

Developing a first application with IBM MQ Light: Lab Instructions

Lab Username = demo

Lab Password = sample

Authors:

Steve Upton, 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 using MQ Light and MQ Light Service in IBM Bluemix.

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 MQ Light Service in IBM Bluemix are a 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.

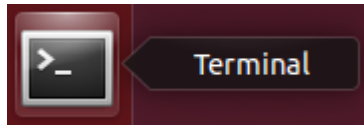
You will also learn how to use the MQ Light user interface to understand and debug the messages that your application has sent and received.

What the lab covers

This lab has the following parts:

1. Running the sample applications.
2. Running and improving an application by separating the web-facing component from the data-processing component.
3. Scaling up to demand with more workers and different languages
4. Deploying your finished application to IBM Bluemix to run it in the cloud using the MQ Light service.

Setup – Extract the source for the labs



1. Open a terminal window and navigate to `/home/demo/mql` by running:
`cd /home/demo/mql`
2. Extract the source files from git by typing “`./extract-lab.sh`” and hitting enter.

This should extract the projects that we will be using to run the lab into the lab directory.

Note: If step 2 failed, for example because of network problems, there is a backup copy of the lab files in `/home/demo/mql/backup-lab-files`

From `/home/demo/mql` run

```
cp -r backup-lab-files/ lab
```

To copy the lab files to the lab directory and continue as normal.

3. If you wish to run the Bluemix parts of the lab you will need to get a Bluemix ID. We suggest you sign up for an account at www.bluemix.net so you can complete the Bluemix sections.

You should now be ready to run the lab!

Part 1 – Running the sample applications

First, we'll start MQ Light and run some sample applications to show the basics of how it works.

1. Start MQ Light

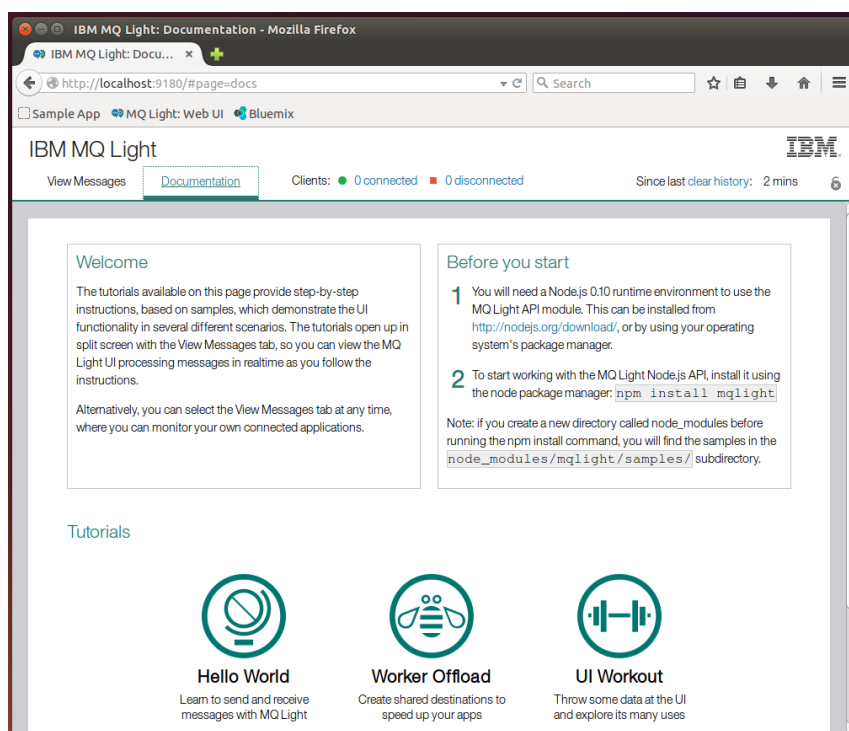
Open a terminal, change run `cd /home/demo/mql/mqlight` and run:

```
./mqlight-start
```

This will start up the MQ Light runtime so other applications can connect to it and use it to send messages. If this is your first time running MQ Light, you will need to press 1 and hit enter to accept a licence, then type 'N' to the security questions as we won't need them for this lab.

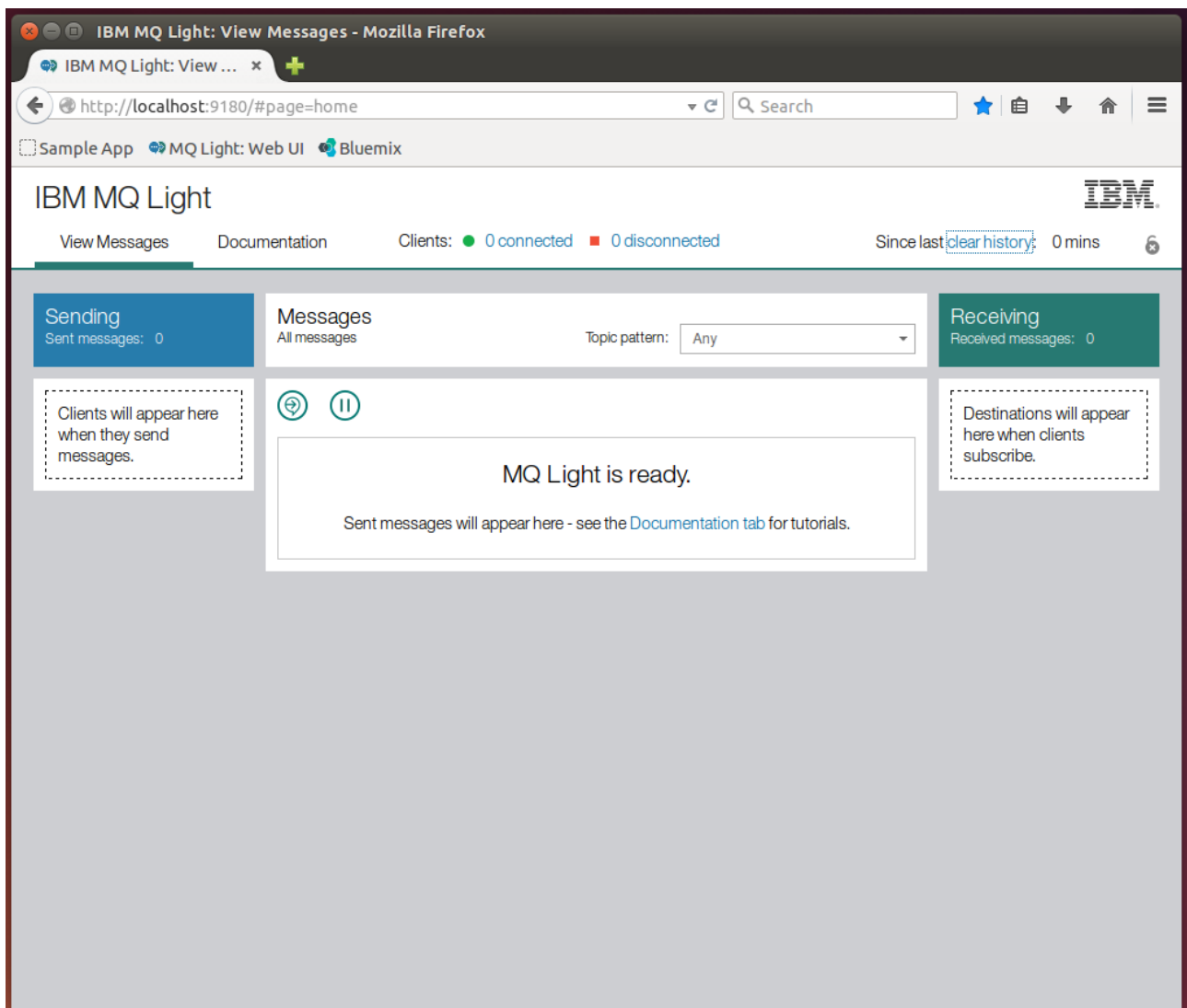
2. Check the MQ Light UI

Once MQ Light has started up, a browser window will open which will look like this:



The Documentation tab explains how to run some sample apps that we will be going through shortly, as well as how to install some MQ Light client

libraries. Click the View Messages tab in the top left and you will be shown the main view of the UI:



This screen shows the current state of the MQ Light runtime, including connected clients and the status of messages. We'll be referring back to and explaining this screen throughout the lab.

