

Impact2014



April 27 - May 1 Las Vegas, NV

Developing scalable applications with IBM MQ Light and ElasticMQ: Lab Instructions

Authors:

Steve Upton, MQ Light Development Team

Matthew Whitehead, MQ Light Development Team

Rob Nicholson, STSM Application Messaging

What you will learn

This lab will teach you how you can improve the responsiveness and scalability of your web applications, both on premise and in the cloud.

It will explain how to off-load heavy workloads to separate worker threads while your web handlers deal quickly and efficiently with the requests from your online users.

As software developers we want our applications to be responsive and scalable to really engage users, but it's not always easy to write code that behaves like this. MQ Light and the ElasticMQ Service in BlueMix are great tools that helps applications off-load work to be dealt with asynchronously thus ensuring your applications responds quickly. Additionally, as workload increases, applications that use MQ Light become very easy to scale up.

What the lab covers

This lab has 5 parts to it:

1. Running the sample node.js application
2. Improving the sample application by separating the web-facing component from the data-processing component
3. Scaling up the number of data-processing threads to cope with the workload
4. Deploying your finished application to IBM BlueMix to run it in the cloud using the ElasticMQ service
5. Changing the worker application from node.js to Java using JMS – illustrating polyglot messaging.

Lab prerequisites

This lab assumes that you have a few common tools already installed. These pre-reqs and where you can find them are listed below.

1. git command line tool

There are required to download the source code for the lab. Installation instructions can be found here <http://git-scm.com/downloads>

2. Cloud foundry command line tools

Required to push code into BlueMix. The installation instructions can be found here <https://github.com/cloudfoundry/cli> . Scroll down for the stable installers section for your OS instructions.

3. IBM Codename:BlueMix account

To push code into the BlueMix PaaS you need to sign up to the BlueMix beta with an account. You can sign up here <https://ace.ng.bluemix.net/> Please let us know if there are any problems getting this activated.

4. node.js

The code for this lab is mostly written for node.js. In order to run it locally you will need a version of node.js which can be found here <http://nodejs.org/>

5. MQLight

MQ Light supports the same node API as ElasticMQ, allowing you to develop your application locally before pushing to the cloud. Download our latest Beta release from <https://www.ibm.com/messaging/mq-light/>

Setup – Extract the source for the labs

1. Open a terminal window and navigate to a location that you want to run the lab from. For the purposes of this lab we will refer to this location as */home/demo/mql/source*

2. Extract the source files from git by using the command

```
git clone https://github.com/ibm-messaging/mqlight-sentiment-sample
```

This should extract the 4 projects that we will be using into the current directory.

3. Extract the MQ Light archive (zip or tar.gz) downloaded from

<https://www.ibmmdw.net/messaging/mq-light/> For the purposes of this tutorial we will assume this is extracted to */home/demo/MQLight/*

Note: If you wish to run the BlueMix part of the lab (parts 4 and 5) you will need to get a BlueMix ID. We would suggest you skip straight to the first steps in part 4 where there are instructions on signing up for BlueMix. That will give us more time to make sure you have a working BlueMix ID by the time you come to run those parts of the lab.

BlueMix is currently in beta and hence logon can be tricky at times. You may find you need to clear your browser cache in order to logon successfully.

When you have requested your BlueMix ID, let one of the lab helpers know so they can make sure the ID request is approved. Then return to part 1 of the lab.

You should now be ready to run the lab!

Part 1 – Running the sample node.js application

To introduce Elastic MQ & MQ Light and demonstrate how they can be incorporated into your projects we have constructed a node.js application which will process a stream of data from Twitter. In our sample scenario, the messages arriving from Twitter are used in place of messages being submitted by web users.

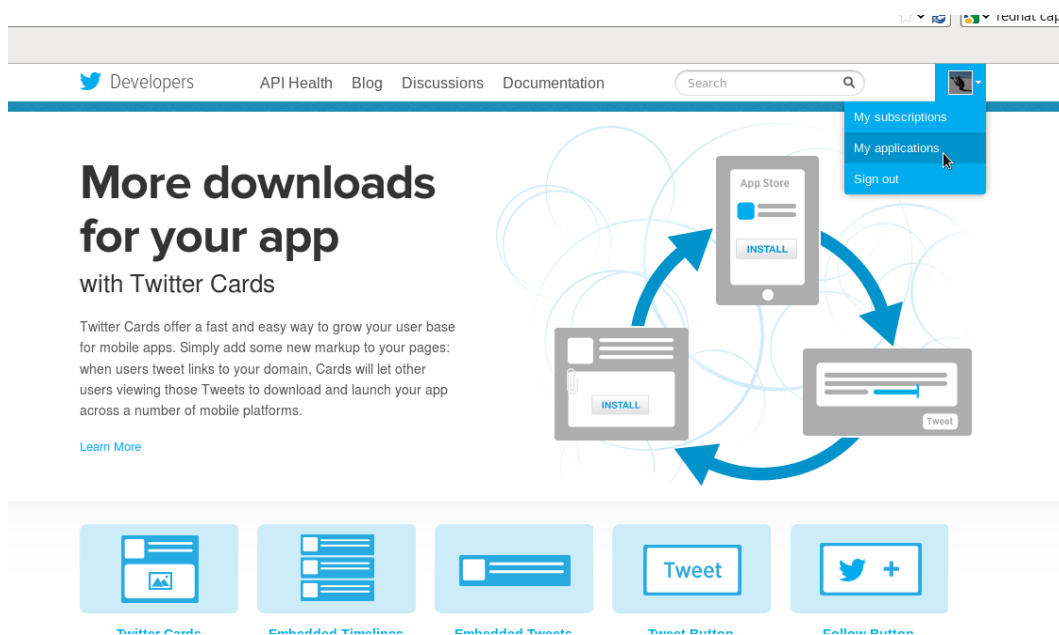
In the sample application, when messages are received from Twitter they are processed by a function running in the same thread that received the message. The processing performs some basic analysis of the data to simulate the sort of backend processing your applications might do.

In this first part of the lab you will start the sample application and see messages arriving from Twitter and being processed.

1. Sign in to Twitter

Twitter requires us to authenticate with a set of temporary developer credentials. Before you can run the sample application you must generate a set of credentials to use.

In your browser navigate to <https://dev.twitter.com> and sign-in at the top right of the screen (you will need a Twitter ID). Once you are signed in, hover over your user icon at the top right of the screen and from the drop down that appears choose “My Applications” to view your applications:



2. Create an application by pressing the 'Create New App' button

Once you are logged in you should see a button called 'Create New App' which you can click on to start creating your first application.

3. Enter your twitter application details

Fill out the fields in the form (don't worry about the URL being a valid URL – just enter something for now)

Create an application

Application details

Name *

Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens. 32 characters max.

Description *

Your application description, which will be shown in user-facing authorization screens. Between 10 and 200 characters max.

