Flask style microservices with AMQP using MQ Light

AIS-M
Published on 26/06/2015 / *Updated on 02/07/2015*

I did an article a few weeks ago about using the Seneca framework in NodeJS to do microservices with AMQP via MQ Light as the network transport (have a read here). In this article I'm going to look at doing something similar with Python.

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.

Flask style?

So what do I mean by flask style? Well, really it's about using decorators on functions, a feature that came to Python in 2003. But the way Flask uses them to do route handling is so elegant, and the fact that people are building HTTP based microservices with it, meant I wanted to do something that felt similar using a different transport.

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.

Your ideas are intriguing to me and I wish to subscribe to your newsletter.

Excellent! So maybe you've already done some microservices with Flask that use HTTP for the transport, here are 3 reasons you might want to use a messaging transport instead;

- 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.

As an example of how easy it is to do this I've written a small Python library that wraps and extends the MQ Light Python client. The code is currently available on github and is easily installed with the `setup.py` file included;

```
git clone https://github.com/ibm-messaging/macaque.git
cd macaque
python setup.py install
```

...A little help?

An example? Of course, let's use some of the flask microservice example code from here, and tweak it to use our macaque library.

```
import macaque

app = macaque.Server()

tasks = [
    {
        'id': 1,
        'title': u'Buy groceries',
        'description': u'Milk, Cheese, Pizza, Fruit, Tylenol',
        'done': False
    },
    {
        'id': 2,
        'title': u'Learn Python',
        'description': u'Need to find a good Python tutorial on the
web',
        'done': False
    }
]

@app.service('/todo/api/v1.0/tasks')
def get_tasks(request):
    return {'tasks': tasks}

if __name__ == '__main__':
    app.run(debug=True)
```

First thing to note is how little we had to change. Rather than import flask we import macaque, app is an instance of the macaque server rather than flask,
and `@app.route` becomes `@app.service`.
To see what the output is like though we need to write another small app, there is no curl equivalent we can use, but it's not a lot of work;

```
import macaque
import json

app = macaque.Client()
```

```
def response_handler(response):
    print json.dumps(response, indent=4)
    exit(0)

app.call("/todo/api/v1.0/tasks", "", response_handler)
```

This simply initialises a macaque client, calls the service at `/todo/api/v1.0/tasks` with an empty message body, and when the response comes back prints the data. If we run both pieces the output from the client should look like this

```
{
    "tasks": [
        {
            "title": "Buy groceries",
            "done": false,
            "description": "Milk, Cheese, Pizza, Fruit, Tylenol",
            "id": 1
        },
        {
            "title": "Learn Python",
            "done": false,
            "description": "Need to find a good Python tutorial on the
web",
            "id": 2
        }
    ]
}
```

It's no surprise that it's exactly the same as the original HTTP service.
Let's continue with this sample, next we want to change this service to return a specific task based on an identifier that is sent to it. The macaque version of this is

```
import macaque

app = macaque.Server()

tasks = [
    {
        'id': 1,
        'title': u'Buy groceries',
        'description': u'Milk, Cheese, Pizza, Fruit, Tylenol',
        'done': False
    },
    {
        'id': 2,
        'title': u'Learn Python',
```

```
        'description': u'Need to find a good Python tutorial on the
web',
        'done': False
    }
]

@app.service('/todo/api/v1.0/tasks')
def get_tasks(request):
    task = [task for task in tasks if task['id'] == request['task_id']]
    return {'task': task}

if __name__ == '__main__':
    app.run(debug=True)
```

You're probably getting tired of reading this, but, look how similar it is to the original! 🙂 Rather than pass the value to be looked for in as part of the topic/URL we are passing a value in data for the request. Also I have skipped over the bounds checking as I haven't put a formalised way to return errors into the library (extensions like this are left as an exercise to the reader 😀) We just need to tweak the calling program slightly to pass in the value of the task we're looking for;

```
import macaque
import json

app = macaque.Client()

def response_handler(response):
    print json.dumps(response, indent=4)
    exit(0)

app.call("/todo/api/v1.0/tasks", {"task_id": 1}, response_handler)
```

And if we run this we get

```
{
    "task": [
        {
            "title": "Buy groceries",
            "done": false,
            "description": "Milk, Cheese, Pizza, Fruit, Tylenol",
            "id": 1
        }
    ]
}
```

I think that'll do for now, although it'd be a good way to explore this technology to continue to adapt the rest of this example app to use macaque instead of flask. The key point though is

that you could fairly easily use both libraries in the same app and serve your microservices over multiple transports. Abstract the core of your microservices from the transport handlers and it'd be a very slim layer you'd have to write to use both transports. So message, much jsons, wow.

