# Cloud Developers Certification Training

## Section3 - First Deploy Exercises

**Version:** 1

**Last modification date:** 30-Jul-15

**Owner:** IBM Ecosystem Development

**Source: Developer Works**

**(https://developer.ibm.com/messaging/mq-light/docs/)**

# Exercise 4.3 – Lab Prerequisites

* Have a IBM Bluemix account
  + Sign up for Bluemix [http://bluemix.net](http://bluemix.net/)
* A Web Browser supported by Bluemix
  + - * Chrome, latest version for your OS
      * Firefox, latest version for your OS and ESR 31 or ESR 38
      * Internet Explorer, version 10 and 11
      * Safari, latest version for Mac
* DevOps Project Access:
  + <https://hub.jazz.net/project/tnevolin/certification-java-mqlight/overview>

# Exercise 4.3.1 – Create an MQ Light Service instance

Complete the following steps to add a MQLight service to your application:

1. In the Bluemix Dashboard, click ADD A SERVICE.
2. In the search field, enter MQ Light. Then, click MQ Light



1. Leave App field blank.
2. Name service “MQLight-sampleservice” or any other name you like.



You can also create the same MQ Light Service instance via command line that can be used by the apps deployed on to Bluemix

Example : cf cs ecodcnc-cert-mqlightservice

# Exercise 4.3.2 – Deploy the Sample Application to Bluemix

1. Download the code from:

* [https://hub.jazz.net/project/ecosysdevcnc/certification-java-mqlight/overview](https://hub.jazz.net/project/ecosysdevcnc/certification-nodejs-mqlight/overview)



1. If you named service not “MQLight-sampleservice” - edit manifest.yml and replace all occurences of “MQLight-sampleservice” to the actual service name you have.
2. Now that you’ve created the service, you’re ready to push the apps themselves, so navigate to the root directory where you downloaded the apps, and run:

* cf push

1. First, this uploads the app files themselves, then binds the service you created to both the apps which were pushed, and starts them up.

Tip: To do all this, this command uses the manifest.yml that is in the root directory to facilitate the deployment. This file is optional, but avoids having to specify all of the arguments each time you run the command (another tip: have a look at cf push -h if you want to specify these manually):

---  
applications:  
- name: MQL.sample.java.backend  
 memory: 256M  
 instances: 2  
 no-route: true  
 path: backend/target/BackendWorker-1.0-jar-with-dependencies.jar  
 buildpack: https://github.com/cloudfoundry/java-buildpack  
 services:  
 - MQLight-sampleservice  
- name: MQL.sample.java.frontend  
 memory: 256M  
 instances: 1  
 host: mqlightsample-java-frontend  
 path: frontend/target/FrontEndRESTApplication-1.0.war  
 services:  
 - MQLight-sampleservice

1. name is the name of your application in Bluemix.
2. disk is how much disk space your application have.
3. command is the command that is run to start your app once it is setup in Bluemix. In this case, it is the same command you used to start them locally.
4. path is the path to the app on your local machine.
5. memory is how much RAM you assign the app.
6. instances indicates how many instances of the app Bluemix creates – this is only relevant to the back end worker.
7. no-route indicates that the back end does not need a route (URL) created for it.
8. host indicates the route (URL) to create for the front end (i.e. where the web app is available). Note that ${random-word} generates a random word that should lower the chances of trying the create an already taken route.
9. services lists the services that you want to bind the app to upon creation. In this case it is the MQ Light service you created in the previous step.
10. When it’s done, you can run the cf apps command or check your Bluemix dashboard to see the apps running.

* cf apps – Command Line to check the status of applications

C:\Users\IBM\_ADMIN\git\certification-java-mqlight>cf apps

Getting apps in org ecodadmi@us.ibm.com / space timnev-dev as tnevoli@us.ibm.com...

OK

name requested state instances memory disk urls

javaplay-rest-dataservices-sample started 1/1 512M 1G javaplay-rest-dataservices-sample.mybl

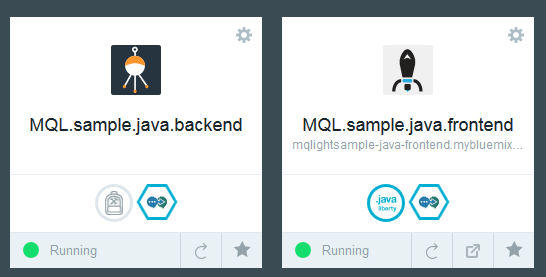
uemix.net

MQL.sample.java.backend started 2/2 256M 1G

MQL.sample.java.frontend started 1/1 256M 1G mqlightsample-java-frontend.mybluemix.