Website *

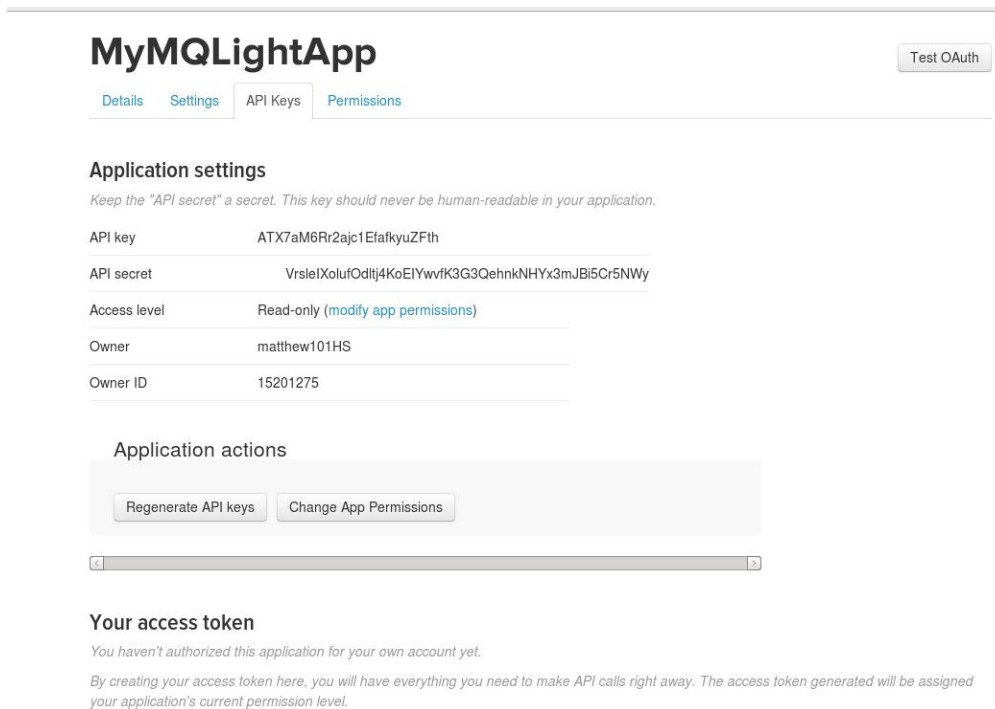
Your application's publicly accessible home page, where users can go to download, make use of, or find out more information about your application. This fully describes your application and will be shown in user-facing authorization screens.
(If you don't have a URL yet, just put a placeholder here but remember to change it later.)

Callback URL

Where should we return after successfully authenticating? [OAuth 1.0a](#) applications should explicitly specify their `oauth_callback` URL on the request token step. If your application is not using OAuth 1.0a, or if you are using OAuth 2.0, you can leave this field blank.

4. Generate the required twitter keys

Once you have created your application, go to the “API Keys” tab and scroll to the bottom of the page. Select the option to “Create my access token”:



The screenshot shows the 'API Keys' tab for an application named 'MyMQLightApp'. The page has a navigation bar with 'Details', 'Settings', 'API Keys', and 'Permissions'. A 'Test OAuth' button is in the top right. Under 'Application settings', there is a table with the following data:

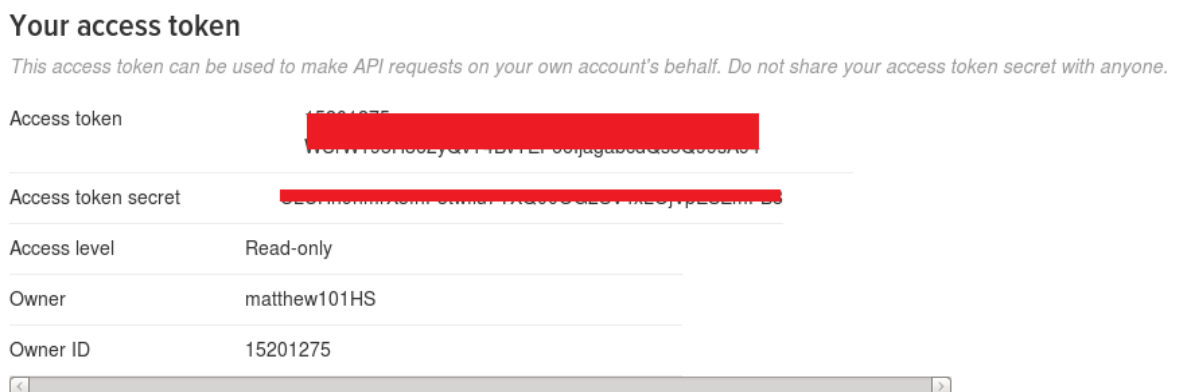
API key	ATX7aM6Rr2ajc1EfafkyuZFth
API secret	VrslelXolufOdtij4KoElywvfK3G3QehnkNHx3mJBi5Cr5NWy
Access level	Read-only (modify app permissions)
Owner	matthew101HS
Owner ID	15201275

Below the table are 'Application actions' with buttons for 'Regenerate API keys' and 'Change App Permissions'. A scroll bar is visible. Under 'Your access token', there is a note: 'You haven't authorized this application for your own account yet. By creating your access token here, you will have everything you need to make API calls right away. The access token generated will be assigned your application's current permission level.'

5. Wait for your twitter keys to be generated

It may take a moment for the keys to be generated. There is a 'refresh' option at the top of the page which you can use if the keys take a while to generate.

Once they have been generated you can minimise the browser and we'll come back to retrieve them later.



The screenshot shows the 'Your access token' page. It includes a note: 'This access token can be used to make API requests on your own account's behalf. Do not share your access token secret with anyone.' Below this is a table with the following data:

Access token	15201275 [REDACTED]
Access token secret	[REDACTED]
Access level	Read-only
Owner	matthew101HS
Owner ID	15201275

A scroll bar is visible at the bottom of the table.

6. locate the files which need updating with your twitter keys

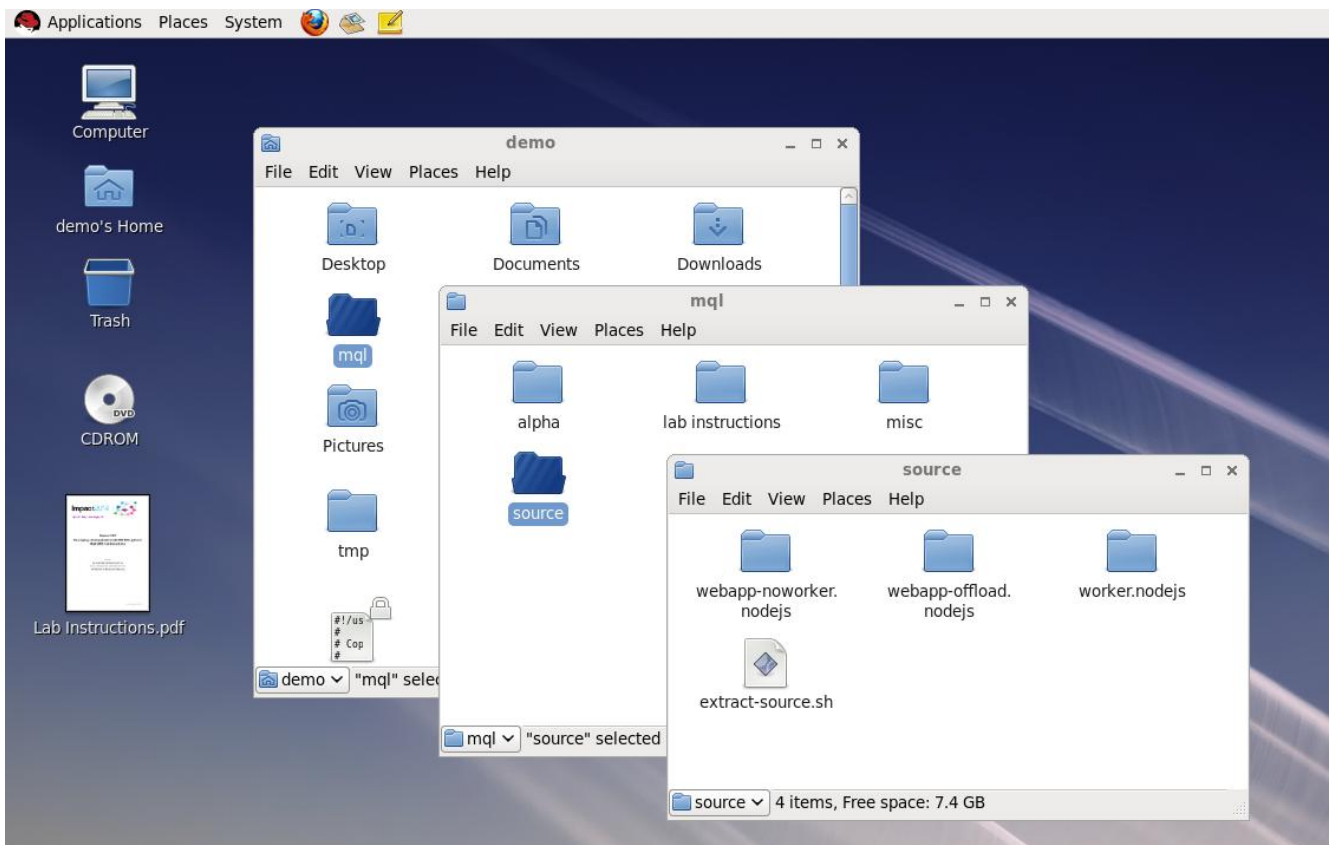
Open the sample application and locate the sample files that we need to insert the tokens you generated into.

