**Title:**

From a Mashup to Python via Ajax, a bidirectional communication using a POST call.

**Summary:**

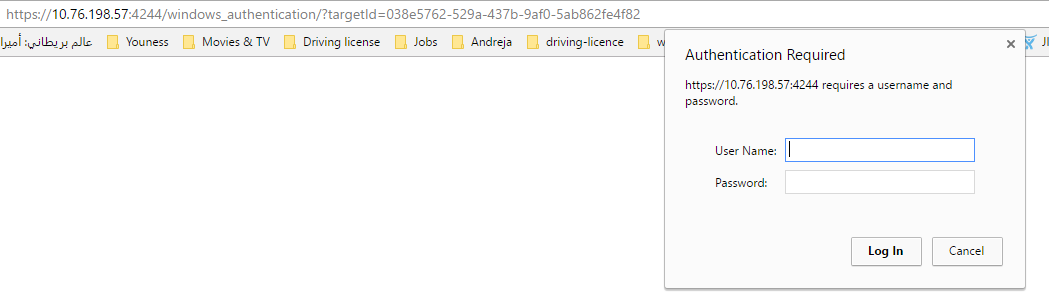
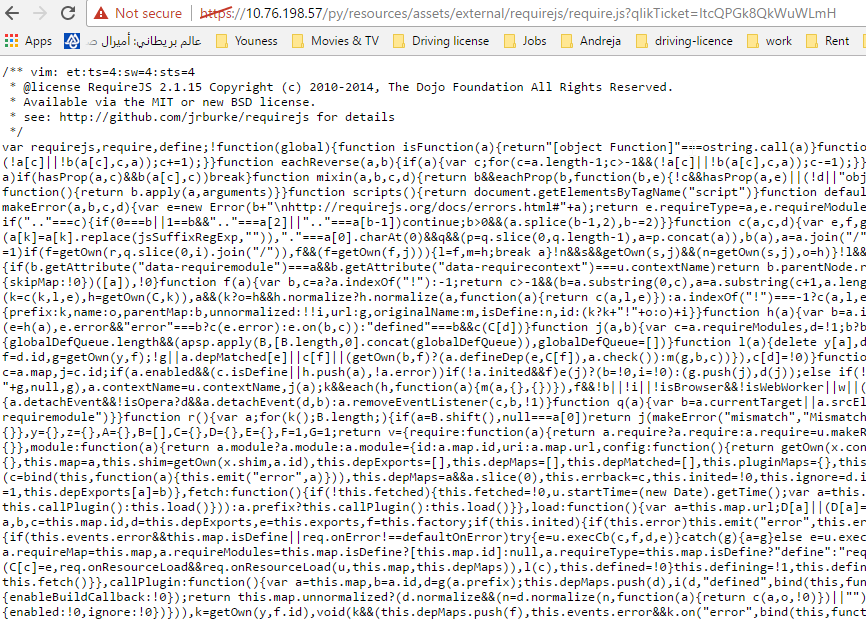
The article illustrates how the syntax and the server setup may look like for a mashup especially the jQuery part (using an Ajax call) when developers attempt to make a bidirectional communication between the mashup context and Python one. Bidirectional in this case means making a POST request which gets consumed by the Python application and then a response is sent back to the mashup (the Ajax part will intercept the response and gets handled in the “success” call back function as we will see in the code snippet).

**Description:**

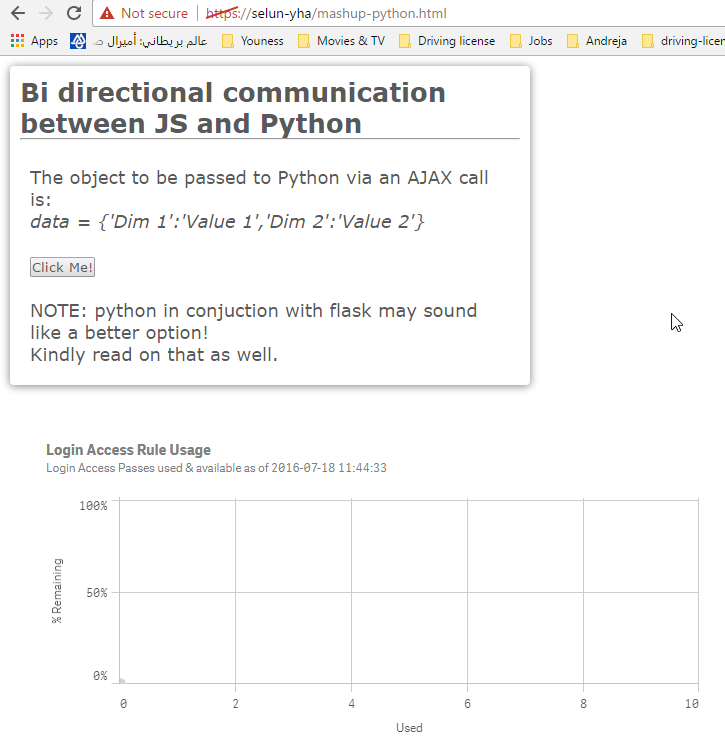
At times developers find themselves in need for working with different systems such as Python on Linux or Windows, DevOps and Continues Integration (CI) tools. Sometimes the CI test cases run in Python and perhaps the management wants to see KPIs or information about these test cases or it may as well be any reason for which Python deems to be necessary and data needs to be grabbed from there. So how do we make a mashup communicate with a Python application (a \*.py file on the server, local hard disk or on another server)? The details below attempt to answer this question and explain how this can be done using Qlik Sense Server, IIS, a Python file and a Qlik Sense Mashup. Basically, we will depend heavily on the Ajax call to make this happen. In addition once the communication is established we can extend the mashup using the capabilities API see <https://help.qlik.com/en-US/sense-developer/3.2/Content/APIs-and-SDKs.htm> for creating visuals on the fly using data that are not necessary part of the native Qlik Sense application but rather from the Python application.

