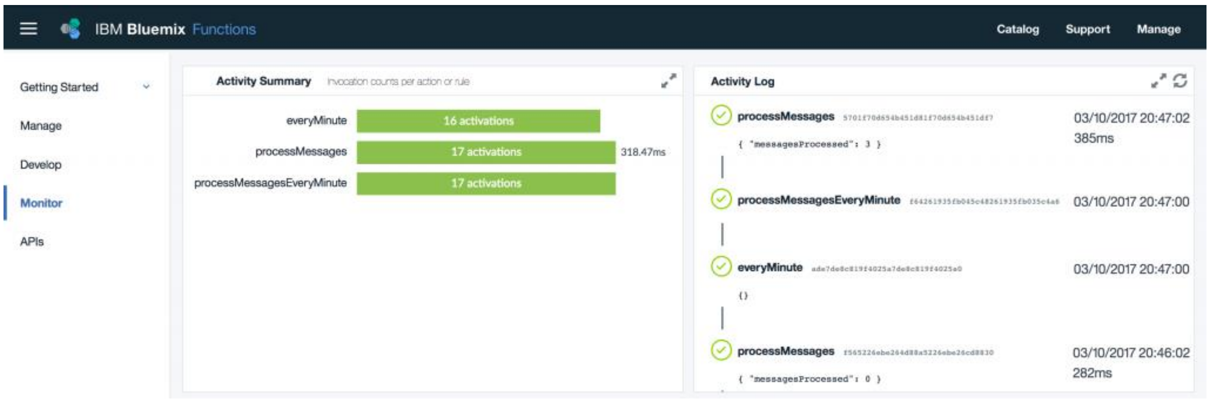
Everything is now in place and we are ready to see our processing logic work for real. Using your favourite tool (such as the IBM MQ Web Console, MQ Explorer, etc), connect to the queue manager and put some messages onto the queue (containing text of your choice, for example "messageOne", "messageTwo", "messageThree"):



Message: messageOne

Put message with MQ Web Console

If we wait for the next alarm to fire – up to 1 minute – we can see that the action successfully processed those three messages by looking at the response object of the action in the Activity Log:

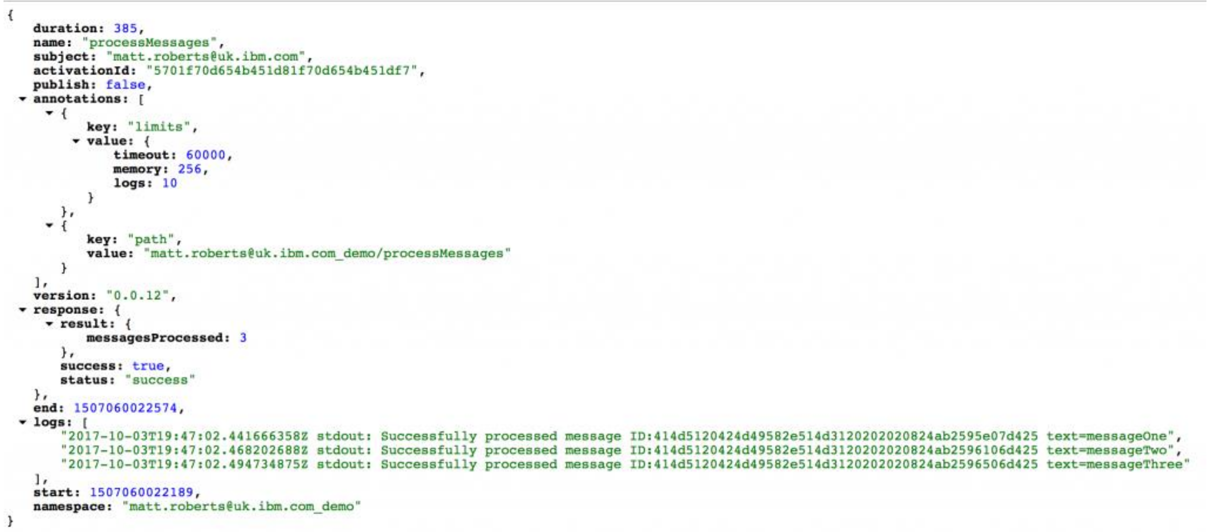


```
everyMinute                   16 activations
processMessages               17 activations   318.47ms
processMessagesEveryMinute    17 activations
```

```
processMessages  5701f70d654b451d81f70d654b451df7                03/10/2017 20:47:02
{ "messagesProcessed": 3 }                                         385ms
processMessagesEveryMinute  f64261935fb045c48261935fb035c4a6     03/10/2017 20:47:00
everyMinute  ade7de8c819f4025a7de8c819f4025a0                    03/10/2017 20:47:00
{}
processMessages  f565226ebe264d88a5226ebe26cd8830                03/10/2017 20:46:02
{ "messagesProcessed": 0 }                                         282ms
```

Activity log showing 3 messages processed

And finally, we can see even more detail we can click on the Activation ID (long uuid to the right of “processMessages” label, which is the name of the action) to see more metadata about the activation, including the “logs” element that shows any information that our application code printed to standard out.



```
{
    duration: 385,
    name: "processMessages",
    subject: "matt.roberts@uk.ibm.com",
    activationId: "5701f70d654b451d81f70d654b451df7",
    publish: false,
  ▾ annotations: [
      ▾ {
            key: "limits",
          ▾ value: {
                timeout: 60000,
                memory: 256,
                logs: 10
            }
        },
      ▾ {
            key: "path",
            value: "matt.roberts@uk.ibm.com_demo/processMessages"
        }
    ],
    version: "0.0.12",
  ▾ response: {
      ▾ result: {
            messagesProcessed: 3
        },
        success: true,
        status: "success"
    },
    end: 1507060022574,
  ▾ logs: [
        "2017-10-03T19:47:02.441666358Z stdout: Successfully processed message ID:414d5120424d49582e514d3120202020824ab2595e07d425 text=messageOne",
        "2017-10-03T19:47:02.468202688Z stdout: Successfully processed message ID:414d5120424d49582e514d3120202020824ab2596106d425 text=messageTwo",
        "2017-10-03T19:47:02.494734875Z stdout: Successfully processed message ID:414d5120424d49582e514d3120202020824ab2596506d425 text=messageThree"
    ],
    start: 1507060022189,
    namespace: "matt.roberts@uk.ibm.com_demo"
}
```

Activation Details for a sample invocation of the Action

Once you are finished with this tutorial, you may wish to disable or delete the Trigger and Rule using the Develop tab so that your logic doesn’t continue firing indefinitely for no reason!

Summary

This tutorial has shown how we can easily use messages arriving on an IBM MQ queue to trigger business logic implemented in a serverless computing platform such as IBM Cloud Functions / OpenWhisk, saving the enterprise time and money compared to deploying and operating a traditional long running application server infrastructure.

We’d love to hear your thoughts on this topic, so please do get in touch if you have questions or would like to discuss further!