There is one application file you will need to work with: `webapp-noworker.nodejs/app.js`.

The application does 2 things:

- Read tweets from Twitter
- Process each tweet, checking for mentions of product names in the tweets and for each one it finds, performing sentiment analysis of the tweet to determine if the person was tweeting something good or something bad about it. For the purposes of this lab, we have used country names in place of product names to guarantee a good flow of matching tweets and the actual sentiment analysis module is replaced with a random function.

Navigate to the folder where the source code for this lab was extracted (`/home/demo/mql/source`).



The file we need to change is `twitterkey.json`. There is a version of this file in each of the application folders inside `/home/demo/mql/source`

Open each of the folders in turn and open `twitterkey.json` in your favourite text editor.

The file should look like this:

```
{  
  "consumer_key": "XXXXXXXXXXXXXXXXXXXX",  
  "consumer_secret": "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",  
  "access_token_key": "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",  
  "access_token_secret": "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"  
}
```

7. Insert your twitter application keys

Replace the key and secret strings in the files with the values in your browser that you minimised earlier.

Set ***consumer_key*** in the file to the API Key string in your browser

Set ***consumer_secret*** in the file to the API Secret string in your browser

Set ***access_token_key*** in the file to the Access Token string in your browser

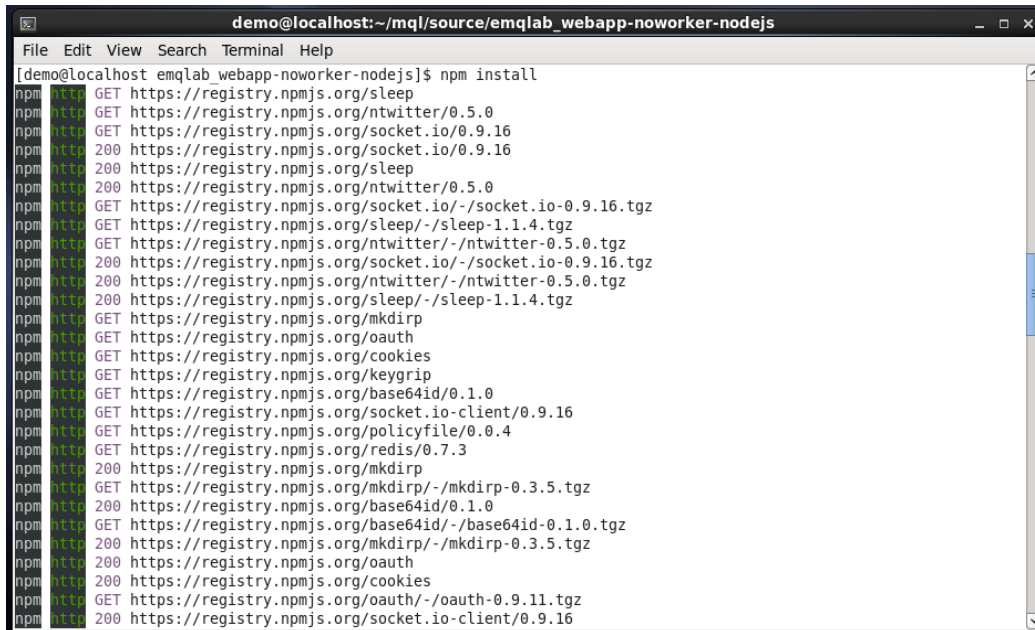
Set ***access_token_secret*** in the file to the Access Token Secret string in your browser

Save the files and close the text editors.

8. Install the node.js dependencies

Before you can run the sample we need to install the dependencies the application has. Open a terminal window and navigate to the first sample application (`/home/demo/mql/source/webapp-noworker.nodejs`).

Type the command **npm install** and hit enter:

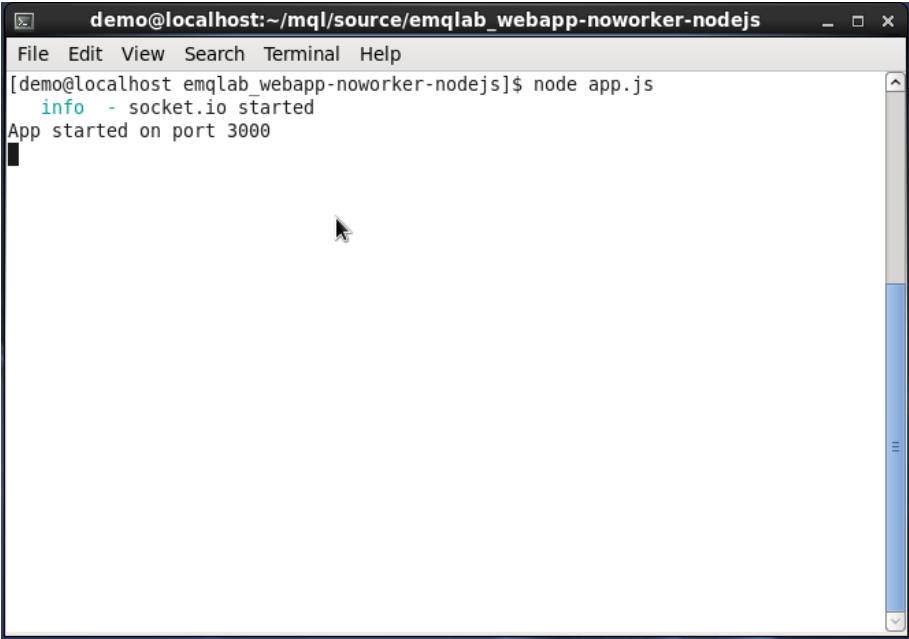


```
demo@localhost: ~/mql/source/emqlab_webapp-noworker-nodejs
File Edit View Search Terminal Help
[demo@localhost emqlab_webapp-noworker-nodejs]$ npm install
npm http GET https://registry.npmjs.org/sleep
npm http GET https://registry.npmjs.org/ntwitter/0.5.0
npm http GET https://registry.npmjs.org/socket.io/0.9.16
npm http 200 https://registry.npmjs.org/socket.io/0.9.16
npm http 200 https://registry.npmjs.org/sleep
npm http 200 https://registry.npmjs.org/ntwitter/0.5.0
npm http GET https://registry.npmjs.org/socket.io/-/socket.io-0.9.16.tgz
npm http GET https://registry.npmjs.org/sleep/-/sleep-1.1.4.tgz
npm http GET https://registry.npmjs.org/ntwitter/-/ntwitter-0.5.0.tgz
npm http 200 https://registry.npmjs.org/socket.io/-/socket.io-0.9.16.tgz
npm http 200 https://registry.npmjs.org/ntwitter/-/ntwitter-0.5.0.tgz
npm http 200 https://registry.npmjs.org/sleep/-/sleep-1.1.4.tgz
npm http GET https://registry.npmjs.org/mkdirp
npm http GET https://registry.npmjs.org/oauth
npm http GET https://registry.npmjs.org/cookies
npm http GET https://registry.npmjs.org/keygrip
npm http GET https://registry.npmjs.org/base64id/0.1.0
npm http GET https://registry.npmjs.org/socket.io-client/0.9.16
npm http GET https://registry.npmjs.org/policyfile/0.0.4
npm http GET https://registry.npmjs.org/redis/0.7.3
npm http 200 https://registry.npmjs.org/mkdirp
npm http GET https://registry.npmjs.org/mkdirp/-/mkdirp-0.3.5.tgz
npm http 200 https://registry.npmjs.org/base64id/0.1.0
npm http GET https://registry.npmjs.org/base64id/-/base64id-0.1.0.tgz
npm http 200 https://registry.npmjs.org/mkdirp/-/mkdirp-0.3.5.tgz
npm http 200 https://registry.npmjs.org/oauth
npm http 200 https://registry.npmjs.org/cookies
npm http GET https://registry.npmjs.org/oauth/-/oauth-0.9.11.tgz
npm http 200 https://registry.npmjs.org/socket.io-client/0.9.16
```

Congratulations! You're now ready to run the basic sample application!