And I'm not easily impressed — WOW, A BLUE CAR!

So that's a simple example, one service listener being called by a basic client. How about something a bit more involved that uses a couple more features? Like the Seneca article I did I have adapted a Python version of the fishalive demo to use macaque instead of just MQ Light messages. This demo 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.
If you've downloaded the macaque source code you'll find a directory called `samples/fishalive` and inside that you'll find `frontend` and `backend` directories.

Tramamampoline!

Lets start by looking at the backend code, these lines are the core of the backend application

```
app = macaque.Server(broker = service, cid = CLIENT_ID, security_options
= security_options)

@app.service(service_address, share_id = SHARE_ID)
def process_message(data):
    print data
    word = data['word']
    reply_data = {
            'word': word.upper(),
            'backend': 'Python: ' + CLIENT_ID
    }
    return reply_data

if __name__ == '__main__':
    app.run()
```

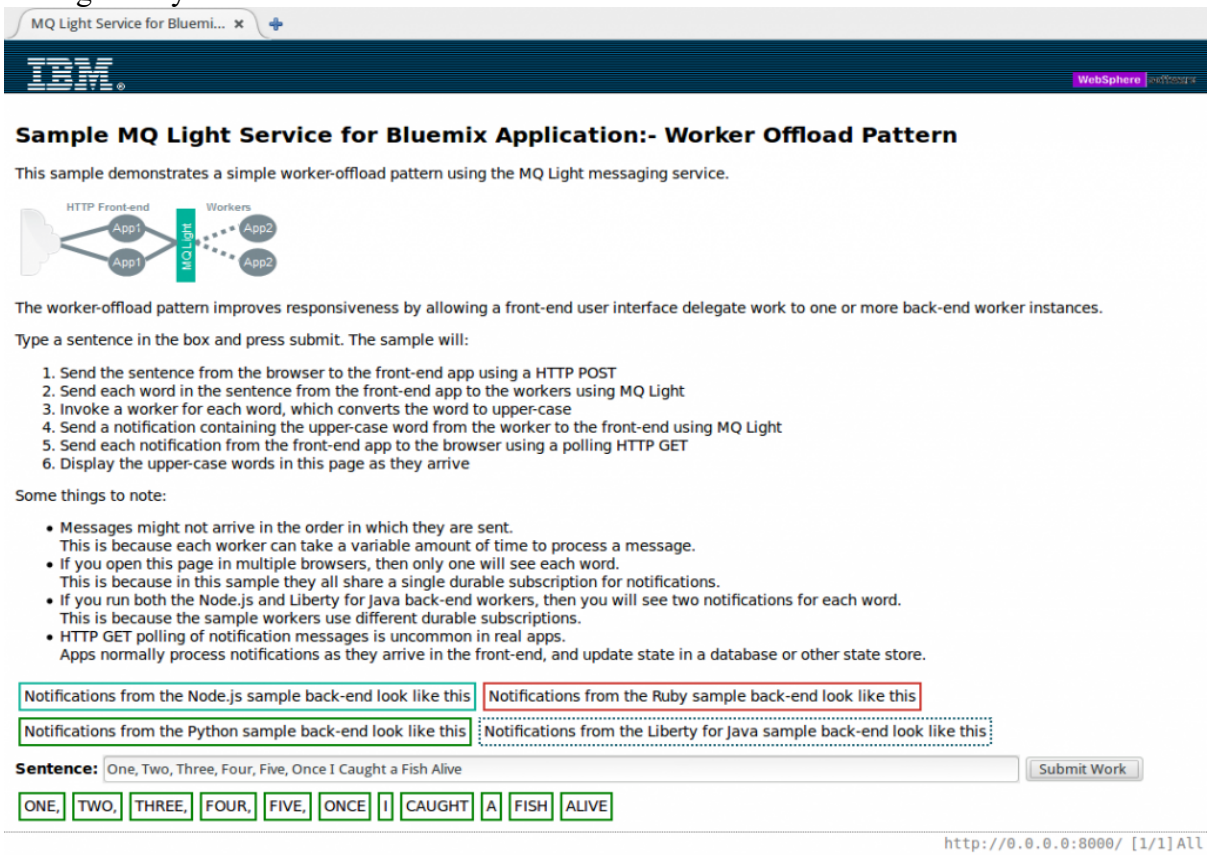
One important difference is that we no longer have a hard coded topic to send our uppercased words to. This is down to the way the macaque library works internally. When you start a client instance it subscribes to a topic of `response/` and whenever you do an `app.call()` it wraps the data being passed to the service with a structure that includes this topic as a reply to address, as well as embedding a UUID so we can map the response in the future to the correct callback. On the server side it automatically strips this wrapper off the message before passing your data to the microservice, taking the response from the microservice and sending it back to you.
Also to note is the `share_id` option on the `@app.service()` this specifies that the service endpoint will potentially be handled by multiple instances of the microservice. The MQ Light server will automatically distribute incoming messages to only one of the clients that are connected with the same share id (but all of the clients with unique or no share ids, if for example you wanted to run a logging service on an endpoint)

Ahoy ahoy?