3. Start a Node.js receiver application

Back in a terminal, navigate to `/home/demo/mql/samples/node`. We have a `node_modules` directory as we have already installed the node modules needed to run these samples, so you can go right ahead and run:

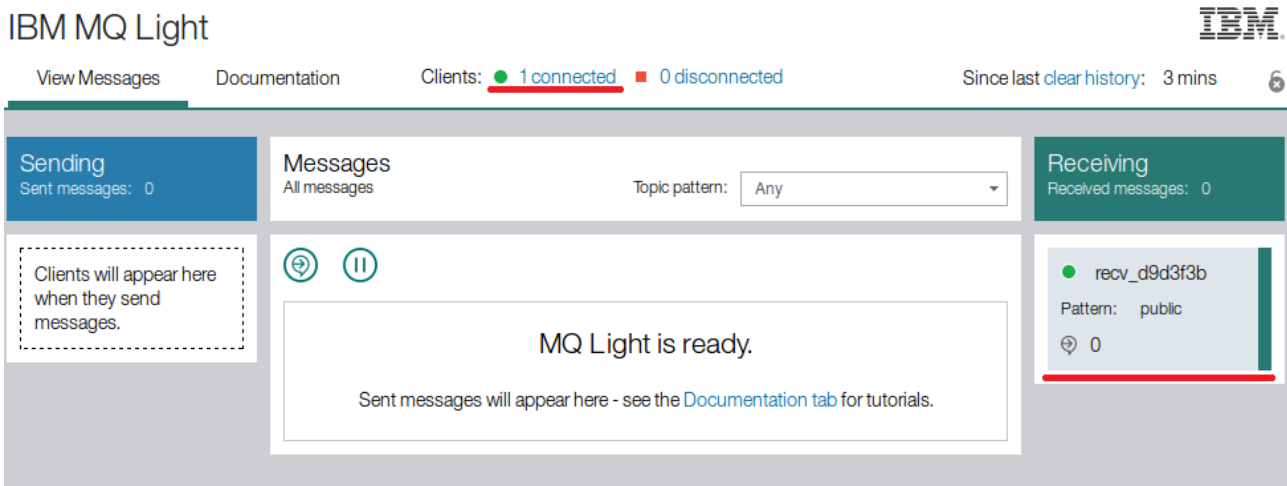
```
node recv.js
```

Which will start a simple node application that will subscribe to the topic

pattern public

```
demo@ubuntu: ~/mql/samples/node
demo@ubuntu:~/mql/samples/node$ node recv.js
Connected to amqp://localhost:5672 using client-id recv_d9d3f3b
Subscribed to pattern: public
```

When a message is sent to the topic public, it will be displayed here. Switch back to the MQ Light UI, we can see one connected client listed as a receiving client (because it has subscribed to a destination):



4. Send a message to the receiver application

Open a new terminal window by right clicking the terminal icon → New Terminal and navigate to the `/home/demo/mql/samples/node` directory (leave the `recv` app running in the original window) and run the command:

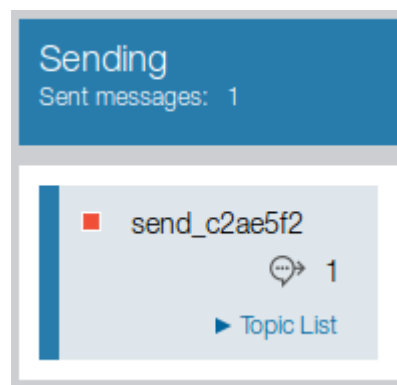
```
node send.js
```

This application will send a message containing the text “Hello World!” to the public 'public', then exit:

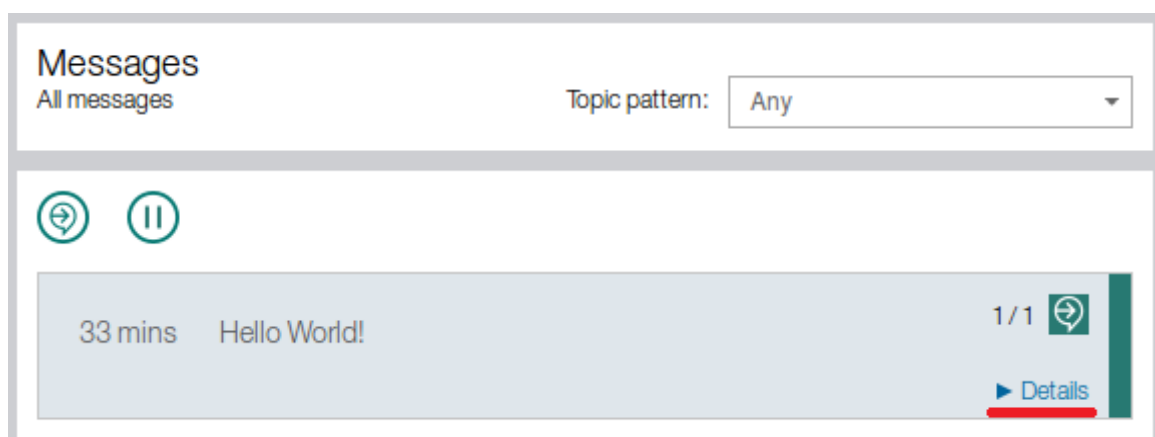
```
demo@ubuntu: ~/mql/samples/node
demo@ubuntu:~$ cd mql/samples/node/
demo@ubuntu:~/mql/samples/node$ node send.js
Connected to amqp://localhost:5672 using client-id send_c2ae5f2
Sending to: public
Hello World!
demo@ubuntu:~/mql/samples/node$
```

Checking the UI, we can see that one sending client is listed as disconnected (because it disconnected right after sending the message):

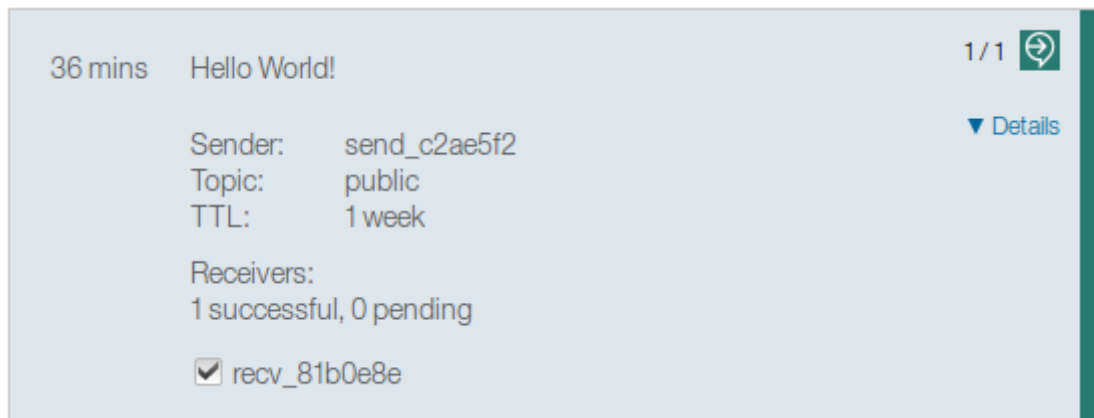
Clients: ● 1 connected ■ 1 disconnected



In the centre of the window, we can see the message that has been sent:



Clicking the details button (highlighted) in the bottom right gives us more details on the sent message:



Here, we can see the payload of the message was 'Hello World!', the sender's id, the topic it was sent to and the time to live (TTL) of the message. We can also see the clients it was successfully delivered to (marked with ticks). If you move the mouse to hover over the icon in the top right, it will tell you that the message has been successfully received.

Now go back to the terminal window where you ran the recv application and we can see that it has received the message:

```
demo@ubuntu: ~/mql/samples/node
demo@ubuntu:~/mql/samples/node$ node recv.js
Connected to amqp://localhost:5672 using client-id recv_81b0e8e
Subscribed to pattern: public
Hello World!
█
```

5. Send a message from a Ruby client to a Node.js client.

Let's send a message from Ruby now. Keeping the original recv app running, open another terminal window and navigate to /home/demo/mql/samples/ruby

From here, run:

```
ruby send.rb
```

As with the Node sender, this will send a message to the topic 'public' and then exit:


```
demo@ubuntu: ~/mql/samples/ruby
demo@ubuntu:~/mql/samples/ruby$ ruby send.rb
Connected to amqp://localhost:5672 using client-id send_3795a02
Sending to: public
Hello World!
demo@ubuntu:~/mql/samples/ruby$
```

Checking the UI, you should see things update much as before. Looking at the Node.js recv app, we can see this message came in as well. Just like that, we have a Node.js and Ruby application talking to each other.

6. Identify a message sent to the wrong topic

So far all the messages we've been sending have ended up at their intended destinations, but what if we get something wrong? Run the Ruby send application again, but this time with another option:

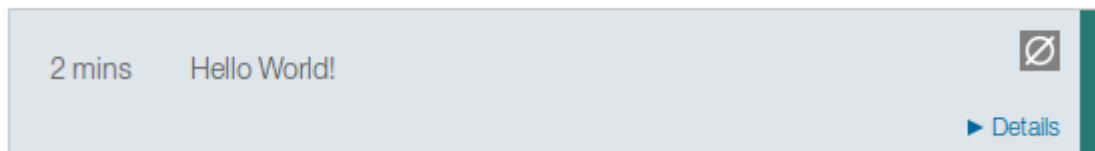
```
ruby send.rb -t wrongtopic
```

The -t option specifies the topic the message will be sent to (in unset, it defaults to public):

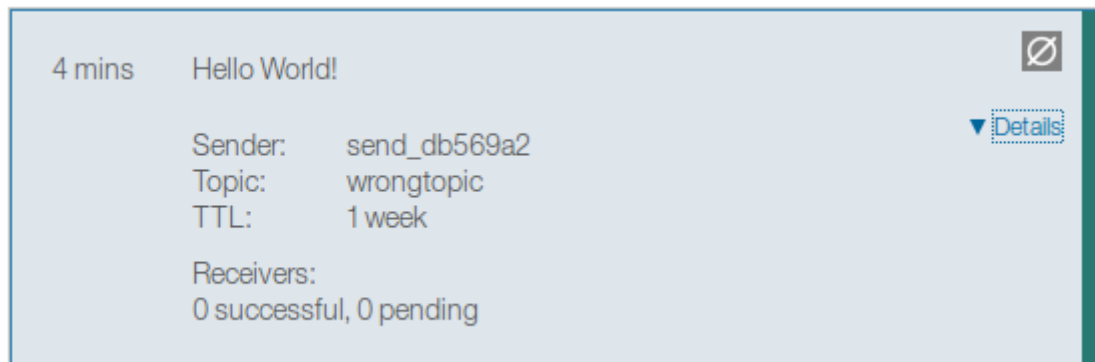
```
demo@ubuntu: ~/mql/samples/ruby
demo@ubuntu:~/mql/samples/ruby$ ruby send.rb -t wrongtopic
Connected to amqp://localhost:5672 using client-id send_db569a2
Sending to: wrongtopic
Hello World!
demo@ubuntu:~/mql/samples/ruby$
```

The recv app we have running is listening on public, so this message won't get through to it. This is the kind of problem you might encounter when developing messaging apps (as we'll be doing in the next section), so the MQ Light UI is designed to help you track down these common problems.