**Resolution:**

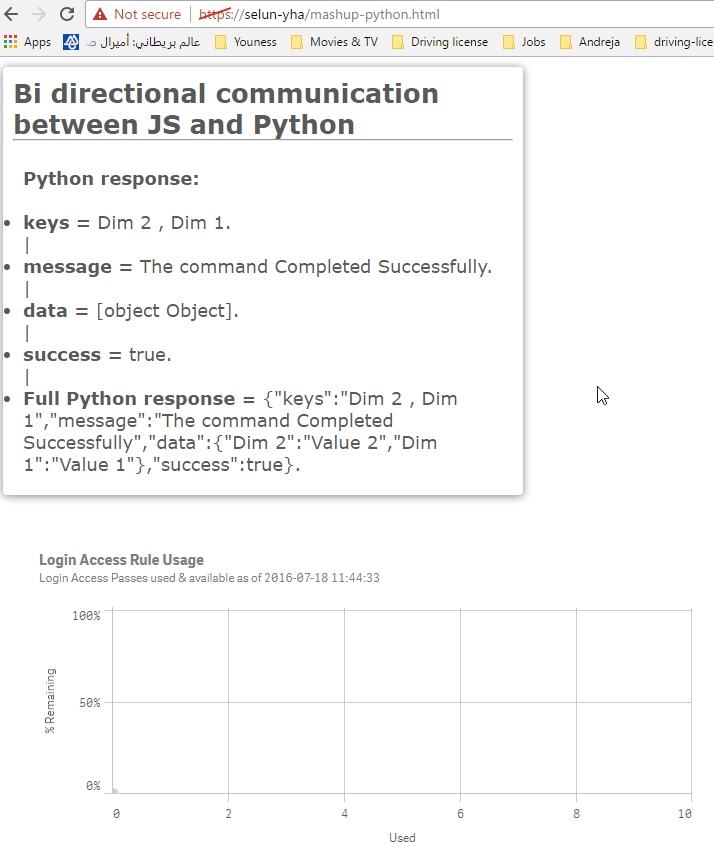
Before we proceed, the example in this article is a proof of concept and it can be implemented in different ways. Thus, the article is flexible and open for other suggestions. Also, there are a few pre-requisites that needs to be in place, these describe the setup I had in place for the bidirectional communication to work properly, the pre-requisites are:

1. A Windows 10 machine.
2. IIS installed and running on the Windows 10 machine, if you run into permission issues then run IIS as an administrator.
3. Python installed on the Windows 10 machine, the version tested with, is 2.7. This can be downloaded from <https://www.python.org/downloads/>
4. A Qlik Sense Server 3.2 installed on a separate machine.
5. In Qlik Management Console (QMC), create a new virtual proxy and assign it the following values:
   1. Description: Python (optional).
   2. Prefix: py, you can name your prefix whatever you wish.
   3. Session cookie header name: X-Qlik-Session-py. Can be named as you wish.
   4. Under **Authentication**, Anonymous access mode: No anonymous user.
   5. Authentication method: Ticket.
   6. Windows authentication pattern: Windows. You can choose other patterns depending on what suites you better.
   7. If you run into CORS origin issues, then under **Advanced >** Additional response headers: Access-Control-Allow-Origin: *xyz*. Where *xyz* is to be replaced by the domain name of your mashup site (the domain name). The syntax in QMC takes only the name stripped from its http or https part and the .com or .net etc. In my case it is, Access-Control-Allow-Origin: selun-yha.
   8. Don’t forget to add your website domain name in the host white list, i.e. selun-yha.
   9. For more information on how to setup virtual proxies please see <https://help.qlik.com/en-US/sense/3.2/Subsystems/ManagementConsole/Content/create-virtual-proxy.htm>
6. Then we need a mashup that has been created initially in Qlik Sense dev-hub and moved from the Qlik Sense server into your IIS. My folder structure for the mashup in IIS looks as follows:
   1. mashup-python.css, you can download this from Qlik Sense Server and use it as a local dependency.
   2. mashup-python.html, copied from dev-hub into this html file.
   3. mashup-python.js, copied from dev-hub into this js file).
   4. myStyle.css, added it myself for additional styling purposes.
   5. PythonApp.py, the Python file our mashup will communicate with.
   6. web.config, this will be autogenerated by IIS.
7. The mashup is running from my IIS as a website on the Windows 10 machine with support for **HTTPS** protocol, a secure communication with Qlik Sense serer is needed to be able to open the QVF application in the mashup, as in   
     
