**Exercise 11b**

*Using OAuth introspection with a third-party identity server to replace client-certificates*

**Prior Knowledge**

Previous exercises

**Objectives**

Replace SSL client authentication with the use of an OAuth2 token

**Software Requirements**

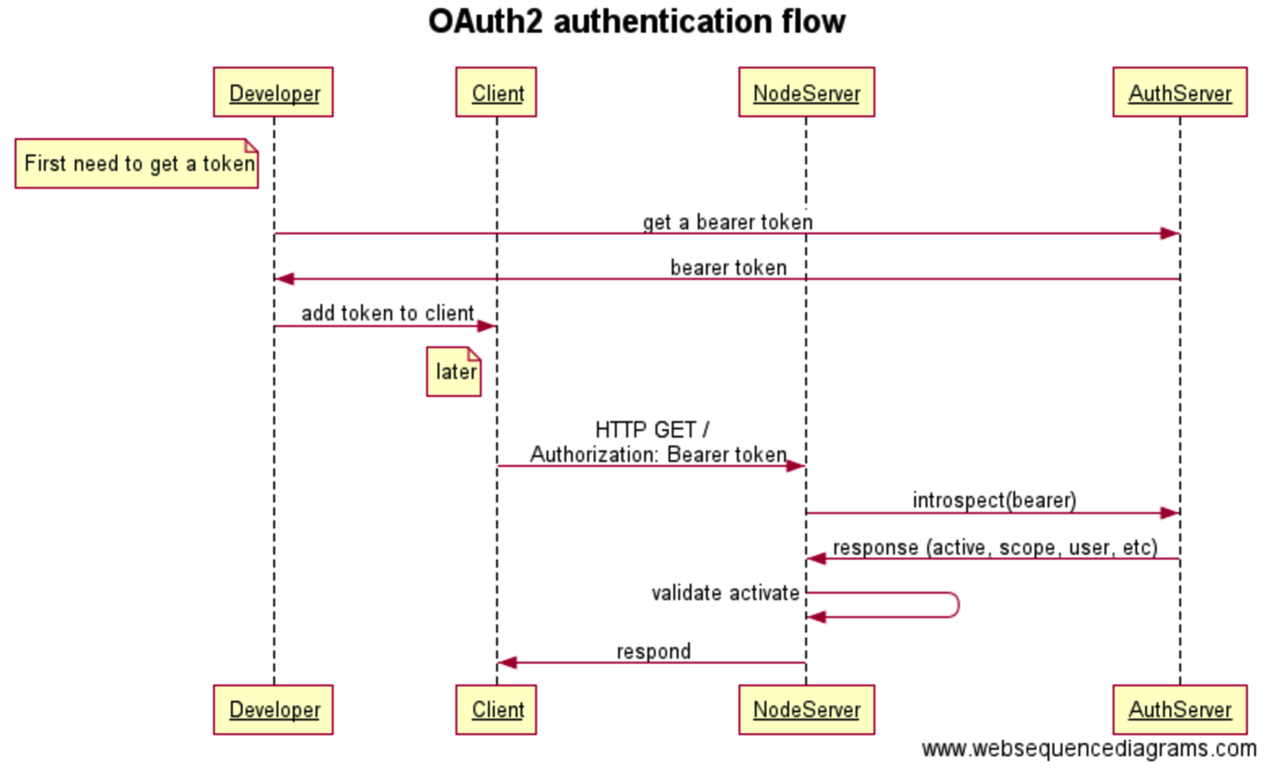
(see separate document for installation of these)

* Docker (and thereby WSO2 IS 5.1.0)
* Node.js
* Python

**Overview**

*In this lab, we will use server-side TLS to validate the server to the client, and use an OAuth2 token to validate the client to the server.  
  
The lab will follow the following overall approach:*

1. *Run a docker image of the WSO2 Identity Server to provide an OAuth2 server.*
2. *Create the OAuth2 client definition inside the Identity Server.*
3. *Issue the token and get approval to access a given scope using client credentials.*
4. *Enhance it to use the authorization grant.*
5. *Enhance the node.js server to validate the token and the scope using the OAuth2 Introspection API*

*Here is a picture of what we are looking at:  
*

**Steps**

1. I have created a docker image of the WSO2 Identity Server 5.1.0, and this also includes an add-in that supports OAuth2 Introspection. (Later versions include this natively).  
     
   docker run -d -p 9443:9443 -p 9763:9763 pizak/wso2is:5.1.0-introspect
2. I have also enhanced the code of the SSL exercise to support OAuth2 tokens. Copy this code to VM:

cd ~

wget [https://freo.me/sec\_oauth -O sec\_oauth.zip](https://freo.me/sec_oauth%20-O%20sec_oauth.zip)  
unzip sec\_oauth.zip

cd ~/sec\_oauth

1. Ensure you have the right Node dependencies for this:

cd ~/sec\_oauth/server  
npm install querystring express

1. Now look at the file server/server.js  
   The main changes are in bold.:  
     
   This is importing a new module called Introspect. Introspect is configured with a introspection endpoint (localhost:9763/introspect).   
     
