Microservices with Seneca and MQ Light

AIS-M

Published on 06/05/2015 / *Updated on 07/05/2015*

update: MQ Light is no longer available as a cloud service

In this article I'm going to look at the Seneca microservices framework and how to use MQ Light as the transport for client/servers looking to consume/provide within this framework.

Microservices?

If you haven't already read Martin Fowlers article about microservices, then you really should. Fundamentally it's about building applications out of small, discrete services that consume each others capabilities through interfaces exposed over lightweight communication layers.

This allows these components to be managed and developed independently, using whichever technologies are appropriate, and typically makes it easier to scale the application to meet demand.

What is Seneca?

http://senecajs.org/ A microservices framework for building scalable applications in Node.js. Seneca provides features that make it simple to build microservices, and clients to use them, along with many plugins to extend the function of your microservice.

And MQ Light?

MQ Light is a AMQP server and cloud service, and a set of client libraries that provide a simple interface to robust messaging. The MQ Light server that you can download provides a developer friendly web interface to help build your messaging application, and when you want to take it to the cloud IBM Bluemix provides MQ Light as a service. Open protocols with IBM support.

I'm sold, now what?

By default Seneca provides HTTP and TCP plugins for the communications transport between microservices and clients, but you've decided you want to use a messaging transport because;

- It allows your application to be agnostic to the location of the microservice (all microservices are in the topic space of the AMQP server). Change the system that your microservice runs on, no problem, and no need to run a service discovery daemon.
- Using shared subscriptions you can easily scale up high use microservices without having to run a load balancer. Your application gets more popular and you need more backend processing, easy.
- you want to take advantage of message queueing to provide delivery guarantees. Your microservice is updated and cycled, messages collect on the queue while it's offline rather than timing out or being lost.

So to help with this I've developed a transport plugin for seneca that utilises MQ Light allowing you to take your Seneca app and easily switch the transport, instant win! 🙂

The code is available in github or downloadable from npm `npm install seneca-mqlight-transport` although typically you'll want to add it to the dependencies section of the `package.json` for your app.

```
{
  "name" : "seneca-mqlight-sample",
  "description" : "Simple example using seneca-mqlight-transport",
  "version" : "1.0.0",
  "dependencies" : {
   "seneca" : ">=0.6.1",
   "seneca-mqlight-transport": ">=0.0.6"
  },
  "engines": {
   "node": ">=0.10.0"
  }
}
```

Got it, can I see an example?

Sure. Let’s start with the sample given on the Seneca homepage that generates and returns ids.

```
require('seneca')()
    .add( { generate:'id'},
          function( message, done ) {
              done( null, {id:''+Math.random()} )
          })
    .listen()
```

That’s the original micro service just using the standard Seneca HTTP transport, and here’s the one that uses MQ Light.

```
var seneca = require('seneca')()
    .use('mqlight-transport', {})
    .add( {service: 'generate_id'},
          function(message, done) {
              done(null, {id: ''+Math.random()})
          })
    .listen({type:'mqlight'})
```

As we can see there are only a couple of differences;

- the `.use('mqlight-transport', {})` line which is telling seneca to load the `seneca-mqlight-transport` plugin
- in the first parameter to `.add()`, which is where the pattern for the service is defined, there is a field called `service`, this is a required field when using this transport as it is used as the topic the microservice will listen on.
- we pass a paramter into `.listen()` telling seneca that this microservice uses the `seneca-mqlight-transport` plugin

And from the client side similar changes are made, here’s the original example client;

```
require('seneca')()
```