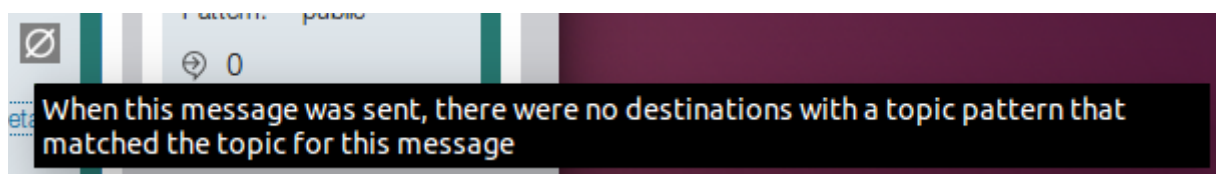
Imagine you sent that message, expecting it to be delivered to a receiver, but you mistyped the topic string and it doesn't arrive. Checking the MQ Light UI will give you info on what went wrong:



At a glance, we can see a different icon in the top right showing that something is wrong (we'll discuss what shortly) but also that message has been sent and picked up by MQ Light, ruling out network issues between the sender and MQ Light. Clicking the details button gives us:



Now, we can see at the bottom that there are no receivers listed (successful or pending). If we had pending receivers, this would indicate an issue communicating with the receiver, helping us narrow down the issue further. If we mouse over the icon in the top right, we can get the information we need to solve the issue:



This tells us that message was sent to a topic pattern that no-one was listening to. We can then check the topic of the message (`wrongtopic`) and compare it to the topic of the receiver (`public`) and clearly see the issue – we sent the message to the wrong topic – an easy mistake that MQ Light helps developers to track down and fix.

Now, let's look at an app that can be improved using MQ Light.

(You can now stop the original `recv.js` app by switching to the terminal window in which it is running and pressing `ctrl-c`)

Part 2 – Running and improving an application by separating the web-facing component from the data-processing component.

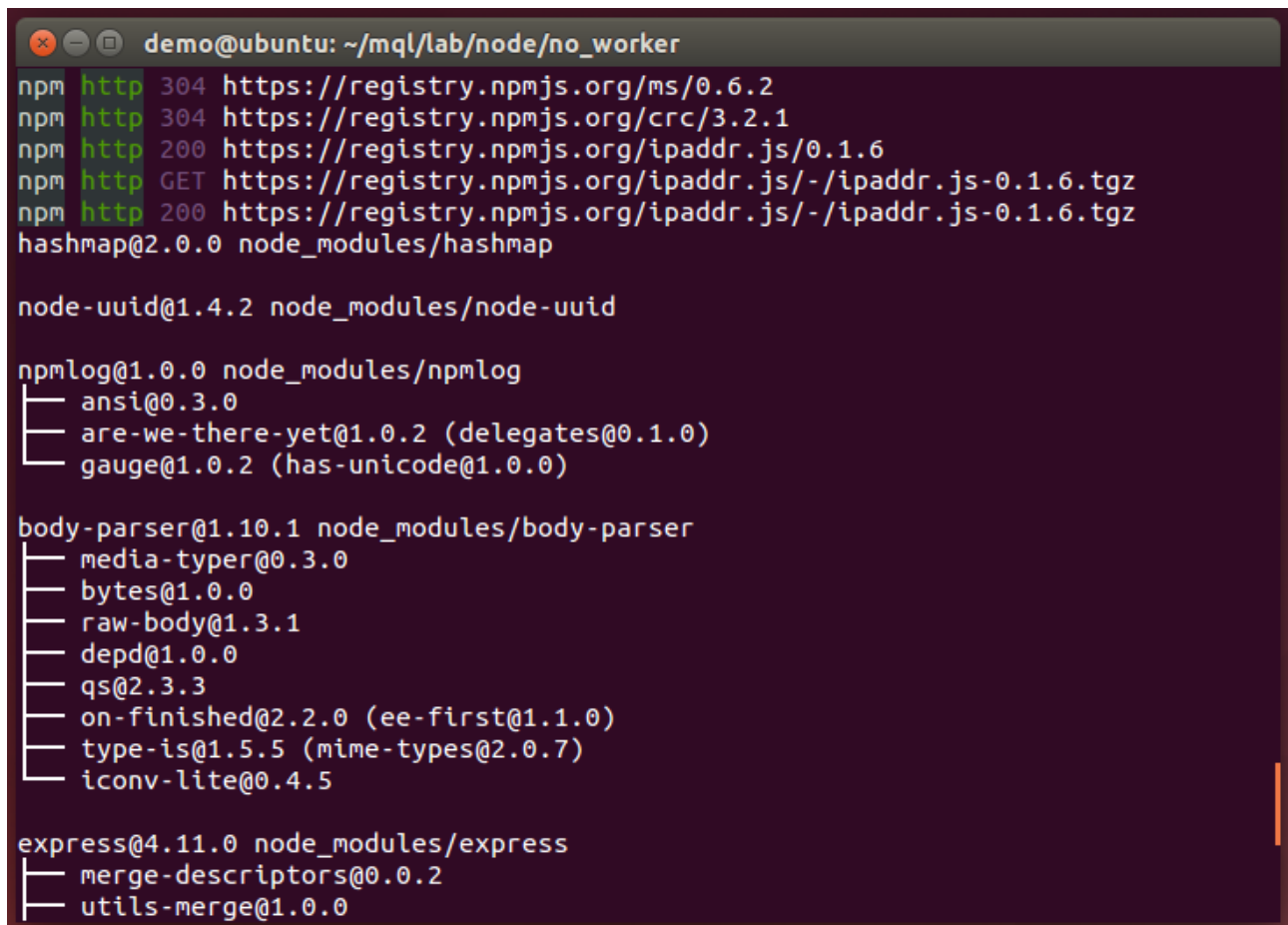
1. Run the application that needs improvement

In a terminal window, navigate to `/home/demo/mql/lab/node/no_worker`.

Here we can see the application files, but before we can run it, we'll need to run:

```
npm install
```

This will read the application's dependencies from `package.json` and install all the necessary modules:

A terminal window titled 'demo@ubuntu: ~/mql/lab/node/no_worker' showing the output of 'npm install'. The output lists various modules being installed from the npm registry, including http, https, ipaddr.js, hashmap, node-uuid, npmlog, ansi, are-we-there-yet, gauge, body-parser, media-typer, bytes, raw-body, depd, qs, on-finished, type-is, iconv-lite, express, merge-descriptors, and utils-merge. The modules are listed with their versions and the path to their node_modules directory.

```
demo@ubuntu: ~/mql/lab/node/no_worker
npm http 304 https://registry.npmjs.org/ms/0.6.2
npm http 304 https://registry.npmjs.org/crc/3.2.1
npm http 200 https://registry.npmjs.org/ipaddr.js/0.1.6
npm http GET https://registry.npmjs.org/ipaddr.js/-/ipaddr.js-0.1.6.tgz
npm http 200 https://registry.npmjs.org/ipaddr.js/-/ipaddr.js-0.1.6.tgz
hashmap@2.0.0 node_modules/hashmap

node-uuid@1.4.2 node_modules/node-uuid

npmlog@1.0.0 node_modules/npmlog
├─ ansi@0.3.0
├─ are-we-there-yet@1.0.2 (delegates@0.1.0)
├─ gauge@1.0.2 (has-unicode@1.0.0)

body-parser@1.10.1 node_modules/body-parser
├─ media-typer@0.3.0
├─ bytes@1.0.0
├─ raw-body@1.3.1
├─ depd@1.0.0
├─ qs@2.3.3
├─ on-finished@2.2.0 (ee-first@1.1.0)
├─ type-is@1.5.5 (mime-types@2.0.7)
├─ iconv-lite@0.4.5

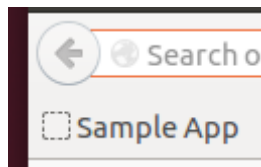
express@4.11.0 node_modules/express
├─ merge-descriptors@0.0.2
├─ utils-merge@1.0.0
```

We can now run the app using:

```
node app.js
```

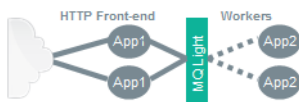
```
demo@ubuntu: ~/mql/lab/node/no_worker
demo@ubuntu:~/mql/lab/node/no_worker$ node app.js
App listening on *:3000
```

The app is listening on localhost:3000, switch to your browser and open that page in a new tab (keep the MQ Light UI open, as we'll be using it later). You can also click the button in the bookmark toolbar to jump straight to localhost:3000:



Sample MQ Light Service for Bluemix Application:- Worker Offload Pattern

This sample demonstrates a simple worker-offload pattern using the MQ Light messaging service.



