Bluemix Hands-On Workshop

Section 6 - Services

Section 6 - Services

Version: 5
Last modification date: 19-Sep-14
Owner: IBM Ecosystem Development

Table of Contents

Bluemix Hands-On Workshop 1
Section 6 - Services 1
Exercise 6.a – Adding a service to an application 3
Exercise 6.b – Creating a user-provided service 8

Exercise 6.a – Adding a service to an application

Services allow you to quickly add capabilities to your application. In this exercise you will create 2 applications that communicate using a messaging service.

We will use Node.js as the programming language for this exercise and you can choose to use Eclipse, the command line or DevOps Services to complete this exercise.

The basic application for both the sender and receiver is shown below. The application should be in a file called app.js:

```
var express = require('express');
var appport = process.env.VCAP_APP_PORT || 3000;

//Setup web server
var app = express();
app.listen(appport);
```

This application uses the express framework, so you need to add the dependency to the package.json:

```
{
  "name": "<app name>",
  "version": "0.0.1",
  "description": "<app description>",
   "dependencies": {
     "express" : "4.9.3"
  },
  "scripts": {
       "start": "node app.js"
  }
}
```

A suggested manifest.yml file for the receiver applications is:

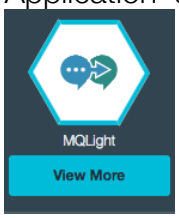
```
---
applications:
 - name: MQLrec
   host: MQLrec-${random-word}
   path: .
   instances: 1
   memory: 128M
```

For the sender application use the name MQLsnd and host MQLsnd-${random-word}

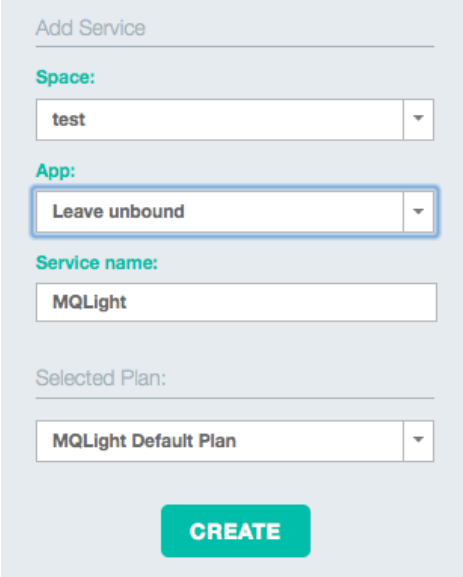
Create 2 folders or projects one called ‘sender’ and the other called ‘receiver’ and create the app.js, manifest.yml and package.json in each folder / project as shown above.

Deploy an instance of the MQLight service using either the Bluemix UI or the command line utility.

At the Bluemix Web UI you can deploy an instance of the MQLight from the ‘Web and Application’ section



When deploying the service use the name MQLight and leave the service unbound.



At the command line use the following command:

```
cf cs mqlight Default MQLight
```

To access the service from the applications we need to import a library to the node application (this information can be accessed from the service documentation).
Modify the package.json to require the additional package. Add the additional package to the dependencies section in both the sender and receiver applications:

```
    "dependencies": {
     "mqlight":"1.0.2014091000",
     "express" : "4.9.3"
  },
```

We can now add the package into the applications (again this needs to be done in both the sender and receiver app.js files). At the top of the application pull in the mqlight package and assign it to a variable to be able to access the features in the package:

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

To be able to connect to the MQLight service the environment variable VCAP_SERVICE needs to be parsed to obtain the connection information. The code below shows how to allow for both local deployment and Bluemix deployment :

```
if (process.env.VCAP_SERVICES) {
  //running in bluemix
  var credentials = JSON.parse(process.env.VCAP_SERVICES)['mqlight'][0].credentials;
  opts = {user:     credentials.username ,
          password: credentials.password,
          service:  credentials.connectionLookupURI};
} else {
  //running locally
  opts = {service:'amqp://localhost:5672'};
}
```

The code above needs to be added to both the sender and receiver application app.js.
The start of the applications should now contain:

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

var appport = process.env.VCAP_APP_PORT || 3000;
```

```
//get connection settings for MQLight
if (process.env.VCAP_SERVICES) {
  //running in bluemix
  var credentials = JSON.parse(process.env.VCAP_SERVICES)['mqlight'][0].credentials;
  opts = {user:     credentials.username ,
          password: credentials.password,
          service:  credentials.connectionLookupURI};
} else {
  //running locally
  opts = {service:'amqp://localhost:5672'};
}
```

To complete the exercise the sender and receiver applications will diverge from here. Starting with the sender application.

The sender application will publish a message each time a web request is received. To connect to the MQLight service and publish a message on starting the connection add the following code to the sender app.js above the **`//Setup web server`** comment:

```
// Connect client to broker
var topic = 'topic1';

var sendClient = mqlight.createClient(opts);

sendClient.on('started', function() {
    sendClient.send(topic, 'Hello World!');
});
sendClient.start();
```

The remaining step is to intercept web requests and send a message when a request is received. Express uses 'middleware' to chain together handlers. The following code inserts a new middleware to publish the request url. This code needs to be added inbetween the var app = express(); and app.listen(appport); lines :

```
//middleware to publish web requests received
app.use(function(req, res, next) {
  sendClient.send(topic, 'URL received = ' + req.url);
  next();
});
```

This completes the sender app. The entire application **`app.js`** should now contain the following – Note I've added a couple of basic handlers to return content to the requesting browser:

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

var appport = process.env.VCAP_APP_PORT || 3000;

//get connection settings for MQLight
if (process.env.VCAP_SERVICES) {
  //running in bluemix
  var credentials = JSON.parse(process.env.VCAP_SERVICES)['mqlight'][0].credentials;
  opts = {user: credentials.username,
          password: credentials.password,
          service:credentials.connectionLookupURI};
} else {
  //running locally
  opts = {service:'amqp://localhost:5672'};
}

// Connect client to broker
var topic = 'topic1';

var sendClient = mqlight.createClient(opts);

sendClient.on('started', function() {
    sendClient.send(topic, 'Hello World!');
});
sendClient.start();
```

```
//Setup web server
var app = express();

//middleware to publish web requests received
app.use(function(req, res, next) {
  sendClient.send(topic, 'URL received = ' + req.url);
  next();
});

// Add a handler for /
app.get('/', function(req, res, next) {
  res.send('Hello World');
});

// Add a handles for /help
app.get('/help', function(req, res, next) {
  res.send('No help here!');
});


app.listen(appport);
```

The receiver application will subscribe to the 'topic1' topic and log to the console any messages received. The code to be added to the receiver app.js file is:

```
// Connect client to broker
var topic = 'topic1';
var recvClient = mqlight.createClient(opts);
recvClient.on('started', function() {
      recvClient.subscribe(topic);
      recvClient.on('message', function(data, delivery) {
         console.log(data);
      });
});
recvClient.start();
```

To complete receiver app.js file should now contain:

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

var appport = process.env.VCAP_APP_PORT || 3000;

//get connection settings for MQLight
if (process.env.VCAP_SERVICES) {
  //running in bluemix
  var credentials = JSON.parse(process.env.VCAP_SERVICES)['mqlight'][0].credentials;
  opts = {user:     credentials.username ,
          password: credentials.password,
          service:  credentials.connectionLookupURI};
} else {
  //running locally
  opts = {service:'amqp://localhost:5672'};
}

// Connect client to broker
var topic = 'topic1';
var recvClient = mqlight.createClient(opts);
recvClient.on('started', function() {
      recvClient.subscribe(topic);
      recvClient.on('message', function(data, delivery) {
         console.log(data);
      });
});
recvClient.start();

//Setup web server
var app = express();
app.listen(appport);
```

The last task to complete is to ensure the MQLight service is bound to the application at deploy time. To achieve this the manifest.yml file for both the sender and receiver needs

to be modified to include the service. The complete manifest file for the receiver should now contain:

```
---
applications:
 - name: MQLrec
   host: MQLrec-${random-word}
   path: .
   instances: 1
   memory: 128M
services:
 - MQLight
```

and the sender manifest.yml file should contain:

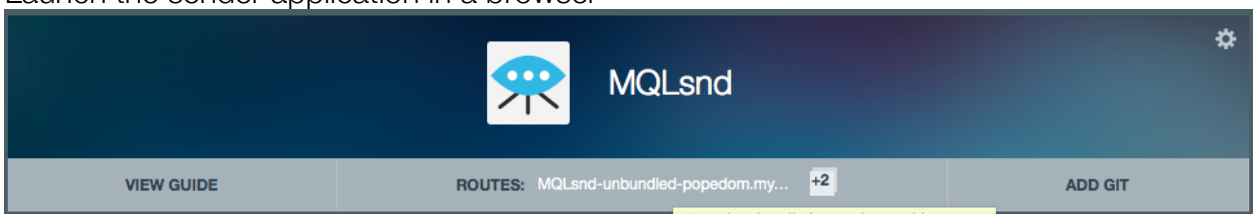
```
---
applications:
 - name: MQLsnd
   host: MQLsnd-${random-word}
   path: .
   instances: 1
   memory: 128M
services:
 - MQLight
```

You can now deploy both the receiver and the sender applications – you need 256MB memory, so ensure you have sufficient resources, if not delete applications from previous exercises.

To test the applications the command line interface can be used to tail the log of the receiver application. In a command line window enter the following:

```
cf logs MQLrec
```

Launch the sender application in a browser



In the console window you should see the message output:

```
cf logs MQLrec
Connected, tailing logs for app MQLrec in org binnes@uk.ibm.com / space test as
binnes@uk.ibm.com...