   //open apps, you may change it to refelect your own setup  
   var app = qlik.openApp('59201a52-2319-4f7a-a21b-0ae0307cd70e', config);  
     
   So, you need to bind your site to HTTPS. You can do this by selecting your site, then in the right menu under **Actions**, click on Bindings… then choose HTTPS and use port 443. You then need to select an SSL certificate. This would be your Qlik Sense Server certificate installed on your IIS using MMC.
8. Enable support for CGI/Python for the site hosting the mashup. You can do this upon adding a new site. Select the site, then double click on **Handler Mapping** under IIS settings section. Kindly keep in mind that the full steps of enabling CGI/Python are not in the scope of this article, thus for the complete details on this you need to refer to another article titled *How to enable Python on IIS* or you may search the Internet and/or Python forums.
9. In my case both the Widows 10 machine and the Qlik Sense 3.2 server are on the same local network.
10. For IIS to work properly I had to add the SELUN-YHA\IIS\_IUSRS in addition to the IUSR *group* or *user names* to the permissions of the site.
11. Remark, this document does not provide Single Sing On mechanism steps while working with the mashup - Python concept, thus for the sake of keeping things simple, once you launch the mashup from your IIS, try to login to your hub using another tab in the browser so that a proper session is created for you. Otherwise in the developer console in Chrome for example you may see an error message like this GET https://10.76.198.57/py/resources/assets/external/requirejs/require.js net::ERR\_INSECURE\_RESPONSE To solve this issue just copy the URL found in the error message to another tab, hit enter and then login as you do when login into a Qlik Sense Hub, if everything goes fine you will get a pop up like this.   
      
      
    *Qlik\_Sense\_Mashup Login.png*
12. Once you login, a ticket will be issued to you by Qlik Sense and the output in the browser will look like as the image below, this is simply the browser loading the RequireJS file:  
      
    *  
    requirejs\_require\_qlikTicke.png*

Once everything is in place, the mashup before making an Ajax request towards Python will look like this:

*  
Qlik\_Sense\_Mashup before.png*

Once “Click me” button is pressed and the Python response is received, the mashup will look like this:

*  
Qlik\_Sense\_Mashup After.png*

**Code wise, the crucial parts look like this:**

***HTML (necessary imports)***

<!-- Import server dependencies -->

<link rel="stylesheet" href="https://10.76.198.57/py/resources/autogenerated/qlik-styles.css">

<script src="https://10.76.198.57/py/resources/assets/external/requirejs/require.js"></script>

<!-- Import client dependencies -->

<link rel="stylesheet" href="mashup-python.css">

<link rel="stylesheet" href="myStyle.css">

<script src="mashup-python.js"></script>

***JavaScript (Ajax call)***

// Make an AJAX call towards Python and handle the response

function runTheTest(){

$.ajax({

type: "POST",

/\* headers: {"Access-Control-Allow-Origin:" : "https://selun-yha"}, NOTE, Access-Control-Allow-Origin might be needed if your Python file is sitting on another web site \*/

url: "PythonApp.py", // Pointer to the Python file that is supposed to consume the AJAX call

data: {'Dim 1':'Value 1','Dim 2':'Value 2'}, // Dummy data passed to Python, this could be pulled from your app instead

success: function (PythonResponse) {  
 displayResponse(PythonResponse);

},

error: function (xhr, status) {  
 console.log(xhr + " " + status); }

});

};

***Python (creating a response of Jason type in the header)***

sys.stdout.write("Content-Type: application/json")

sys.stdout.write("\n")

sys.stdout.write("\n")

result = {}

result['success'] = True

result['message'] = "The command Completed Successfully"

result['keys'] = " , ".join(fs.keys())

d = {}

for k in fs.keys():

d[k] = fs.getvalue(k)

result['data'] = d

sys.stdout.write(json.dumps(result, indent=1)) # Encoding JASON hierarchies

sys.stdout.write("\n")

sys.stdout.close()

For the entire project with source code used in this proof of concept, please see the attached zip file.

**Tags:**

Mashup, Python, Ajax, API, JSON, Integration