   We extract the bearer token from the headers and if it exists we call introspect. Notice that since this is node.js, we need to preserve the non-blocking nature of the system, so we pass a callback to Introspect that will actually do our “GET” logic. This is because Introspect is going to call out to another HTTP endpoint and we don’t want to block node threads while it is waiting for the response.  
     
   If introspect succeeds, it passed a username and scope to the callback, otherwise null, null.

**var Introspect = require('./introspect.js');**

**introspect = new Introspect("localhost", 9763, "/introspect");**

app.get("/",function(req,res){

console.log(req.headers);

**auth = req.headers.authorization;**

**if (!auth) res.sendStatus(401);**

**bearer = introspect.getBearer(auth);**

**if (bearer) introspect.introspect(bearer, function (username, scope) {**

**if (!username) {**

**res.status(401).send();**

**}**

else {

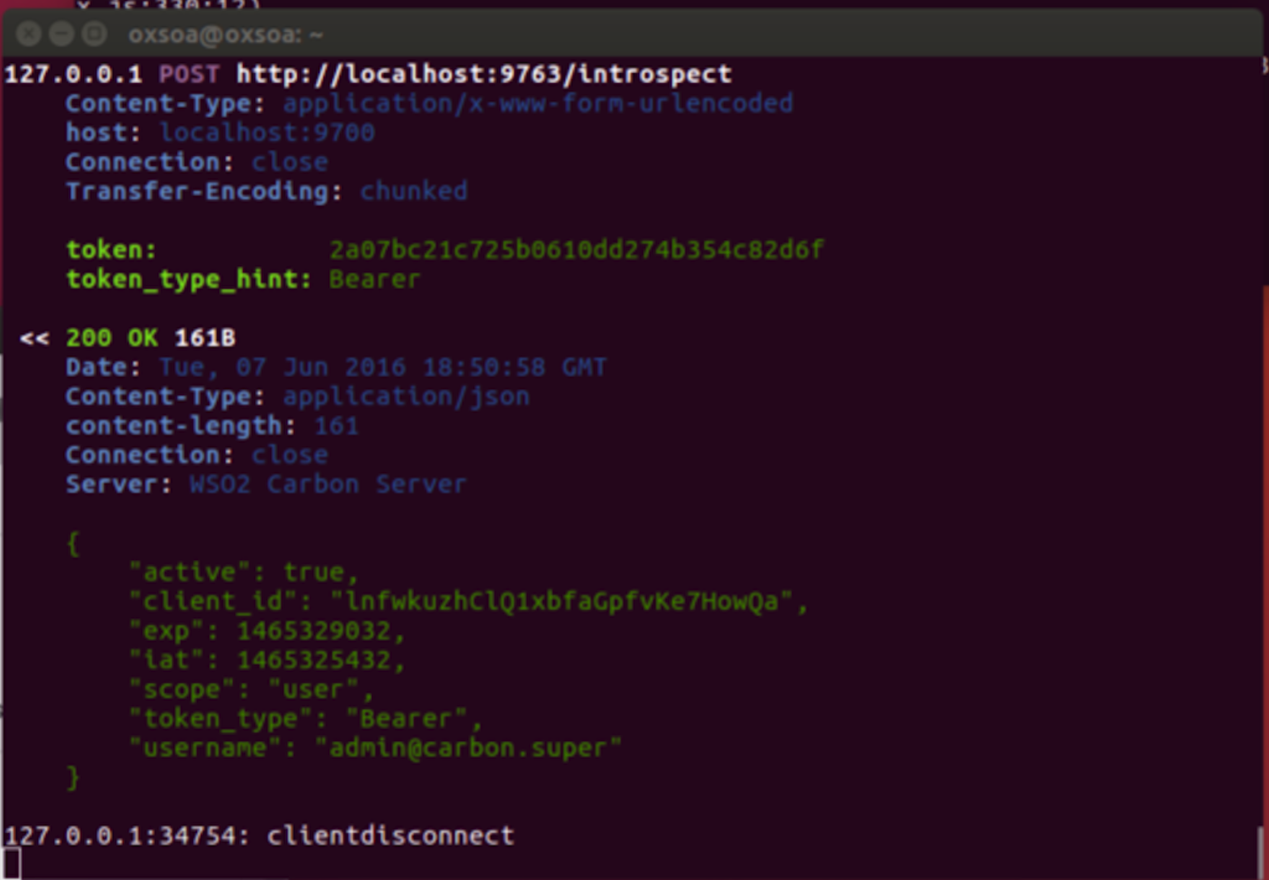
obj = {random : Math.floor((Math.random() \* 100) + 1),   
 **username:username,   
 scope:scope};**

res.json(obj);