The worker-offload pattern improves responsiveness by allowing a front-end user interface delegate work to one or more back-end worker instances.

Type a sentence in the box and press submit. The sample will:

1. Send the sentence from the browser to the front-end app using a HTTP POST
2. Send each word in the sentence from the front-end app to the workers using MQ Light
3. Invoke a worker for each word, which converts the word to upper-case
4. Send a notification containing the upper-case word from the worker to the front-end using MQ Light
5. Send each notification from the front-end app to the browser using a polling HTTP GET
6. Display the upper-case words in this page as they arrive

Some things to note:

- Messages might not arrive in the order in which they are sent.
This is because each worker can take a variable amount of time to process a message.
- If you open this page in multiple browsers, then only one will see each word.
This is because in this sample they all share a single durable subscription for notifications.
- If you run both the Node.js and Liberty for Java back-end workers, then you will see two notifications for each word.
This is because the sample workers use different durable subscriptions.
- HTTP GET polling of notification messages is uncommon in real apps.
Apps normally process notifications as they arrive in the front-end, and update state in a database or other state store.

Notifications from the Node.js sample back-end look like this

Notifications from the Liberty for java sample back-end look like this

Sentence:

One, Two, Three, Four, Five, Once I Caught a Fish Alive

Submit Work

You can ignore the text for now (it is written assuming you are already using

MQ Light). When you click the 'Submit Work' button, the words in the text box are sent to the Node app by way of a HTTP POST, where they are 'processed' and sent back to the webpage using HTTP where they are displayed. The processing involves capitalizing the words and sleeping for a short time to simulate the more intensive processing that might need to take place in a real application.

The problem is that although the capitalised words are displayed, there is a significant lag before they appear:

Sentence:

One, Two, Three, Four, Five, Once I Caught a Fish Alive

Submit Work

ALIVE FISH A CAUGHT I ONCE FIVE, FOUR, THREE, TWO, ONE,

This wait is due to the fact the Node.js is single threaded by design and all time consuming processing (or waiting, in this case) will hold up the rest of the app. This is bad, as you always want your apps to be as responsive as possible – luckily, it's exactly the kind of problem MQ Light can help with.

If we move the time consuming processing to a separate application, it can take its time processing while the front-end app is free to be as responsive as possible. Normally, it is not possible to communicate between Node.js threads, but by running the samples, we've seen how MQ Light can send messages between different Node.js applications – we can use this move the heavy work to separate 'worker' apps.

2. Update the front-end app to send its work away using MQ Light, instead of processing it locally

Open the original app.js in gedit by doubling clicking on the file in the file browser and make sure you have the following 2 lines at the top of the file so they can be used later:

```
var PUBLISH_TOPIC = "mqlight/sample/words";
```

```
var SUBSCRIBE_TOPIC = "mqlight/sample/wordsuppercase";
```

The SUBSCRIBE_TOPIC is the topic we'll be listening for messages from the worker on and the PUBLISH_TOPIC is the topic where we'll send our work to be processed.

Now, paste the following code near the top of the file, after all the require statements:

```
var mqlight = require('mqlight');  
var opts = {};  
opts.service = 'amqp://localhost:5672';
```

This will use the MQ Light module and configure the start-up options to connect to the default MQ Light address.

Next, we'll add code to create an MQ Light client:

```
var mqlightClient = mqlight.createClient(opts, function(err) {  
  if (err) {  
    console.error('Error ' + err);  
  } else {  
    console.log('Connected with id ' + mqlightClient.id);  
  }  
});
```

Finally, we can add the code to actually send the worker. Look for the line

```
processMessage(msgData);
```

in `app.post('/rest/words'` ... which is where we process. All we need to do is replace that line with:

```
mqlightClient.send(PUBLISH_TOPIC, msgData);
```

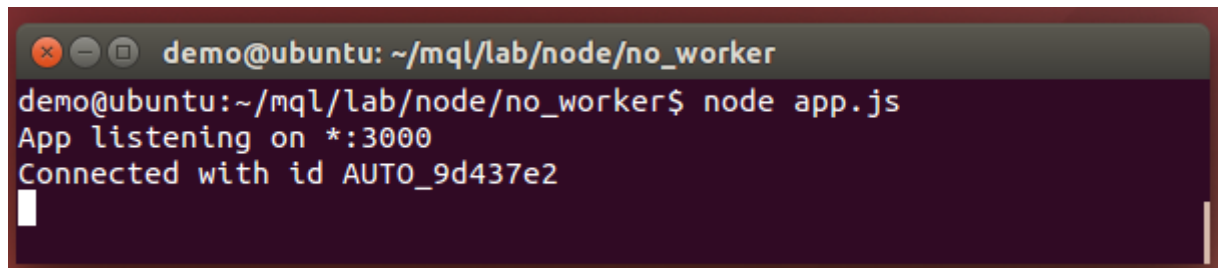
Before we can run this, we'll need to run:

```
npm install mqlight
```

In the `/home/demo/mql/lab/node/no_worker` directory. This will install the MQ Light node module you'll need to connect your apps to MQ Light. You can now run the updated front-end with:

```
node app.js
```

You should see it start as before, but now it will also report that is connected to MQ Light and print its client id:

A terminal window with a dark background and light text. The title bar shows 'demo@ubuntu: ~/mql/lab/node/no_worker'. The prompt is 'demo@ubuntu:~/mql/lab/node/no_worker\$'. The command 'node app.js' has been entered. The output shows 'App listening on *:3000' and 'Connected with id AUTO_9d437e2'. A cursor is visible on the line following the output.

```
demo@ubuntu: ~/mql/lab/node/no_worker
demo@ubuntu:~/mql/lab/node/no_worker$ node app.js
App listening on *:3000
Connected with id AUTO_9d437e2

```

If point your browser at `localhost:3000` again, you should see the same interface as before, but now if you press the 'Submit Work' button nothing will be output on the screen. If we check the MQ Light UI, we can confirm that messages were sent, but no worker received them.

Sending
 Sent messages: 11

Messages
 All messages

Topic pattern:

● AUTO_9d437e2
 11
 ▶ Topic List

2 mins	{"word": "Alive", "frontend": "Node.js"}	Details
2 mins	{"word": "Fish", "frontend": "Node.js"}	Details
2 mins	{"word": "a", "frontend": "Node.js"}	Details
2 mins	{"word": "Caught", "frontend": "Node.js"}	Details

This is expected because we don't have a worker to do the processing and send it back to the front-end. Let's make that worker now.

3. Move the time intensive work to a separate worker app.

In app.js look for the processMessage function:

```

/*
 * Handle a word sent from the web page
 */
function processMessage(data) {
  var word = data.word;
  try {
    // Convert JSON into an Object we can work with
    data = JSON.parse(data);
    word = data.word;
  }
}

```

Here, we are taking the word sent from the web page and performing the (time intensive) processing on it. This is what it slowing the app down, so let's move it to a separate app. Create a file in the same directory called worker.js and cut and paste it (and the sleep function below it) into that file:


```

/*
 * Handle a word sent from the web page
 */
function processMessage(data) {
  var word = data.word;
  try {
    // Convert JSON into an Object we can work with
    data = JSON.parse(data);
    word = data.word;
  } catch (e) {
    // Expected if we already have a Javascript object
  }
  if (!word) {
    console.error("Bad data received: " + data);
  }
  else {
    console.log("Received data: " + JSON.stringify(data));
    sleep(500); // This blocks the node worker thread for 1 second
    // you would normally never do this. We are doing it to _simulate_
    // a complex algorithm that takes a long time to run.

    // Upper case it and publish a notification
    var replyData = {
      "word" : word.toUpperCase(),
      "backend" : "Node.js"
    };
    // Convert to JSON to give the same behaviour as Java
    // We could leave as an Object, but this is better for interop
    replyData = JSON.stringify(replyData);
    console.log("Sending response: " + replyData);
    heldMsgs.push({"data" : replyData});
  }
}

function sleep(time) {
  var end = new Date().getTime();
  while(new Date().getTime() < end + time) {;}
}

```

4. Set up the worker to receive work from the front end.

In isolation, this code won't do anything, so we'll need to add some code that will allow it to connect to MQ Light. Copy the following code to the top of the file, much as we did for the front-end:

```
var mqlight = require('mqlight');
var opts = {};
opts.service = 'amqp://localhost:5672';
```

Next, we'll define some variables based on the topics the worker will be using:

```
var SUBSCRIBE_TOPIC = "mqlight/sample/words";
var PUBLISH_TOPIC = "mqlight/sample/wordsuppercase";
var SHARE_ID = "node-back-end";
```

The SUBSCRIBE_TOPIC is the topic we'll be listening for messages from the front-end on and the PUBLISH_TOPIC is the topic we'll send back our processed message on. We'll talk about share SHARE_ID later.

Now add the following code, which will create an MQ Light client, have it subscribe to the correct topic and then send the messages it receives to the processMessage function we copied over earlier:

```
var mqlightClient = mqlight.createClient(opts, function(err) {
  if (err) {
    console.err('Error: ' + err);
  } else {
    console.log('Connected');
  }
  mqlightClient.on('message', processMessage);
  mqlightClient.subscribe(SUBSCRIBE_TOPIC, function(err) {
    if (err) console.err("Failed to subscribe: " + err);
    else { console.log("Subscribed"); }
  });
});
```

In the 'message' event (ie. When a message arrives), the message payload is passed straight to the processMessage function, where it will be processed just as in the original. Now we just need to make it send the processed word back to the front-end.