2014-09-19T14:57:09.33+0100 [App/0]    OUT URL received = /
```

You can also try adding different ending to the URL in the browser, such as /help – this should display 'no help here!' in the browser window and also output

```
OUT URL received = /help
```

to the console where the log is being tailed.

This concludes the exercise. You should now be familiar how to add a service to an application, how to parse the connection information for the service. The service documentation should explain how to use the service.

Exercise 6.b – Creating a user-provided service

Bluemix allows you to create a service local to your organization. The service needs to be running, not necessarily on Bluemix, but must be reachable from Bluemix. The service definition serves as a link between the running service and your Bluemix application.

To create a service use the Command Line Interface:

```
cf cups testService -p "host, port, user, password"
```

You will then be prompted for the values of the parameters. The values you enter will be stored with the service definition and passed to the applications binding to your service

```
cf cups testService -p "host, port, user, password"

host> myhost.ibm.com

port> 12345

user> user1

password> passw0rd
Creating user provided service testService in org binnes@uk.ibm.com / space test as
binnes@uk.ibm.com...
```

It is also possible to specify the values in the command rather than being prompted:

```
cf cups testService -p '{"host":"myhost.ibm.com", "port":"12345", "user":"user1",
"password":"passw0rd"}'
```

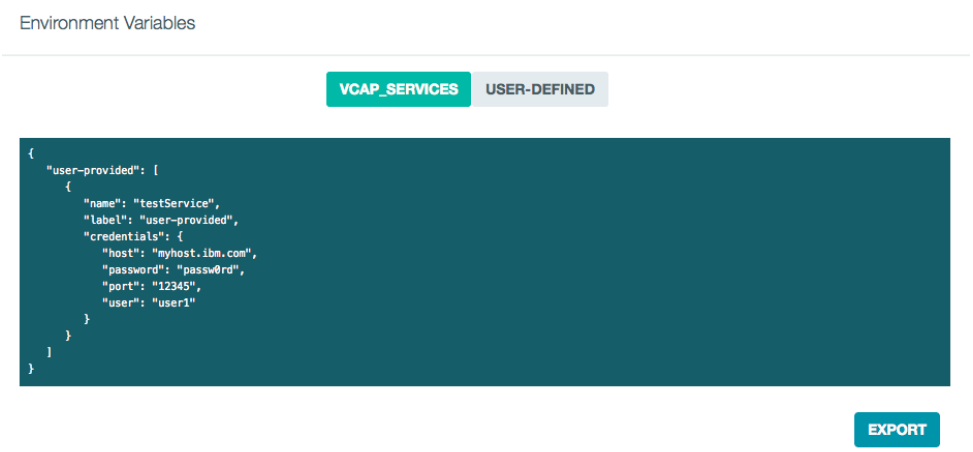
Create your own user-defined service and add parameters for the service using the –p option. You can provide the values on the command line or be prompted for the parameters.

You can list the services and see the service is now available using **`cf s`**:

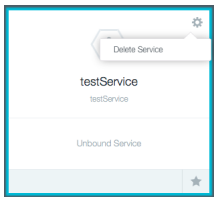
```
cf s
Getting services in org binnes@uk.ibm.com / space test as binnes@uk.ibm.com...
OK

name                                service           plan      bound apps
BIJavaCloudantTest:cloudantNoSQLDB  cloudantNoSQLDB   Shared    BIJavaCloudantDBApp
BINodeWebStarter:DataCache          DataCache         free      BINodeWebStarter
MQLight                             mqlight           Default
MQLight-yh                          mqlight           Default   BINodeWebStarter,
MQLrec, MQLsnd
testService                         user-provided
```

If you need to update the values for the service use **`cf uups`**.

You can bind the service to an application like any other Bluemix service and the application can parse VCAP_SERVICES to access the parameters you provided:



User-provided services can be deleted like any other Bluemix service using the Bluemix Web UI :



or using the command line **`cf ds`** :

```
cf ds testService

Really delete the service testService?> y
Deleting service testService in org binnes@uk.ibm.com / space test as
binnes@uk.ibm.com...
OK
```