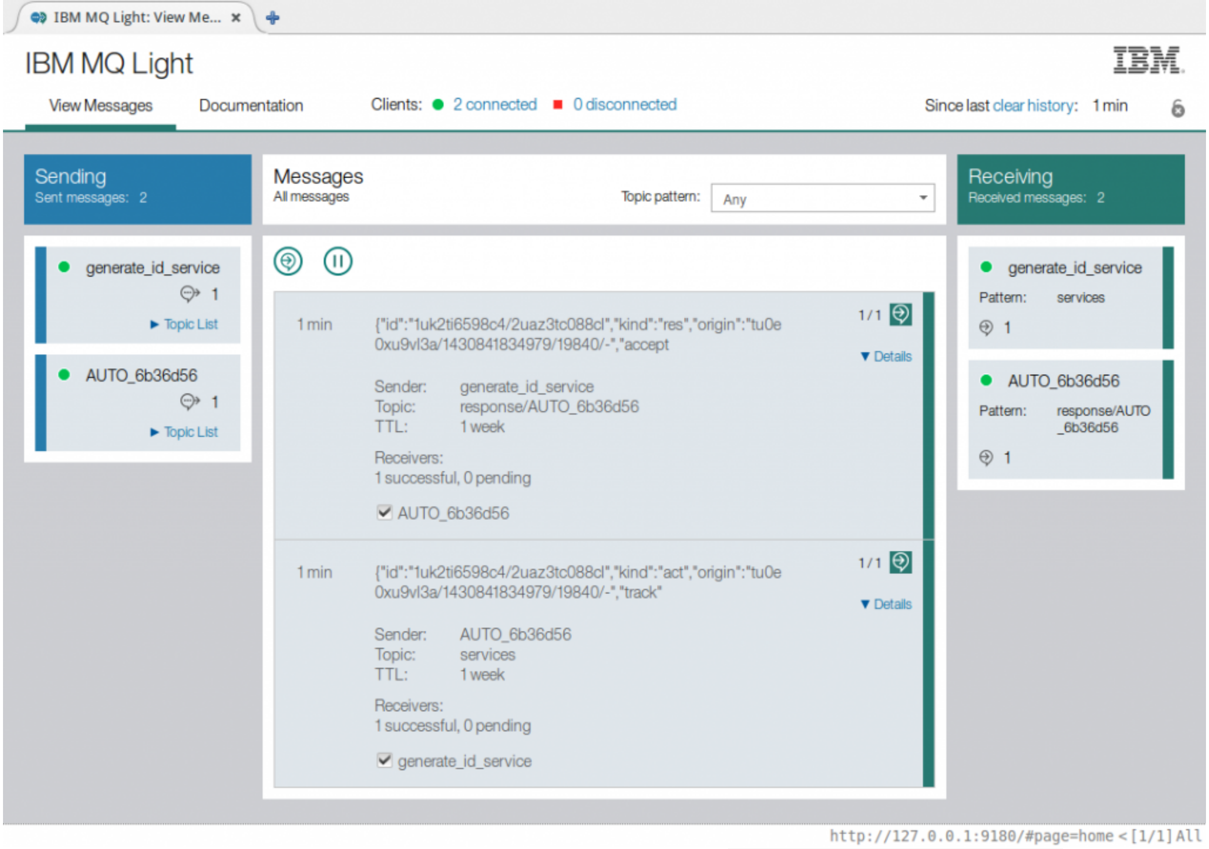
```
    .client()
    .act( { generate:'id' },
          function( err, result ) {
              console.log(JSON.stringify(result))
          })
```

and the same client code using MQ Light;

```

var seneca = require('seneca')()
    .use('mqlight-transport')
    .client({type: 'mqlight'})
    .act( { service:'generate_id' },
          function(err, result) {
              console.log(JSON.stringify(result))
          })
```

I mentioned earlier that MQ Light has a developer friendly web interface (by default running on http://127.0.0.1, so let's see what it looks like when running the microservice and client examples.



1 min {"id":"1uk2ti6598c4/2uaz3tc088cl","kind":"res","origin":"tu0e0xu9vl3a/1430841834979/19840/-","accept

Sender: generate_id_service
Topic: response/AUTO_6b36d56
TTL: 1 week

Receivers:
1 successful, 0 pending

☑ AUTO_6b36d56

1 min {"id":"1uk2ti6598c4/2uaz3tc088cl","kind":"act","origin":"tu0e0xu9vl3a/1430841834979/19840/-","track"

Sender: AUTO_6b36d56
Topic: services
TTL: 1 week

Receivers:
1 successful, 0 pending

☑ generate_id_service

So two clients connected (I gave the microservice the id `generate_id_service` and the client gets an auto generated id), they've both sent and received one message and we can see some details for those messages in the centre such as the topics used and that they were successfully received.

We need to go deeper

So that's a basic example of how to use the `seneca-mqlight-transport` plugin with Seneca, but let's look at something a bit more involved and the scalability I talked about. https://github.com/ibm-messaging/mqlight-fishalive-node is one of the MQ Light samples that demonstrates the "worker offload" pattern. The frontend submits a series of messages of work which are processed by a listener and the results are returned as new messages, in this case it's a series of words which are converted from lowercase to uppercase. Sounds like an ideal use case.