5. Set up the worker to send work back to the front end.

Looking at the processMessage function, all we need to change is the final line from:

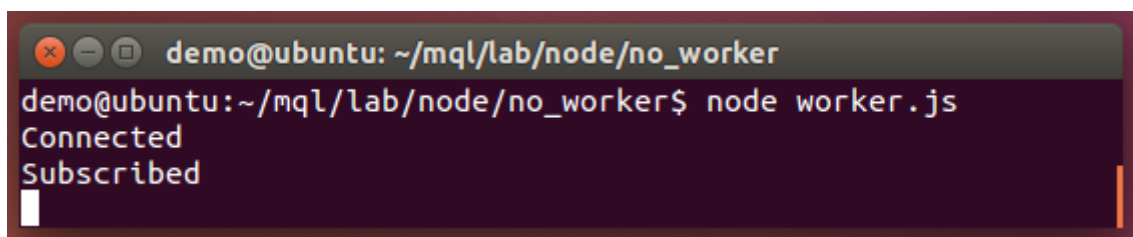
```
heldMsgs.push({"data" : replyData});
```

to:

```
mqLightClient.send(PUBLISH_TOPIC, replyData);
```

In the original app, we pushed the data to an array from which it was popped and sent to the web-page, but now, we just need to send it to the front-end using MQ Light. We can run the worker app now with

```
node worker.js
```

A terminal window with a dark background and light text. The title bar shows 'demo@ubuntu: ~/mql/lab/node/no_worker'. The prompt is 'demo@ubuntu:~/mql/lab/node/no_worker\$'. The command 'node worker.js' has been entered and executed, resulting in two lines of output: 'Connected' and 'Subscribed'. A cursor is visible on the line following 'Subscribed'.

```
demo@ubuntu: ~/mql/lab/node/no_worker
demo@ubuntu:~/mql/lab/node/no_worker$ node worker.js
Connected
Subscribed
```

Leave this running and (ensuring your updated front-end app.js is also running) point your browser back to localhost:3000 and remove all but one



word from the text box (this will allow us to better keep track of what is being sent through MQ Light).

Sentence:

One

Submit Work

Click the Submit Work button. Again, nothing will be output on the screen. Check the MQ Light and expand the message details.

1 min	<code>{"word":"ONE","backend":"Node.js"}</code>	
Sender: AUTO_ae19102		▼ Details
Topic: mqlight/sample/wordsuppercase		
TTL: 1 week		
Receivers: 0 successful, 0 pending		
<hr/>		
1 min	<code>{"word":"One","frontend":"Node.js"}</code>	1 / 1 
Sender: AUTO_870a492		▼ Details
Topic: mqlight/sample/words		
TTL: 1 week		
Receivers: 1 successful, 0 pending		
✓ AUTO_ae19102		

The bottom (earliest) message is from the front-end to the worker. We can now see that it has been successfully delivered to the worker as expected. The newer (top) message is the processed (capitalized) word from the worker. We can see that it has been sent, but not received by any clients. Let's fix that.

6. Update the front-end to receive the processed message from the worker

Looking back at app.js, we need to add a function that will get the received message where it needs to be:

```
function handleProcessedMessage(msg) {  
    heldMsgs.push({"data" : msg});  
}
```

Then, looking back at where we create the MQ Light client, we'll need to add a bit of code to that block that will subscribe to the correct topic and send any received messages to the `handleProcessedMessage` function. Copy in the code such your file looks like this:

```
var mqlightClient = mqlight.createClient(opts, function(err) {  
    if (err) {  
        console.error('Error ' + err);  
    } else {  
        console.log('Connected with id ' + mqlightClient.id);  
    }  
  
    mqlightClient.on('message', processMessage);  
    mqlightClient.subscribe(SUBSCRIBE_TOPIC , function(err) {  
        if (err) console.err("Failed to subscribe: " + err);  
        else { console.log("Subscribed"); }  
    });  
});
```

Finally, re-run your app.js and press the Submit Work button. The words should now appear on screen.

Sentence:

One, Two, Three, Four, Five, Once I Caught a Fish Alive

Submit Work

TWO, THREE, FOUR, FIVE, I A ALIVE FISH CAUGHT ONCE ONE,

If you check the MQ Light UI, you can see all messages listed as received. Importantly, we no longer have a huge lag before messages start appearing because we aren't holding up the main thread with all the processing. The messages still take time to come in though (the processing still needs to be done, of course). We can address this issue by scaling up the number of workers, which will be the subject of our next section.

Part 3 – Scaling up to demand with more workers and different languages

Currently, we have a Node.js front-end and a Node.js worker. The CPU intensive work has been moved to worker app, making the front-end more responsive, but the work still takes roughly the same length of time. We can improve on this by scaling up the number of workers we are using. By having multiple workers processing each word in parallel, they can complete more work in the same amount of time.

1. Switch to the pre-made front-end and worker.

If you followed the previous section, will switch to a new, ready-made version of the front-end and worker apps. We've made a few small changes to the apps that we'll talk about in the next section.

First, stop any of the Node.js apps you currently have running by switching to the terminal window in which they are running and pressing ctrl+c. You can then close those windows.

Open up 2 new terminal windows. In one, navigate to `/home/demo/mql/lab/node/worker_frontend/` and in the other navigate to `/home/demo/mql/lab/node/worker_backend/`. These are the front-end app and back-end (worker) app, respectively and we'll refer to them as such from now on.

2. Examine the differences in the new worker.

To start with, in the `worker_frontend` directory, run:

```
npm install
```

followed by:

```
node app.js
```

This will start the front-end app. You can leave this running for the remainder of this section.

In your version the worker thread, we subscribed to a private destination with a topic pattern 'mqlight/sample/words'. This means that it receives a copy of every messages published to that topic. If we simply started another instance of that worker application, both instances would receive a copy of every word. This would mean that each word would get 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 “shared destination”. A shared destination allows a group of applications to share the processing of messages arriving at the destination. Each message arriving at the destination is only given to one of them.

In the original worker app, we subscribed using this line of code:

```
mqlightClient.subscribe(SUBSCRIBE_TOPIC, function(err) {
```

The first argument is the topic we will subscribe to and the last argument is the function that will be run when the subscribe operation completes. We'll need to change this if we want to use a shared destination. If we look at the subscribe call in the new worker_backend/app.js we see:

```
mqlightClient.subscribe(SUBSCRIBE_TOPIC,  
    SHARE_ID,  
    {credit : 5, autoConfirm : true, qos : 0},  
    function(err) {
```

The first and last arguments are the same. The third argument lists some options. Many of these are default values and relate to how many messages we will take on before waiting to process what we have (`credit`), whether or not we automatically confirm that we have received a message (`autoConfirm`) and the quality of service (`qos`). Importantly, the second argument specifies a name for the shared destination we will be subscribing to. In this case, the `SUBSCRIBE_TOPIC` is `node-back-end`.

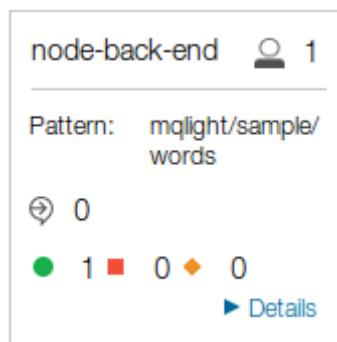
In the worker_backend directory, run:


```
npm install
```

followed by:

```
node app.js
```

Now, switch to the MQ Light UI and look in the Receiving column on the right of the screen:

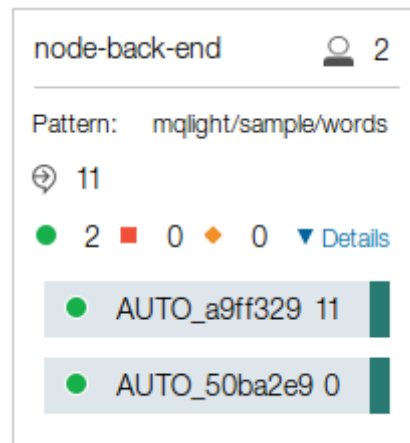


You'll notice this looks different to the apps you've run previously, because this is a shared destination. The head icon in the top right shows number of clients connected to this shared destination. The shared destination name (node-back-end) is listed at the top, as is the actual topic pattern it is subscribed to.

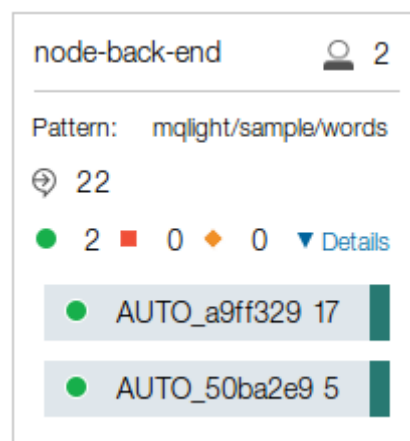
If we open the web-app itself (still on localhost:3000), we should see the same app and if we press the Submit Work button, we should see the same speed as when we were using the our worker app. This is because we still have only one worker app – shared destinations shine when we have more than one worker app. Open another terminal window, navigate to the worker_backend directory, and again run:

```
node app.js
```

Check the MQ Light UI again and we can see the node-back-end share has updated to reflect that another client has connected to it. Click the Details button to get more details:



Here can see the ids of both clients connected to this shared destination and how many messages they have each processed (the 11 is the messages we just sent by pressing the Submit Work button). Heading back to the web app, click the Submit Work button again and we should see better performance. Importantly, if we look at the MQ Light UI again, we can see that the messages have been distributed between the clients, while at the same time, no work has been duplicated.