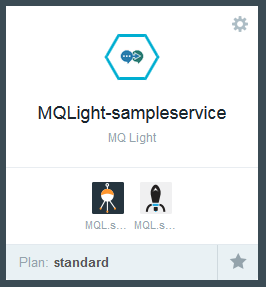
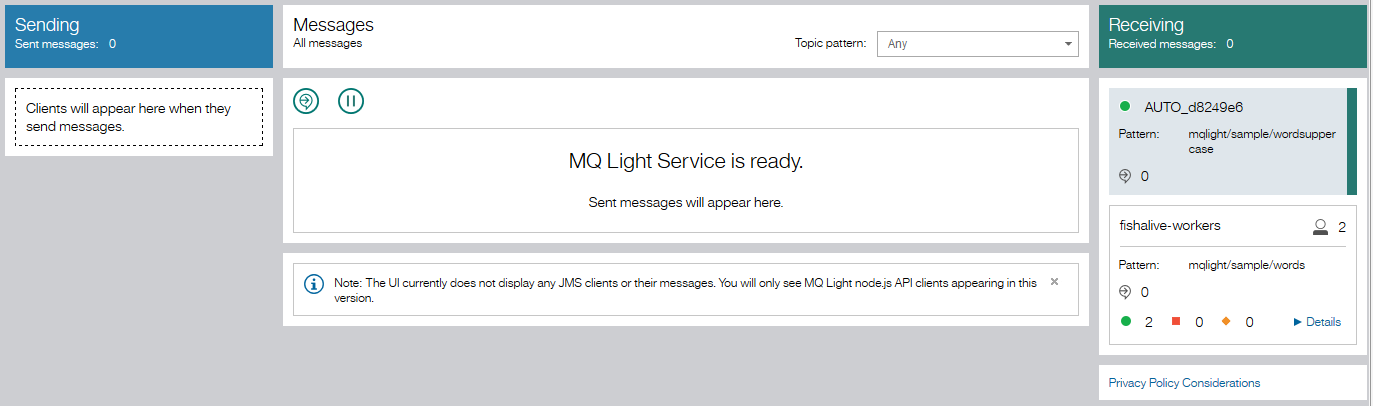
net

* Bluemix Dashboard to check the status of the applications



# Exercise 4.3.3 – MQ Light Dashboard UI in Bluemix

To get to the MQ Light UI:

1. From Bluemix dashboard click on MQ Light service.  
   
2. You'll see MQ Light UI.  
   
3. You should be able to see senders appearing on the left-hand side, and receivers on the right-hand side, with any messages flowing in the middle. For example, going back to the sample, after clicking on the ‘Submit work’ button, you should see something similar to the screenshot below.
4. You can click on ‘Topic List’ for senders and on ‘Details’ for receivers to get more information about the connected applications.
5. Click on ‘Details’ in any of the messages to get some more information on it, such as the payload, who sent it, to where, and whether or not anyone is waiting for it. This is perfect for troubleshooting messaging applications as you can see what your messages are actually doing.



1. For More information about the MQLight UI.
   * <https://developer.ibm.com/messaging/mq-light/docs/ui-reference/>

# Exercise 4.3.4 – Appendix Code Walkthrough

## Backend

Main backend class: BackendWorker.java.

Creating MQ Light client:

mqlightClient = NonBlockingClient.create(null, new NonBlockingClientAdapter<Void>() {

…

}

This code declares and defines MQ Light client by subclassing NonBlockingClientAdapter class and overriding event methods. Event “onStarted” fires when client initializes. The event method subscribes client to target topics and declares action to take in the “onMessage” event.

client.subscribe(SUBSCRIBE\_TOPIC, opts, new DestinationAdapter<Void>() {  
 public void onMessage(NonBlockingClient client, Void context, Delivery delivery) {  
 logger.log(Level.INFO,"Received message of type: " + delivery.getType());  
 StringDelivery sd = (StringDelivery)delivery;  
 logger.log(Level.INFO,"Data: " + sd.getData());  
 processMessage(sd.getData());  
 }  
 }, …

The important part is calling processMessage method with message payload. This method in turn converts message payload text to uppercase and sends it to another message topic for front-end to consume it.

public void processMessage(String message) {  
  
 JsonParser parser = new JsonParser();  
 JsonObject messageJSON = (JsonObject)parser.parse(message);  
  
 String word = messageJSON.get("word").getAsString();  
  
 JsonObject reply = new JsonObject();  
 reply.addProperty("word", word.toUpperCase());  
 reply.addProperty("backend", "JavaAPI: " + toString());  
  
 SendOptions opts = SendOptions.builder().setQos(QOS.AT\_LEAST\_ONCE).build();  
 mqlightClient.send(PUBLISH\_TOPIC, reply.toString(), null, opts, new CompletionListener<Void>() {  
 public void onSuccess(NonBlockingClient client, Void context) {  
 logger.log(Level.INFO, "Sent reply!");  
 }  
 public void onError(NonBlockingClient client, Void context, Exception exception) {  
 logger.log(Level.INFO,"Error!." + exception.toString());  
 }  
 }, null);  
 }

## Front end

Front end consists of index.html page with java script that sends initial words to message topic and a poller that polls another subscription topic to get processed messages.