9.Run the sample application

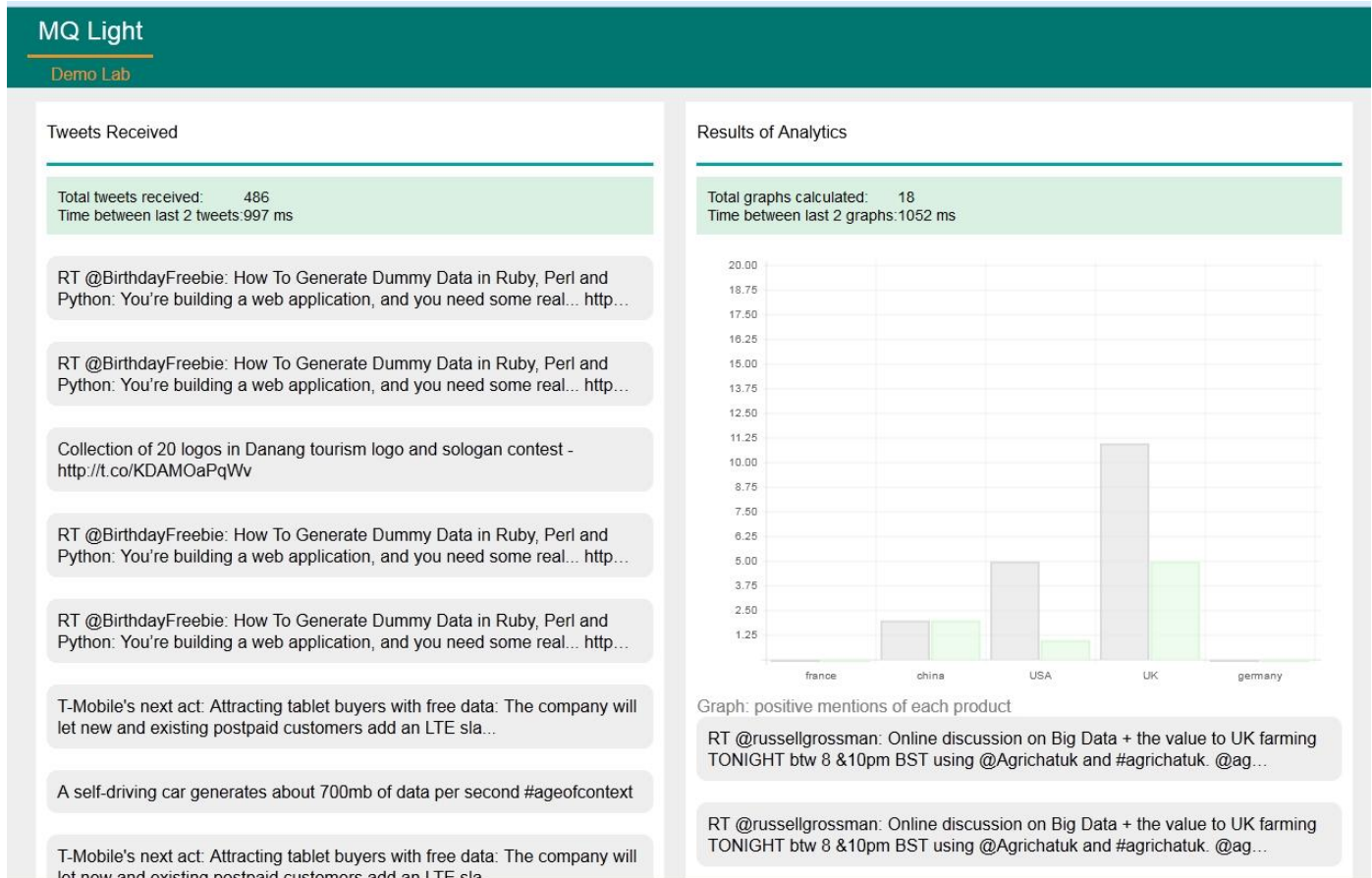
To run the application type **node app.js** and hit enter:



```
demo@localhost:~/mql/source/emqlab_webapp-noworker-nodejs
File Edit View Search Terminal Help
[demo@localhost emqlab_webapp-noworker-nodejs]$ node app.js
  info - socket.io started
App started on port 3000
```

You will notice that nothing much happens. That's because the sample application waits for a browser to connect before it starts collecting data from Twitter. To start the application running, open a web browser and navigate to localhost port the application is listening on (<http://localhost:3000>)

The web page should look something like this:



Tweets should start appearing on the left hand side of the screen. On the right hand side of the screen is a graph of tweets that mention the “products” we are interested in. For the purposes of this lab we have replaced the product names with the names of countries to ensure a good flow of matching tweets. Whenever a new tweet arrives from the twitter API, the application scans the tweet to determine if it matches a “product”. If it does then the tweet will appear on the right hand side and the gray bar for that product is updated with the cumulative total of matching tweets. If the tweet is analysed to reflect a positive sentiment then the height of green bar of positive tweets is also increased.

You will notice that tweets are being processed and displayed at a rate not exceeding one per second. You will also notice that new tweets aren't shown on the left hand side until the graph has finished generating.

The reason for this is because the sentiment analysis and graph generating code is running in the same thread as the code that communicates with Twitter. A new tweet cannot be received until the graph is generated and the application can ask for the next tweet.

If the tweets were actually requests for orders from your customers, every thread handling a customer's request would be locked up waiting for the order processing to complete. The thread would not be available to service another customer's request.

10. Stop the application and clear your browser

Terminate the app.js application in the terminal by pressing Ctrl-C in the terminal window.

Hit refresh on the browser window to clear it and prevent the web page from automatically connecting to the applications we run later in the lab.

Part 2 – Improving the sample by decomposing the app to separate the web-facing and data-processing components

In part 1 we ran a single-threaded application which handled receiving tweets and processing them all in the same thread.

This meant that the number of tweets we could receive and handle was limited to the rate that we could process them.

In this part of the lab we are going to separate the code that receives the tweets from the code that processes them. This will allow us in the later parts of the lab to scale the number of threads we have processing the tweets to allow the application to cope with more workload.

There are 2 applications you will need for this section.

webapp-offload.nodejs/app.js

This does a similar thing to the original app.js which we used earlier. We've removed the logic that processed the tweets and moved it into worker.js (see below). We've also added the code required to connect it to an MQ Light server. This will be necessary for offloading the tweets to MQ Light.

worker.nodejs/worker.js

The worker application is a new application that we didn't need in part 1 because all of the processing was done in one place. The worker has the same code that the original app.js used to process tweets, but it also has the code required to connect it to MQ Light and consumer the messages that have been offloaded by webapp-offload.nodejs/app.js

Note: Before you go through the following set of steps, make sure you have edited app.js to include your Twitter API keys in the way you did during part 1 of the lab.