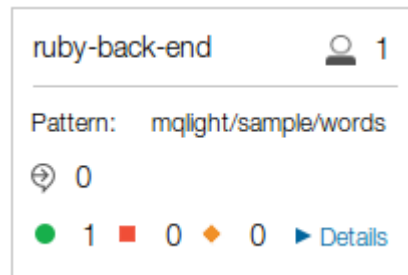
If you choose to, you can now start even more instances of the worker app and observe how they improve performance. With MQ Light, we have improved an app so its performance is now only bound by the physical machines it is running on, rather than the single threaded nature of Node.js.

3. Run a Ruby worker alongside the Node.js worker.

We can also run a Ruby worker app alongside the Node.js ones we are already running. To do so, open another terminal window and navigate to `/home/demo/mql/lab/ruby/backend`. Here we have a Ruby worker application that does the same as the Node.js workers. Run this app with:

ruby Sinatra_backend.rb

Check the Web UI and we can see the Ruby app has subscribed to another shared destination:



As this is a different shared destination, it means each word will be processed twice, once by the node-back-end shared destination and once by the ruby-back-end shared destination. Click the Submit Work button in the web app and observe the results:

Sentence:

One, Two, Three, Four, Five, Once I Caught a Fish Alive

Submit Work

ONE, ONE, TWO, FOUR, I CAUGHT FISH THREE, FIVE, ONCE A ALIVE TWO,
THREE, FOUR, FIVE, ONCE I CAUGHT A FISH ALIVE

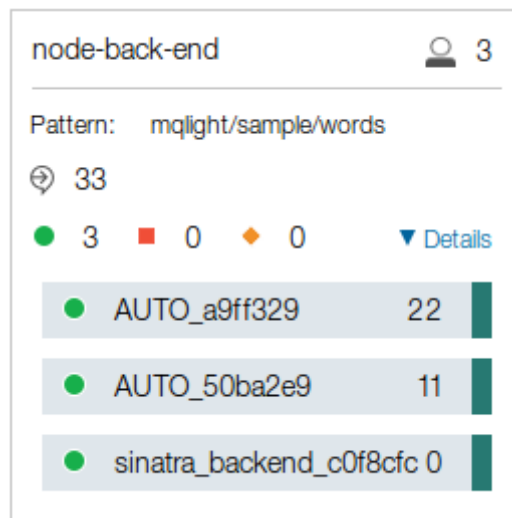
Here, we can see each word has been processed twice by Ruby (red border) and Node.js (blue border). If we wanted each word to be processed only once, we just need to make the Ruby worker subscribe to the same shared destination as the Node.js workers. To do that, simply open /home/demo/mq1/lab/ruby/backend/sinatra_backend.rb in a text editor and change line 18 from:

```
SHARE_ID = 'ruby-back-end'
```

to

```
SHARE_ID = 'node-back-end'
```

Then restart the app (ctrl+c, then run it again) and the app will connect to the same share as the Node.js workers:



Finally, click the Submit Work in the web app and we can see the work distributed between Ruby and Node.js workers:

Sentence: One, Two, Three, Four, Five, Once I Caught a Fish Alive

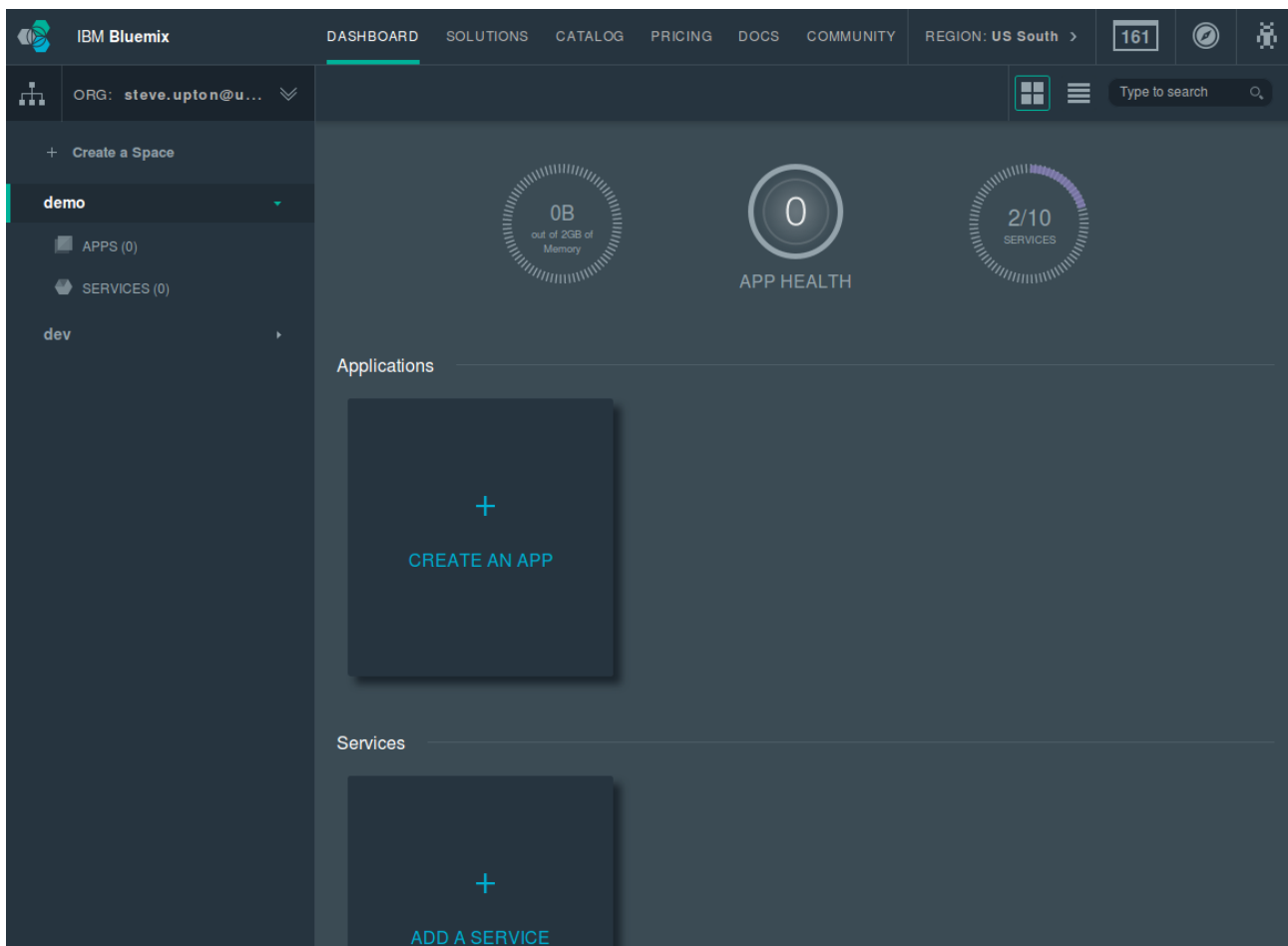
ONE, TWO, FOUR, THREE, ALIVE, ONCE, CAUGHT, A, FIVE, I, FISH

So far in this lab, we have taken single Node.js app that was unresponsive, improved it using MQ Light to make it much more scalable and responsive. We have even distributed work between Node.js and Ruby worker apps, empowering programmers with different skillsets to work on the same problem using the worker-offload pattern. The final step is to push it to production.

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

Now we have an application running locally, we are ready to push it to Bluemix. There, it will run in the cloud, exactly as it did locally, with services such as MQ Light, runtimes such as Node.js and Ruby being managed for you.

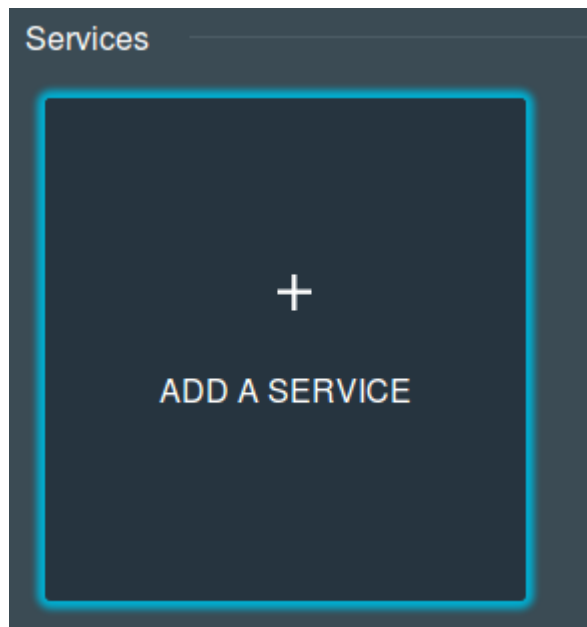
To do this part of the lab you will need to sign up to IBM Bluemix which you can do at <https://bluemix.net/>. Once you have signed up and logged in to Bluemix you will be presented with the Bluemix dashboard:



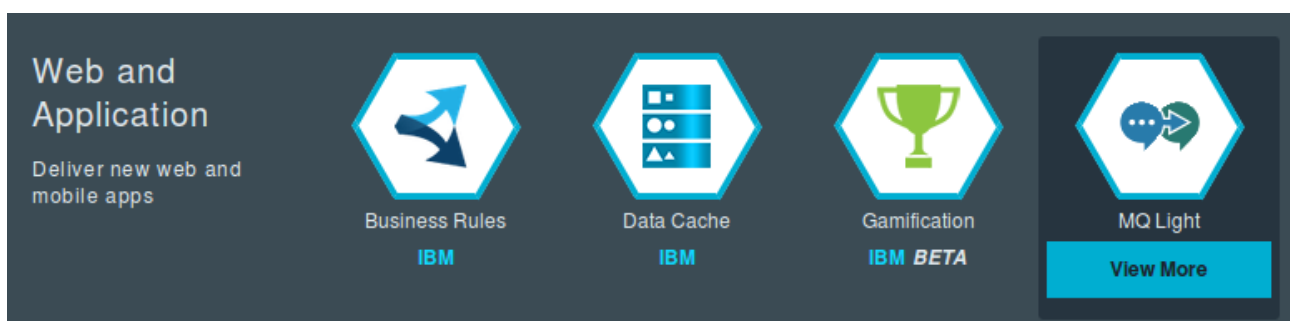
You will start with zero apps and zero services, so let's change that by getting an MQ Light service.