If you've downloaded or `npm install`ed the `seneca-mqlight-transport` you'll find there is a `samples/fishalive` directory and inside that you'll find `frontend` and `backend` directories. Run `npm install` in each to install the dependencies.

From the front…

Let's take a closer look at what's happening in the front end. The first major change is where we load Seneca and the `seneca-mqlight-transport` plugin and initialise it.

```
var seneca = require('seneca')()
    .use('mqlight-transport', {
        user: opts.user,
        password: opts.password,
        service: opts.service
    })
    .client({type: 'mqlight'})
    .ready(function() {
        console.log("Client ready")
        mqlightSubInitialised = true
    })
```

This shows how we can pass configuration options to initialise the AMQP client. Here we're passing in a username, a password and the MQ Light service address, all of which were picked up by the code below which will extract the relevant details from the application's environment when it is running in Bluemix (assuming an MQ Light service has been bound to the app).

```
var opts = {};
var mqlightService = {};
if (process.env.VCAP_SERVICES) {
    var services = JSON.parse(process.env.VCAP_SERVICES);
    console.log( 'Running BlueMix');
    if (services[ mqlightServiceName ] == null) {
        throw 'Error - Check that app is bound to service';
    }
    mqlightService = services[mqlightServiceName][0];
    opts.service = mqlightService.credentials.connectionLookupURI;
    opts.user = mqlightService.credentials.username;
    opts.password = mqlightService.credentials.password;
}
```

There are actually 4 fields that can be passed into the client intialisation;

- user – The username that will be used to connect to the service.
- password – the password associated with a username.
- service – The full URL of where to find the AMQP server to connect to, if this field is not populated a default of `amqp://localhost:5672` is used.
- id – A unique identifier for this connection, in the simple example earlier this is the field I set to have `generate_id_service` show as the identifier for the id microservice. If no id is specified the MQ Light client library will generate one of its own. NB: This identifier must be unique across all clients connected to a server, if you want to use shared subscriptions to scale a microservice a naming scheme for each instance such as `microservice-` might be useful.

Going back to the initialisation code, once the service reports that it is ready it logs this to the console and sets a global variable indicating this.

Unlike the original frontend sample we no longer have to manually subscribe to a topic on which to receive our responses, the `senenca-mqlight-transport` handles that automatically.

Now let’s compare the two sections that send the requests to the backend for processing, first the original code;

```
req.body.words.split(" ").forEach(function(word) {
        // Send it as a message
        var msgData = {
            "word" : word,
            "frontend" : "Node.js: " + mqlightClient.id
        };
        console.log("Sending message: " + JSON.stringify(msgData));
        mqlightClient.send(PUBLISH_TOPIC, msgData);
        msgCount++;
    });
```

And now the version that uses Seneca;