}

});

});

1. What we need introspect to do is to call an introspection API defined in <https://tools.ietf.org/html/rfc7662>   
     
   Here is a sample mitmdump:  
   
2. Now lets look at introspect.js  
   The main logic (subset of the full file) we care about is here:

introspect : function (token, callback) {

console.log(token);

data = { token : token,

token\_type\_hint : "Bearer"

}

encoded = qs.stringify(data);

var post\_options = {

host: introspect\_host,

port: introspect\_port,

path: introspect\_path,

method: 'POST',

headers: {

'Content-Type': 'application/x-www-form-urlencoded'

}

};

var post\_req = http.request(post\_options, function(r) {

var body = ""

r.setEncoding('utf8');

r.on('data', function (chunk) {

body += chunk;

});

r.on('end', function() {

try {

var response = JSON.parse(body);

} catch (e) {}

if (response && response.active) {

callback(response.username, response.scope);

}

else

{

callback(null,null);

}

});

});

// post the data

post\_req.write(encoded);

post\_req.end();

}

};

1. The code is actually pretty simple, but the async nature of node.js slightly obfuscates things. Basically we start a post operation, and will get called asynchronously as chunks of response come in. This is slightly overkill as the response to a introspection call will probably always fit in a network buffer, but this is good node.js coding.
2. I have also modified random-client.py to look for a bearer token in the command-line:

bearer = ""

headers = dict()

url = "https://localhost:8443"

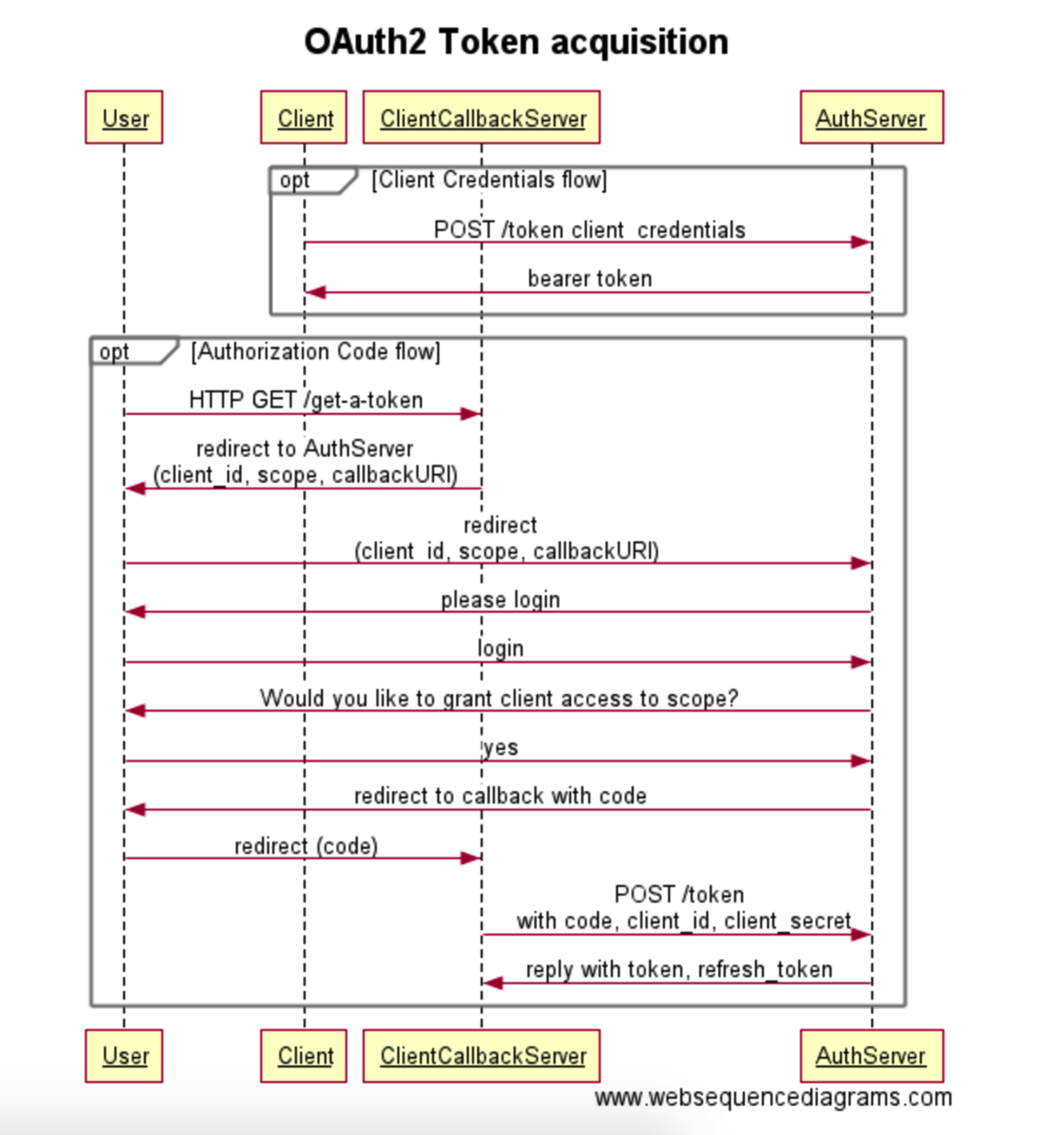
if (len(sys.argv) > 1):