1. Create an MQ Light service instances

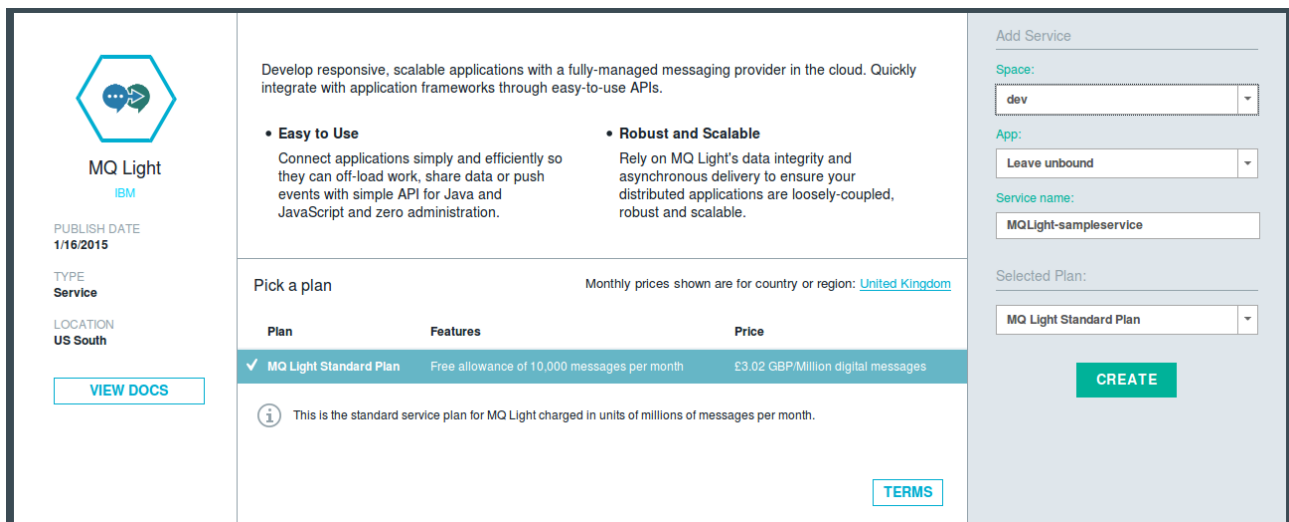
We're going to MQ Light service called MQLight-sampleservice that our apps can connect to when we push them to Bluemix. Scroll down to the Services section of the Bluemix dashboard and click Add A Service:



You'll be presented with a catalog of services available to you. Scroll down to the Web and Application section and click MQ Light:



On the next screen, you can configure the service before you create it. On the right hand side, you can see the Space (leave it as your default space), the App to bind it to (leave it unbound, as you don't have any apps yet) and the Service Name (enter MQLight-sampleservice) and click CREATE:



MQ Light
IBM

PUBLISH DATE
1/16/2015

TYPE
Service

LOCATION
US South

[VIEW DOCS](#)

Develop responsive, scalable applications with a fully-managed messaging provider in the cloud. Quickly integrate with application frameworks through easy-to-use APIs.

- **Easy to Use**
Connect applications simply and efficiently so they can off-load work, share data or push events with simple API for Java and JavaScript and zero administration.
- **Robust and Scalable**
Rely on MQ Light's data integrity and asynchronous delivery to ensure your distributed applications are loosely-coupled, robust and scalable.

Pick a plan

Monthly prices shown are for country or region: [United Kingdom](#)

Plan	Features	Price
✓ MQ Light Standard Plan	Free allowance of 10,000 messages per month	£3.02 GBP/Million digital messages

[This is the standard service plan for MQ Light charged in units of millions of messages per month.](#)

[TERMS](#)

Add Service

Space:

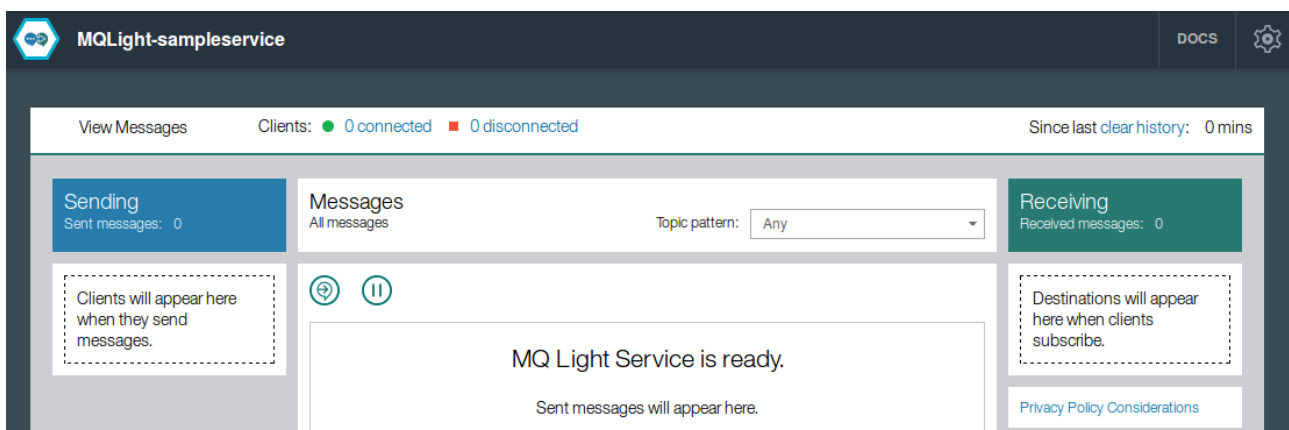
App:

Service name:

Selected Plan:

[CREATE](#)

An MQ Light instance will then be set up for you. After a short wait, you will be shown the same MQ Light UI you were using locally:



MQLight-sampleservice

DOCS

View Messages Clients: 0 connected 0 disconnected Since last clear history: 0 mins

Sending
Sent messages: 0

Messages
All messages Topic pattern: Any

Receiving
Received messages: 0

Clients will appear here when they send messages.

MQ Light Service is ready.
Sent messages will appear here.

Destinations will appear here when clients subscribe.

[Privacy Policy Considerations](#)

Click Back to Dashboard to return to the Dashboard for now. Now, let's look at getting your apps into Bluemix.

2. Configure the cf tool

We will be using the cf tool to push our applications into Bluemix, so let's configure that first. We've installed the tool for you, but you'll need to enter your Bluemix login details before you can use it. From a terminal window, run:

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

This will tell your cf tool where you will be deploying your apps (Bluemix):

```
demo@ubuntu: ~  
demo@ubuntu:~$ 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: 2.14.0)  
Not logged in. Use 'cf login' to log in.  
demo@ubuntu:~$
```

Next, you'll need to enter your login details. Run:

```
cf login
```

and enter your Bluemix login details when you are prompted:

```
demo@ubuntu: ~  
demo@ubuntu:~$ cf login  
API endpoint: https://api.ng.bluemix.net  
Email>
```