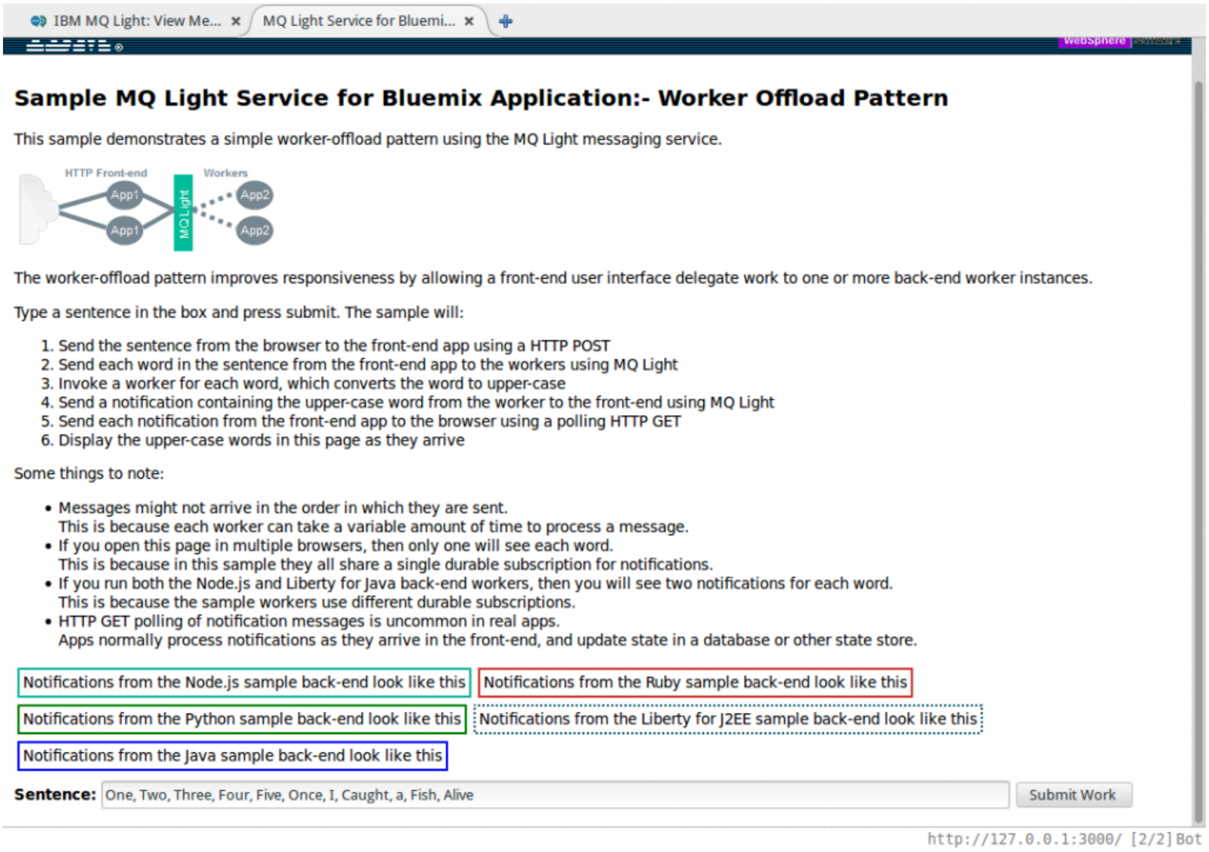
```
req.body.words.split(" ").forEach(function(word) {
        // Send it as a message
        var msgData = {
            "service" : "uppercase",
            "word" : word,
            "frontend" : "Node.js:"
        };
        console.log("Sending message: " + JSON.stringify(msgData));
        seneca.act(msgData, function(err, result) {
            console.log(result)
            heldMsg.push({"data" : result});
        })
        msgCount++;
```

```
    });
```

Pretty similar, the key differences being that we specify in the `msgData` object the service we want to send the message to along with the data fields the service expects. And rather than calling `mqlightClient.send()` we call `seneca.act()`. The two parameters passed are the data for the microservice and a callback to be executed when we get a response, in this case as the function to handle responses is so simple I have just inlined it.

The rest of the code for the app is involved with handling the express web front end and displaying the interactions and responses. With these changes the Seneca version is about 20 lines shorter than the original.

If you run `node app.js` and go to http://127.0.0.1:3000 in your browser you should see this screen.



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 Ruby sample back-end look like this
Notifications from the Python sample back-end look like this
Notifications from the Liberty for J2EE sample back-end look like this
Notifications from the Java sample back-end look like this

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

…to the back…

Time to see what the microservice backend looks like. Similarly to the frontend the first major difference in the initialisation of Seneca.

```

var seneca = require('seneca')()
    .use('mqlight-transport', {
        user: opts.user,
        password: opts.password,
        service: opts.service
    })
    .add({service: 'uppercase'}, function(message, done) {
        done(null, processMessage(message))
    })
    .listen({type:'mqlight', share_id: SHARE_ID, credit: 5})
```

In this case you'll see there are a couple of extra options on the call to `.listen()`, specifying a `share_id` and a `credit` value. The is used to form a shared subscription, when more than one client starts listening on a service and has been initialised with the same `share_id` only one client will be sent the request. `credit` is the maximum number of unconfirmed messages a single service can have outstanding.

Unlike the original sample which has to set up a subscription for receiving requests on, and set the message handler function.

```
mqlightClient.on('message', processMessage);
mqlightClient.subscribe(SUBSCRIBE_TOPIC, SHARE_ID,
        {credit : 5,
         autoConfirm : true,
         qos : 0}, function(err) {
             if (err) console.err("Failed to subscribe: " + err);
             else {
                 console.log("Subscribed");
                 mqlightSubInitialised = true;
             }
         });
```

in the Seneca version this is all done when we call `.act()`. We set the service name as `uppercase`, the second parameter is the function that is invoked when a message is received, this function is passed `message` which is the object representing the request, and a function `done`, this function handles returning the response to the requestor and should be invoked with any error (in this call explicitly `null`) and the result of processing, here being the output of the `processMessage` function.