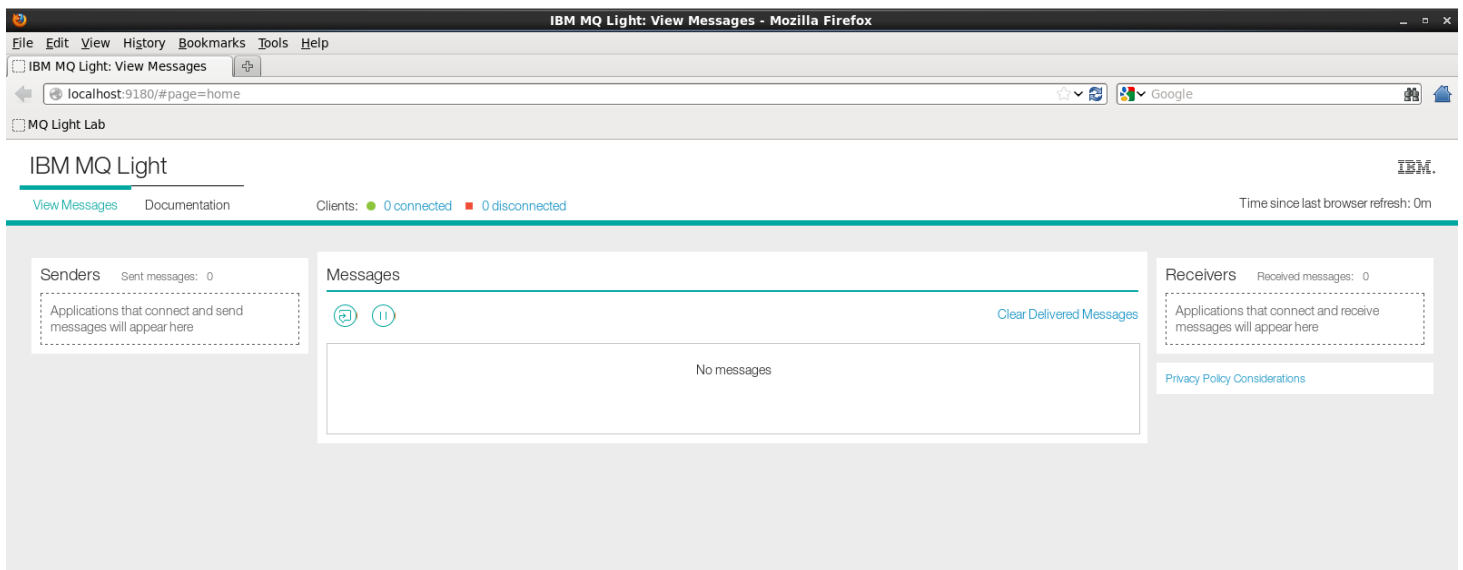
1. Start MQ Light

Because we're going to start offloading work to MQ Light, we need to start the MQ Light runtime so that the applications can connect to it.

Open the folder where MQ Light has been extracted (*/home/demo/MQLight/*)

To start MQ Light, double click on “Start IBM MQ Light”. A new terminal window will open and you will be asked to accept the license. Press 1 and hit enter. MQ Light will start up.

2. Checkout the MQ Light GUI



When MQ Light has finished starting it will automatically start the GUI in the browser. The first page you will see is the documentation tab. To switch to the development screen click on “View Messages” at the top left of the page. You should now see something like this:

We will explore the UI in more detail as we go through running the different parts of the application.

3. Examine the code that makes use of MQ Light

Open the application to inspect the calls it makes to connect to MQ Light

If you open *home/demo/mql/webapp-offload.nodejs/app.js* in a text editor you can see the code we've added to allow the application to connect to MQ Light and offload the tweets for processing.

```
var mqlight = require('mqlight');
```

This line at the top of the file simply loads the mqlight node.js libraries.

```
var opts = { host: mql_ip , port: mql_port, service:'amqp://localhost'};

var client = mqlight.createClient(opts);
```

These lines set up the options for connecting to the mqlight service, and then create a new mqlight client with those options.

```
client.connect(function(err) {
  if (err) {
    console.log(err);
  }
});
```

On this line we tell the client to connect to the MQ Light server. Notice how we pass in a callback function which will be called if there is a problem connecting to MQ Light.

```
client.on('connected', function() { . . . }
```

Here we are telling the client which function should be called when a successful connection to MQ Light has been made.

```
client.send(topic, body, function(err, msg) {
```

When a tweet has been received, this line tells the client to send a message to MQ Light on the specified topic. Again we pass in a callback function which will be called when the message has been sent, or if an error occurred.

```
var destination = client.subscribe('processedData', callback);
```

The final line we'll look at creates a subscription to the topic that the worker threads will send their responses on. When a message is received on the 'processedData' topic the callback method will be called.

The worker.js application uses similar calls to subscribe to tweet messages sent to it by the offloading application and send responses back to it.

Let's try running them!

4. Download the required node dependencies.

Make sure you have 2 terminal windows open. In one of them navigate to `/home/demo/mql/worker.nodejs` and in the other one navigate to `/home/demo/mql/webapp-offload.nodejs`

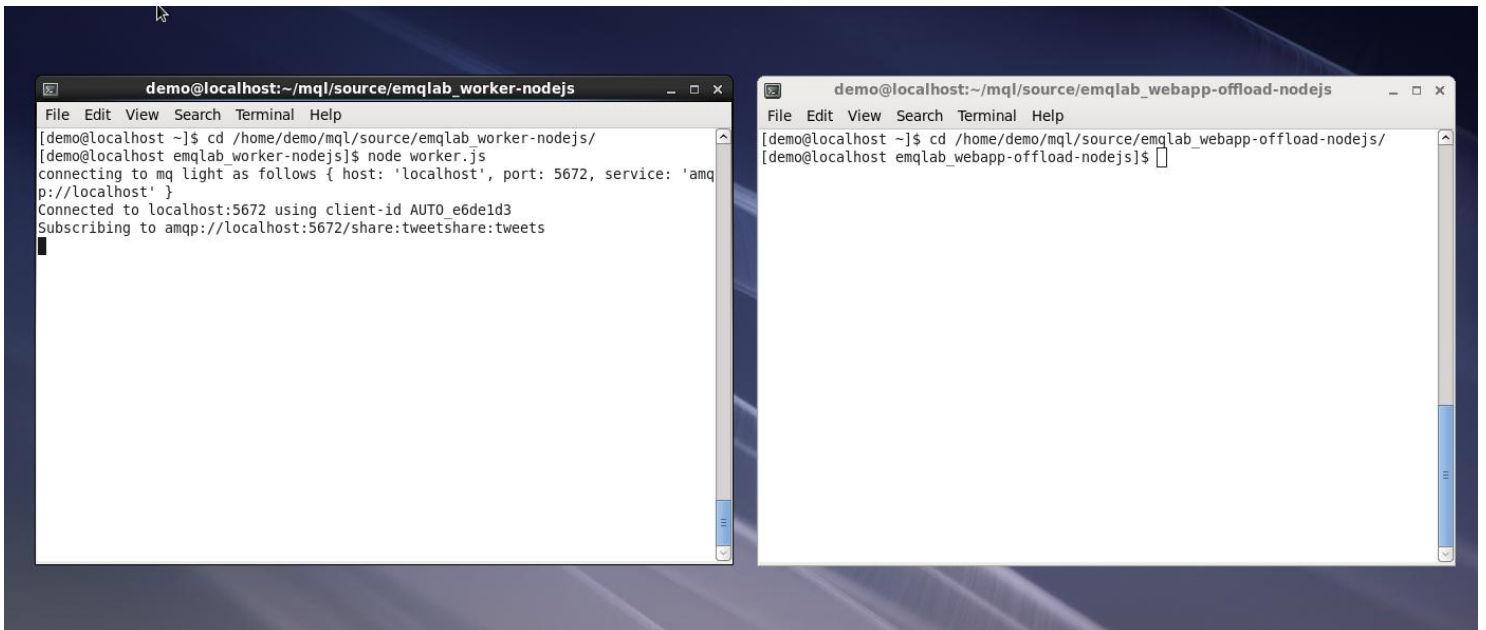
As before, we need to install the node.js dependencies for each application using the npm command.

Type **npm install** into each terminal and press enter.