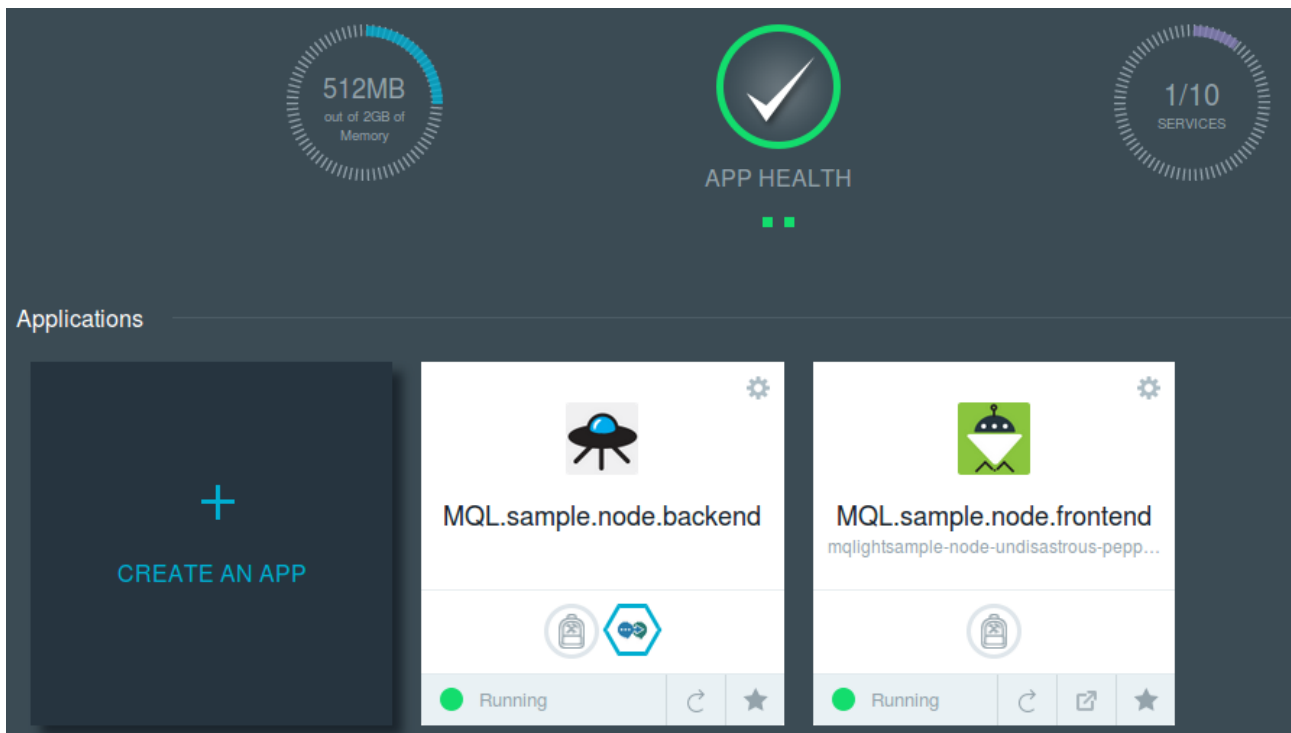
If you are prompted, select a n Org and a Space. You are now ready to push your apps into Bluemix!

3. Push your apps into Bluemix

This is the simple part. Open a new terminal window and navigate to /home/demo/mq1/lab/node. Here, run:

```
cf push
```

The cf tool will then read metadata from manifest.yml file in that directory to configure and run the app for you. After the command has finished running, check your Bluemix Dashboard to see 2 newly created apps:



We have 1 instance of the Node.js front-end and 2 Node.js worker instances, both connected to the same MQ Light service – all with one command! If you open the `/home/demo/mql/lab/node/manifest.yml` file in a text editor, you can see how we're configuring things:

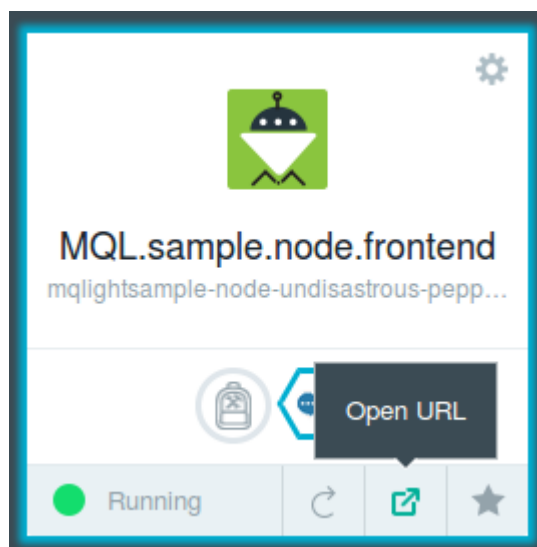
```
---
applications:
- name: MQL.sample.node.backend
  disk: 1024M
  command: node app.js
  path: worker_backend
  memory: 128M
  instances: 2
  no-route: true
  services:
  - MQLight-sampleservice
- name: MQL.sample.node.frontend
  disk: 1024M
  command: node app.js
  path: worker_frontend
  memory: 128M
  host: mqlightsample-node-${random-word}
  services:
  - MQLight-sampleservice
```

We can see 2 applications defined, one for the front-end and the other for the

back-end. The options here are read by the `cf push` command and used to configure the apps:

- `name` is the name of your application in Bluemix.
- `disk` is how much disk space your application have.
- `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.
- `path` is the path to the app on your local machine.
- `memory` is how much RAM you assign the app.
- `instances` indicates how many instances of the app Bluemix creates – this is only relevant to the back end worker.
- `no-route` indicates that the back end does not need a route (URL) created for it.
- `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.
- `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 earlier.

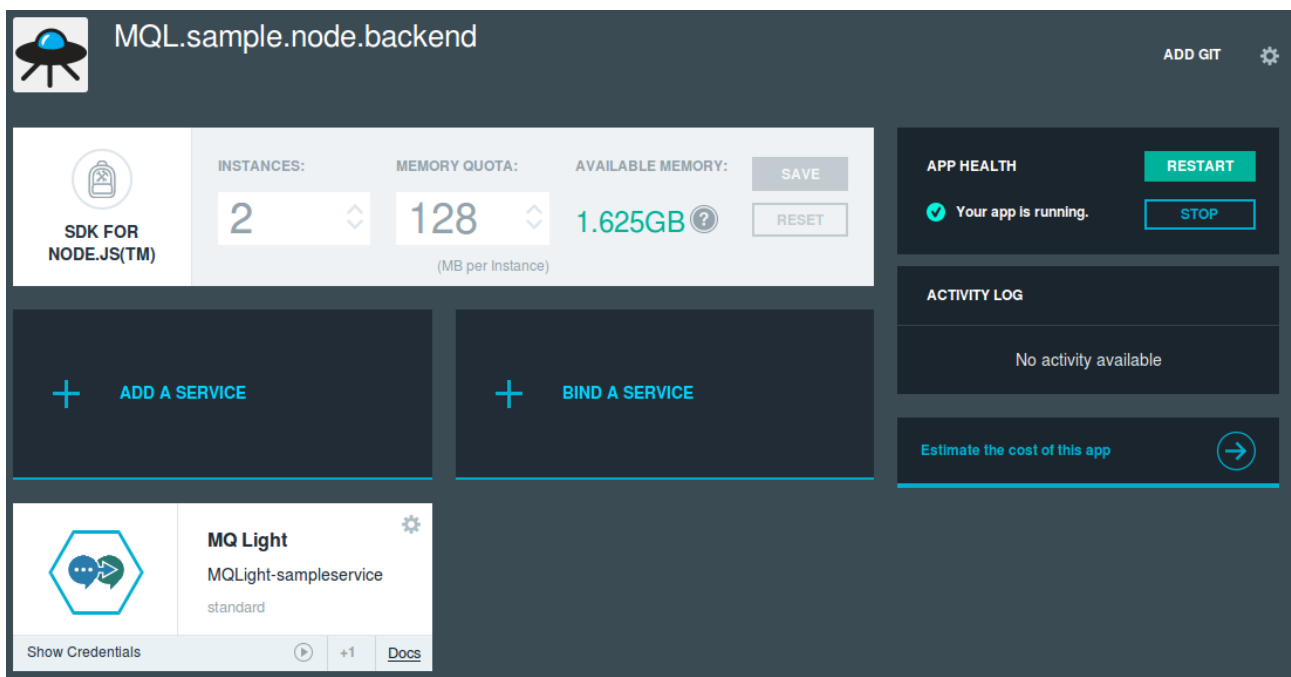
This allows us to push the apps with a single command. In the Bluemix dashboard, click the Open URL button on the front-end app:



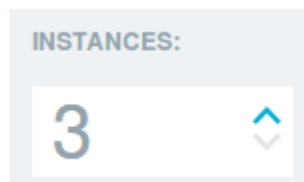
This will open the same web-app you were using locally. Click the Submit Work button to see it work in Bluemix.

4. Scale your apps in Bluemix

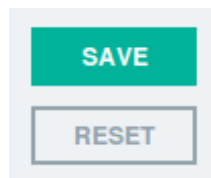
Right now we have 2 instances of the worker app running in Bluemix. But if we want more, scaling them in the cloud is even easier than locally. Returning to the dashboard, click the MQL.sample.node.backend app to be brought to its configuration page:



To start another instance of the worker app, perhaps to coter to demand, or increase performance, simply click the up arrow in the Instances box to increase the number of instances from 2 to 3:



Then click save, to the right:



A new instance will be started, with no further need for action. Scaling is as simple as a single button press!

5. Push Java and Ruby apps to Bluemix

If you choose to, you can also push equivalent Java and Ruby apps to Bluemix to run alongside the Node.js apps. For Ruby, open a terminal navigate to `/home/demo/mq1/lab/ruby` where you can run:

```
cf push
```

As before, configuration details are read from the `manifest.yml` file and used set up the apps. For the Java apps, run `cf push` in `/home/demo/mq1/lab/java`. When the apps are done deploying, head back to the original Node.js web app you had running in Bluemix. Click the Submit Work button to see that all the workers are processing the messages sent from the Node.js front-end app:

Notifications from the Node.js sample back-end look like this

Notifications from the Ruby sample back-end look like this

Notifications from the Liberty for Java sample back-end look like this

Sentence:

One, Two, Three, Four, Five, Once I Caught a Fish Alive

Submit Work

ONE, THREE, FOUR, TWO, FIVE, ONE, THREE, I FOUR, FIVE, ONCE
ONCE I A CAUGHT TWO, A FISH CAUGHT FISH ALIVE ALIVE ONE,
I TWO, ALIVE THREE, FOUR, FISH ONCE A FIVE, CAUGHT