So how about the frontend, how much different is that? I haven't included the code this time as there's not much new to see, rather it's all code that we've cut away. We don't need the same client setup, we don't need the `send_message()` function and the `process_message()` function is so much simpler. In total we've cut off 30 lines from the program.

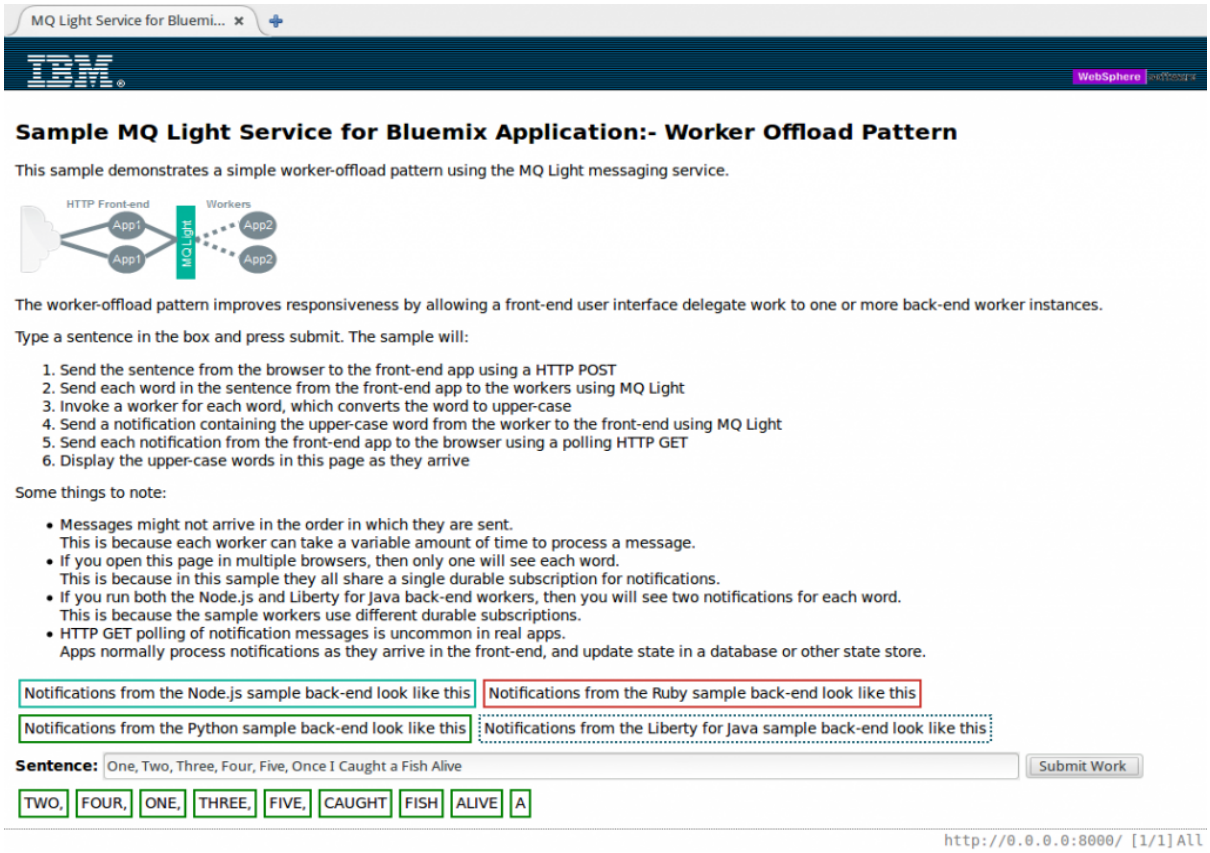
Lisa, I'd like to buy your rock.

So if we put it all together what does it look like? Start up the two programs and go to the URL given by the frontend and click "Submit work".



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.



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.

I really like the vest.

Hopefully this article has given you a good overview of how easy it can be to build microservices using AMQP messaging with MQ Light as the transport. 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.