5. Launch the worker part of the application

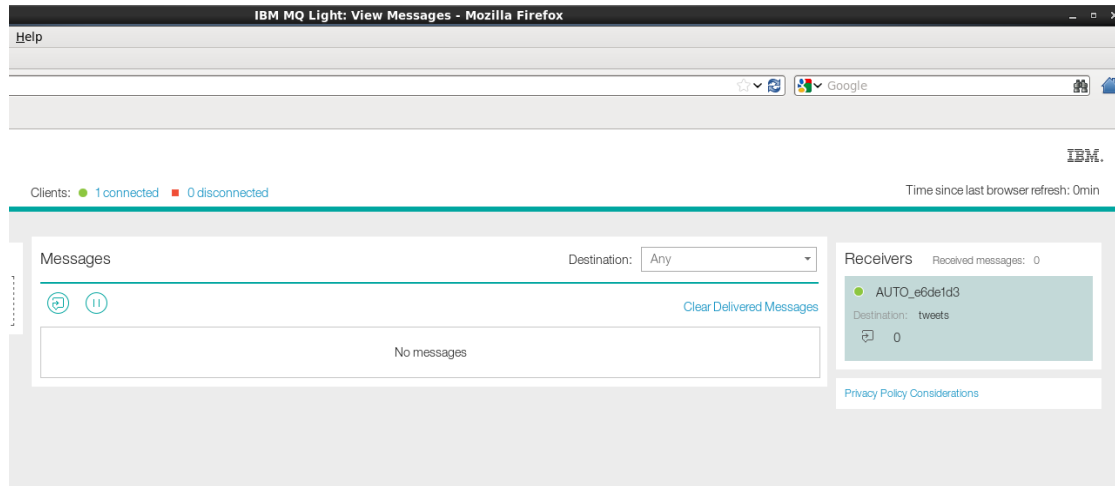
In the first terminal run the command **node worker.js**

The worker application will connect to MQ Light and wait for messages to arrive:



If you switch back to the GUI in your browser you should now see that the worker application has connected successfully, and that it is in the receivers view on the right hand side of the screen because it has created a subscription to receive messages from.

You can see that the topic that the worker application is subscribed to is 'tweets'.



The MQ Light GUI is designed to make development of apps very quick and easy. When you're using MQ Light to develop your own applications you can use the GUI to check that your app is connecting to MQ Light correctly, and that any subscriptions you create are been successfully made.

6. Launch the web front end part of the application

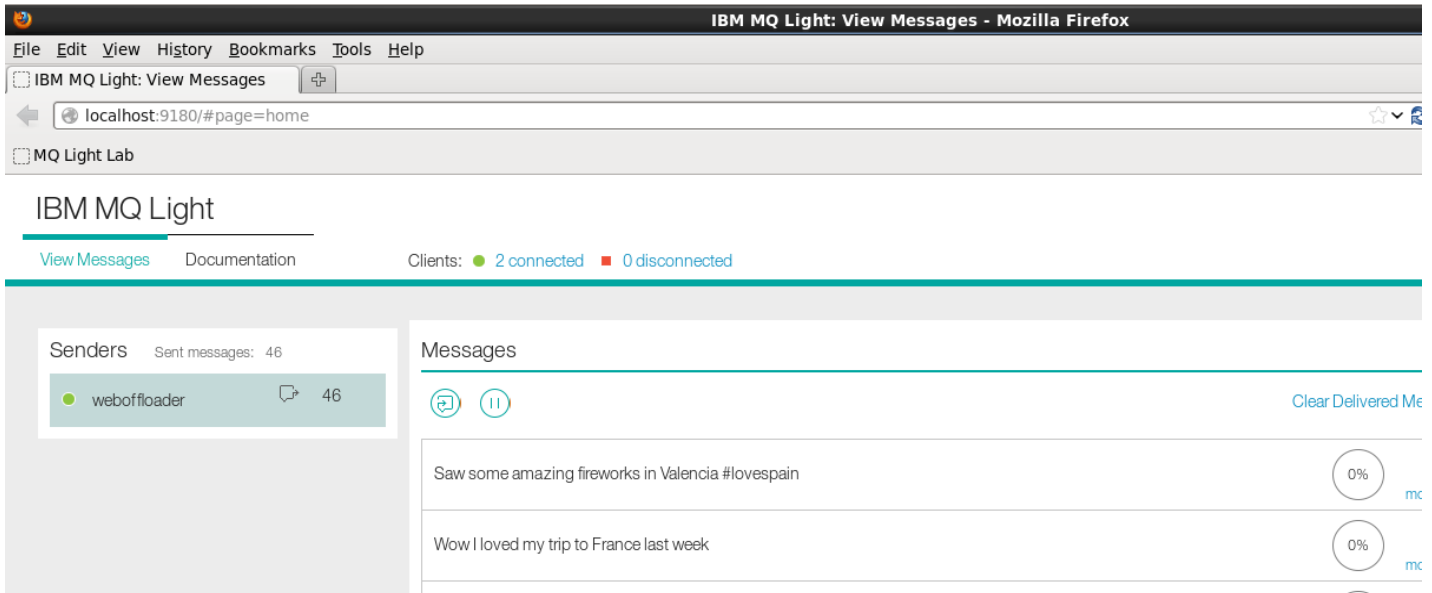
In the second terminal run the command **node app.js**

The application will start and, as before, it will wait for the browser to connect to it before it begins to receive data from Twitter.

7. Try the running application

In your browser open a new tab and navigate back to the MQ Light demo page using the bookmark on the toolbar. The browser connects to the app.js application which will now start receiving messages from Twitter.

Now that the application has started properly, you can use the MQ Light GUI to check that it is connected to MQ Light and that it is publishing messages to the worker application. If everything is working correctly you should see the application in the list of senders on the left hand side of the screen:



Switch back to the demo tab in your browser. Notice how the left hand side of the screen is updating as soon as new tweets arrive. This is because instead of processing each tweet it simply hands it off to MQ Light and goes back to receive and display the next one.

The worker thread is asynchronously working through the list of messages arriving on the 'tweet' topic. When it has finished processing a tweet, if the tweet contained a reference to a product name it sends a response to the browser with the result of the sentiment analysis. Any tweets which don't contain references to products are simply discarded.

We have improved the basic application by separating the steps of receiving tweets and processing them. There is still a problem though. If you at the MQ Light GUI and select the icon to hide completely delivered messages, you will see that the number of unprocessed messages slowly builds up.

The reason for this is that although we've freed up the offloading application, we still only have a single worker thread processing the data. The worker thread can only process one tweet per second and if tweets arrive more frequently than that then the worker thread cannot keep up!

In the next part of the lab we will demonstrate how to scale up the number of worker threads which are available to process the tweets.

8. Stop the applications and clear your browser

Terminate the `app.js` and `worker.js` applications in their terminals by pressing `Ctrl-C` in the terminal windows.

Hit refresh on the MQ Light demo browser window to clear it and prevent the web page from automatically connecting to the applications we run later in the lab. You leave the MQ Light GUI running.

Part 3 – Scaling up the number of data-processing threads to cope with the workload

As we saw in the previous section, the single worker thread is unable to cope with the number of tweets being offloaded to it. We can't just make the worker thread go more quickly. What we need to do is create more of them!

The first thing we have to do is change the way the worker application subscribes. In the current version the worker thread subscribes to the topic 'tweets' and receives all of the messages published to that topic. If we simply started another instance of the worker application both instances would receive a copy of every tweet. This would mean that each tweet got processed twice which we don't want to do.

To change this behaviour we need both of the worker applications to share the messages between them. The way we can do this using the MQ Light API is to make the worker threads join a “share”. A “share” is a group of applications that are all subscribed to the same topic and where each message published on that topic is only given to one of them.

To change the worker application we need to do 2 things:

Choose a name for the share group

Edit the worker application to make it subscribe as part of that share

In these instructions we will use the share group name “tweetprocessors” but you can choose something else if you prefer.