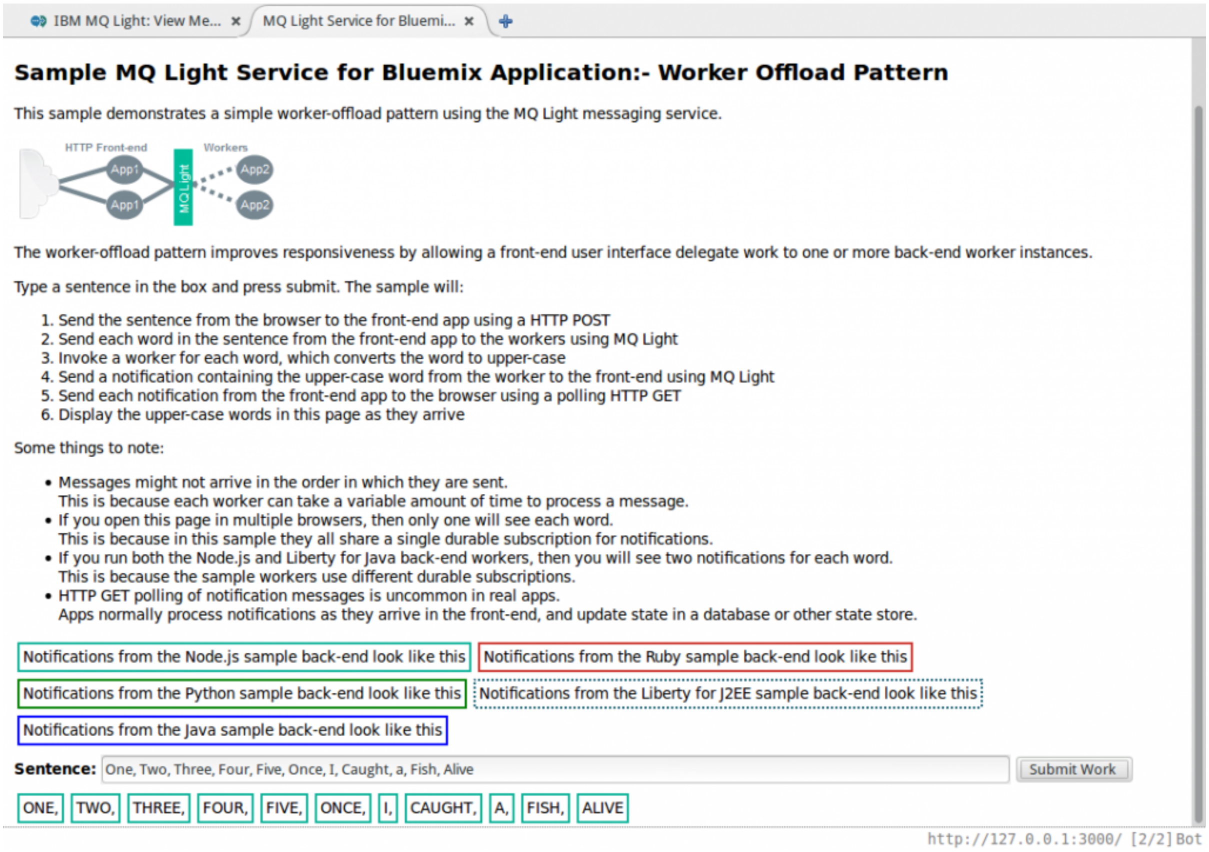
The `processMessage` function is functionally identical to the original, only returning the response to be sent rather than explicitly sending the response itself.

Similar to the frontend changes we've cut the size by of the code by 24 lines, nearly 25%.

If you run `node app.js` we'll have both parts of our sample running.

…that's where I was at

So with both pieces going, click the `Submit Work` button on the frontend webpage, you should see output that looks like this;



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:

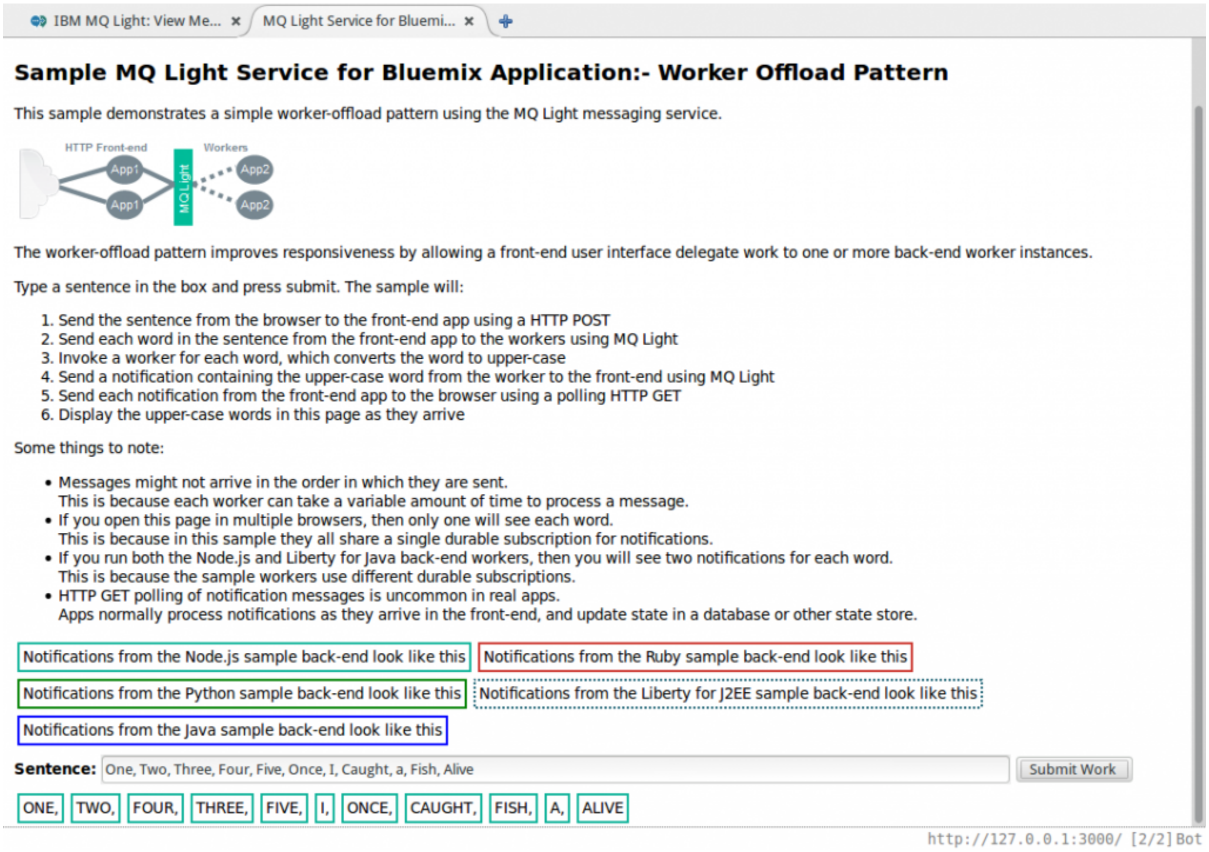
- Messages might not arrive in the order in which they are sent.
  This is because each worker can take a variable amount of time to process a message.
- If you open this page in multiple browsers, then only one will see each word.
  This is because in this sample they all share a single durable subscription 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 Ruby sample back-end look like this
Notifications from the Python sample back-end look like this
Notifications from the Liberty for J2EE sample back-end look like this
Notifications from the Java sample back-end look like this

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

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

And if you look at the console output of the frontend and backend programs you should see the activity of them sending, processing and receiving the requests and responses.
That scalability I promised? Easy. All we have to do is run another instance of the backend, the MQ Light service will handle distributing the requests to all the available instances of the microservice. Something we do need to be aware of though is that this is all asynchronous, your responses will not necessarily arrive in the order that the requests were sent. You can see this in this image showing the output from pressing `Submit Work` when I have two backend processes running.



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 Ruby sample back-end look like this

Notifications from the Python sample back-end look like this

Notifications from the Liberty for J2EE sample back-end look like this

Notifications from the Java sample back-end look like this

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

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

If the order of your work is important but you still want multiple instances of the microservice consider embedding timestamps and source identifiers into your requests.

Fin

Hopefully this article has given you a good overview of how to make use of the Seneca microservices framework using AMQP messaging with MQ Light as the transport for your requests and responses. Simple, scalable microservices that exploit the power of messaging. You’re enthused and want to get started, you can download the developer edition of MQ Light from the homepage. Don’t want to install it? It’s available as a docker image. When you’re ready to move your app into the cloud, IBM Bluemix has a free trial and offers MQ Light as a service, including the fantastic developer focussed console.