1. Open the worker.js file ready to update it

Find the worker.js application in the /home/demo/mql/worker.nodejs folder, and open it for editing using your editor.

2. Locate the line in the application where it creates the subscription to the 'tweets' topic:

```
client.subscribe("tweets", function(err, address) {
```

We need to add another parameter to the subscribe(...) method to specify the name of the share we want our worker to join. Edit the line in the file to look like the following:

```
client.subscribe("tweets", "tweetprocessors", function(err, address) {
```

3. Save the file and close the editor

4. Get ready to run your modified worker application

Open 2 terminal windows to run the 2 worker applications. In both terminals navigate to */home/demo/mql/source/worker.nodejs*

5. Launch the modified worker

Enter the command **node worker.js** into each terminal and hit enter. This will start both of the worker applications. Open the browser with the MQ Light GUI in it and check that the 2 worker applications are running correctly and connected to MQ Light.

6. Re-launch the web front end

In a third terminal window navigate to */home/demo/mql/source/webapp-offload.nodejs* and run **node app.js** to start the main application.

As before you must open a browser and launch the MQ Light demo page using the bookmark on the toolbar. This will start the application consuming messages from Twitter and offloading them to the worker threads via MQ Light.

Now the application is running, check that the application is connected to MQ Light and publishing messages by opening the MQ Light GUI in the browser. You should see the

number of messages the application has published so far, and the number of messages each worker thread has processed.

Note: The number of messages received by each worker thread may not be equal. You would expect them to get 1 each in turn, but because of the way MQ Light decides which application is best able to process the work, one instance may process more work than the other.

7. Increase the number of worker threads

Obviously 2 worker threads is better than 1, but we can start as many as we need to cope with the data processing workload. Open a new terminal window and repeat step 5 above, checking in the MQ Light GUI that the new worker thread is running correctly and processing some of the messages.

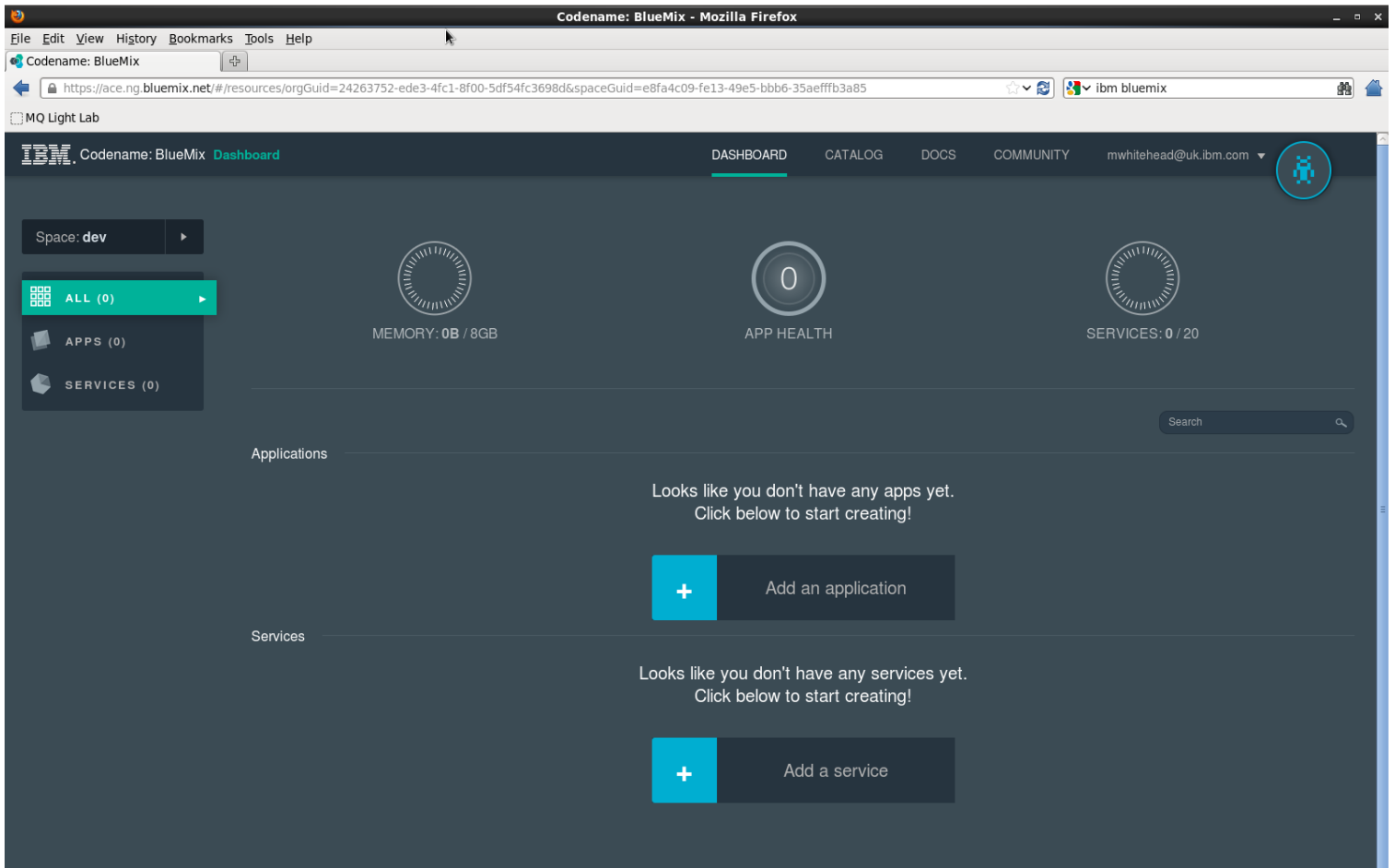
Part 4 – Deploying your finished application to IBM BlueMix to run it in the cloud

You've successfully developed and tested your application using the standalone MQ Light runtime. However, the same API is supported in IBM BlueMix by the Elastic MQ service.

In this part of the lab we are going to deploy the two applications – `offloadapplication.js` and `worker.js` - into IBM BlueMix and bind them both to a single Elastic MQ service. Elastic MQ will act in the same way that the local MQ Light runtime did. The applications will connect to it, and `offloadapplication.js` will publish its messages while the `worker.js` applications subscribe and receive them.

To do this part of the lab you will need to sign up to IBM BlueMix which you can do here <https://ace.ng.bluemix.net/>

Once you have signed up and logged in to BlueMix you will be presented with the BlueMix dashboard:



You can define applications and create instances of services using the BlueMix dashboard, but for the rest of the lab we'll use the command line to speed things up a bit. All you need is the username and password you used to sign up to BlueMix.

First of all we can try deploying the simple application that doesn't offload any work to MQ Light. This will allow you to familiarise yourself with the commands for deploying applications to BlueMix and check that your credentials are working.

1. Set up your BlueMix command environment

Open a terminal window and enter the command :

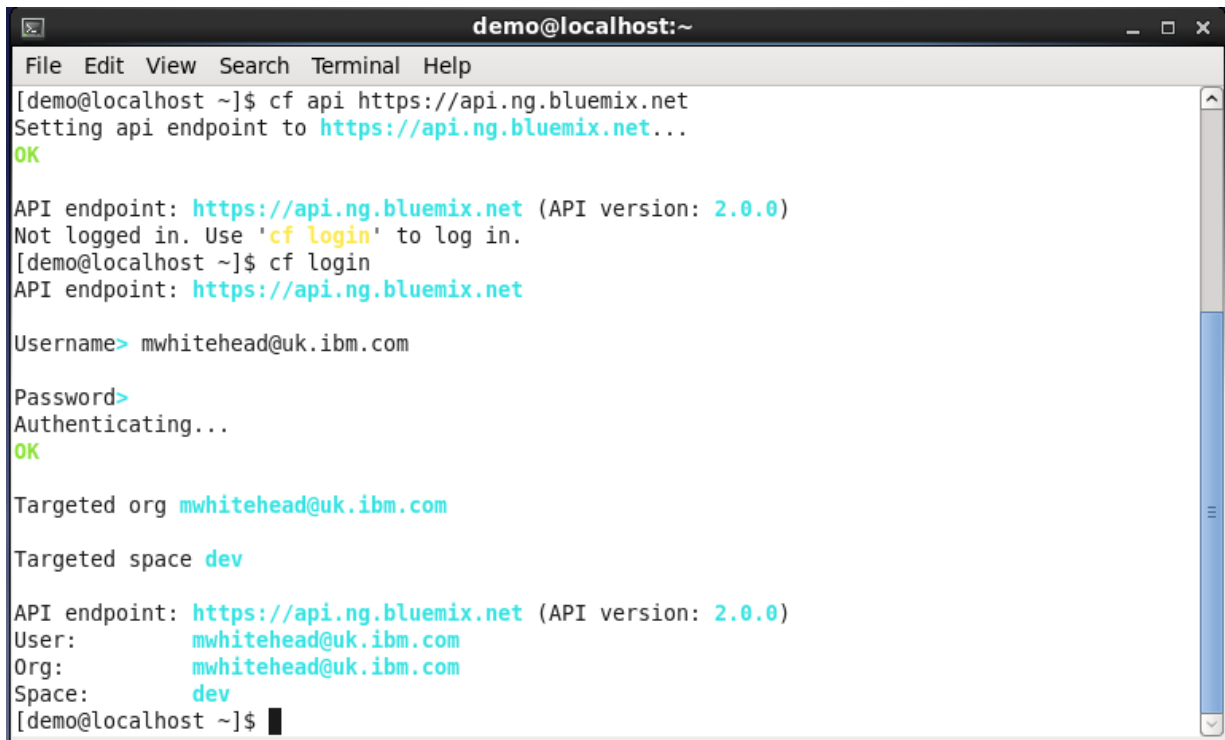
```
cf api https://api.ng.bluemix.net
```

This tells the cf tool to connect to the BlueMix cloud environment rather than another CloudFoundry environment.

Now enter the command

```
cf login
```

You will be asked to enter your BlueMix username and password. Once you have logged in you can enter the rest of the cf commands without needing to supply your credentials again.



```
demo@localhost:~
File Edit View Search Terminal Help
[demo@localhost ~]$ cf api https://api.ng.bluemix.net
Setting api endpoint to https://api.ng.bluemix.net...
OK

API endpoint: https://api.ng.bluemix.net (API version: 2.0.0)
Not logged in. Use 'cf login' to log in.
[demo@localhost ~]$ cf login
API endpoint: https://api.ng.bluemix.net

Username> mwhitehead@uk.ibm.com

Password>
Authenticating...
OK

Targeted org mwhitehead@uk.ibm.com

Targeted space dev

API endpoint: https://api.ng.bluemix.net (API version: 2.0.0)
User: mwhitehead@uk.ibm.com
Org: mwhitehead@uk.ibm.com
Space: dev
[demo@localhost ~]$
```

2. Navigate to the first sample application

In the same terminal window navigate back to the `/home/demo/mql/source/webapp-noworker.nodejs` directory we used earlier.

The `app.js` file should still contain your Twitter API keys from earlier.

3. Push the application to BlueMix

Run the following command in the terminal window:

```
cf push -c "node app.js" -n <CHOOSE A URL PREFIX HERE> webapp-noworker-nodejs
```

Note: You must include the double quotes in the command.

You must specify a URL prefix so that when the application is deployed BlueMix can create a unique hostname for it. Choose something like your initials

4. View your application running in BlueMix

In your browser open a tab and navigate to the URL which BlueMix generated for your application, e.g. <http://<your-prefix-here>.ng.bluemix.net/> demo web page which shows the tweets arriving on the left and graph being drawn on the right. You can see the same behaviour we saw when we ran the application locally. Each tweet that arrives has to be processed in the same thread so news tweets can't be shown as they are received.

Congratulations! You've run your first application in BlueMix!

Now let's improve it by using the offloading application and the worker application that we used earlier.

5. Stop the application

Run the command

```
cf stop webapp-noworker.nodejs
```

6. Push the web front end of the decomposed application

Using the same terminal window navigate to `/home/demo/mql/source/webapp-offload.nodejs` and push it to BlueMix using the command

```
cf push -c "node app.js" -n <YOUR URL PREFIX HERE> --no-start webapp-offload.nodejs
```

Notice how we use the `--no-start` directive to prevent the app from starting once it's deployed.

7. Create an instance of the Elastic MQ

An instance of Elastic MQ is required for our application to connect to.

Run the command

```
cf cs ElasticMQ Default mqsampleservice
```

This will create an Elastic MQ service called `mqsampleservice`.

8. Bind the application to the service.

This tells the application which Elastic MQ service to use. Advanced BlueMix users might have several different Elastic MQ services which they use for different applications. Even though we only have one service we still need to tell the application which service to bind to.

Run the command

```
cf bs webapp-offload.nodejs mqsampleservice
```

This will bind the application to the mqsampleservice.

9. Deploy the worker application and bind it to the same Elastic MQ service.

Navigate to `/home/demo/mql/source/worker.nodejs` in the terminal window.

Run the following commands to push the worker application into BlueMix and bind it with the same Elastic MQ service:

```
cf push -c "node worker.js" --no-start --no-route worker.nodejs
```

```
cf bs worker.nodejs mqsampleservice
```

10. Start both of the applications – the offloader and the worker

Run the following commands:

```
cf start worker.nodejs
```

```
cf start webapp-offload.nodejs
```

11. Scale up the number of worker applications

Because BlueMix manages our applications for us we can easily ask it to create more than one instance of them.

To tell BlueMix to create 3 more worker thread applications, run the command

```
cf scale worker.nodejs -i 4
```

Part 5 – Changing the worker application from node.js to JMS

So far both the offloading application and the workloader application have been written in node.js using the MQ Light API.

As well as node.js, BlueMix has first class support for deploying and running JMS applications.

The Elastic MQ service supports MQ Light apps and JMS apps binding to it, allowing you to communicate between applications written using either API.

In the final part of the lab we will change the worker application from an MQ Light application running in node.js to a JMS application running in Java. The offloading application will remain the same. All we're changing is the worker application.

1. Stop the two applications that we currently have running in BlueMix

Run the commands

```
cf stop worker.nodejs
```

```
cf stop webapp-offload.nodejs
```

You can check that no apps are running by using the command:

```
cf apps
```

2. Deploy the JMS application

In the terminal window navigate to */home/demo/mql/source/worker.JMS*

Using the same commands that we used earlier, push the JMS worker to BlueMix and bind it to the mqsampleservice:

```
cf pushJMS --no-start --no-route worker.JMS
```

```
cf bs worker.JMS mqsampleservice
```

The offloading application is still running so all we need to do start the offloader and the JMS worker:

```
cf start worker.JMS
```

```
cf start webapp-offload.nodejs
```

3. Check that the applications are running correctly

Use the **cf apps** command to check that your applications are running correctly.

In your browser navigate back to the demo web page and check that the Twitter feed is being updated as soon as tweets arrive, and that the graph is updating correctly.

Remember we now only have 1 instance of the JMS worker application – we need to tell BlueMix to scale up the number of worker applications again.

4. Scale the java workers

Run the same command that we used earlier to scale up the number of JMS workers:

```
cf scale worker.JMS -i 4
```

Check that the graph on the web page is updating more quickly as the worker threads cope with the workload more easily.

Congratulations! You've reached the end of the lab!

We've shown you how to use MQ Light to rapidly develop and debug your applications and you've learnt how MQ Light can make your applications more responsive.

You've also seen how the MQ Light API allows you to create multiple workers by joining them into a share, allowing you to build a scalable app that can respond to change in demand.

Finally you've seen how you can deploy your applications to BlueMix, leaving you to worry about coding your applications while BlueMix manages the infrastructure and scales your application quickly and easily.