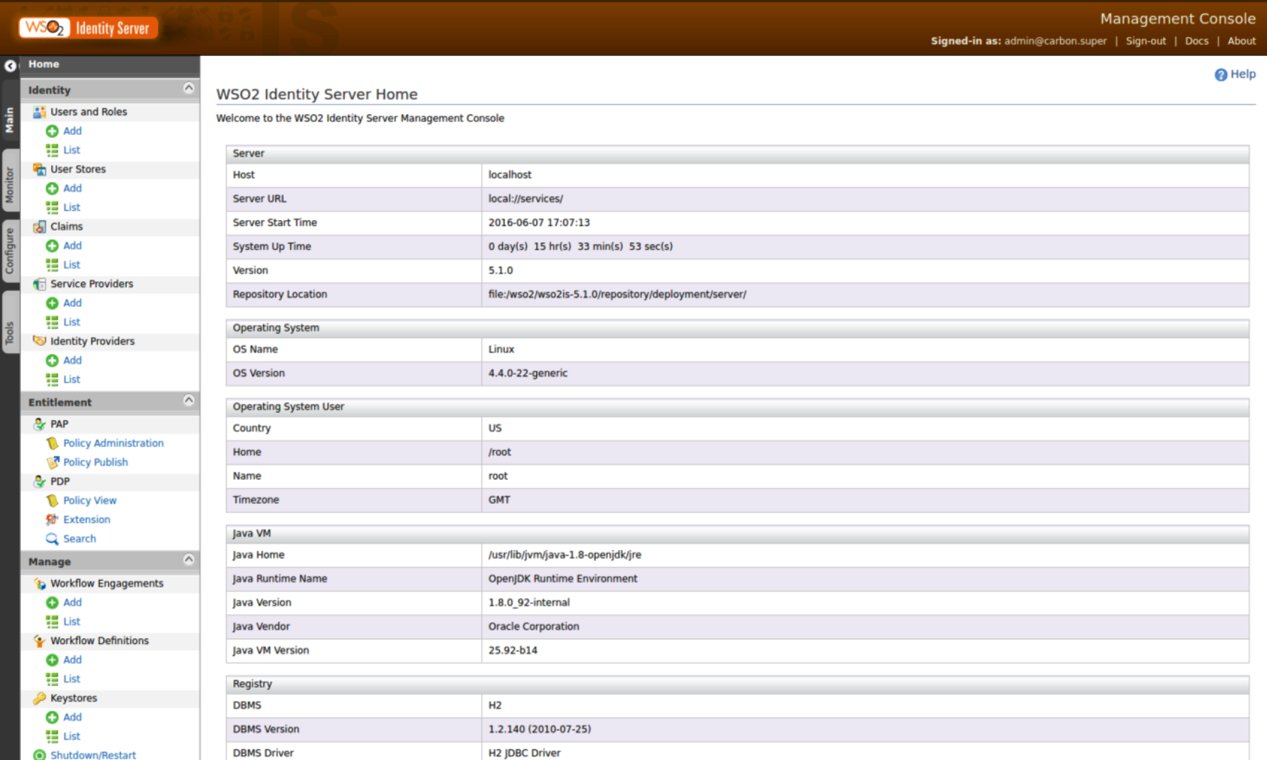
bearer = sys.argv[1]

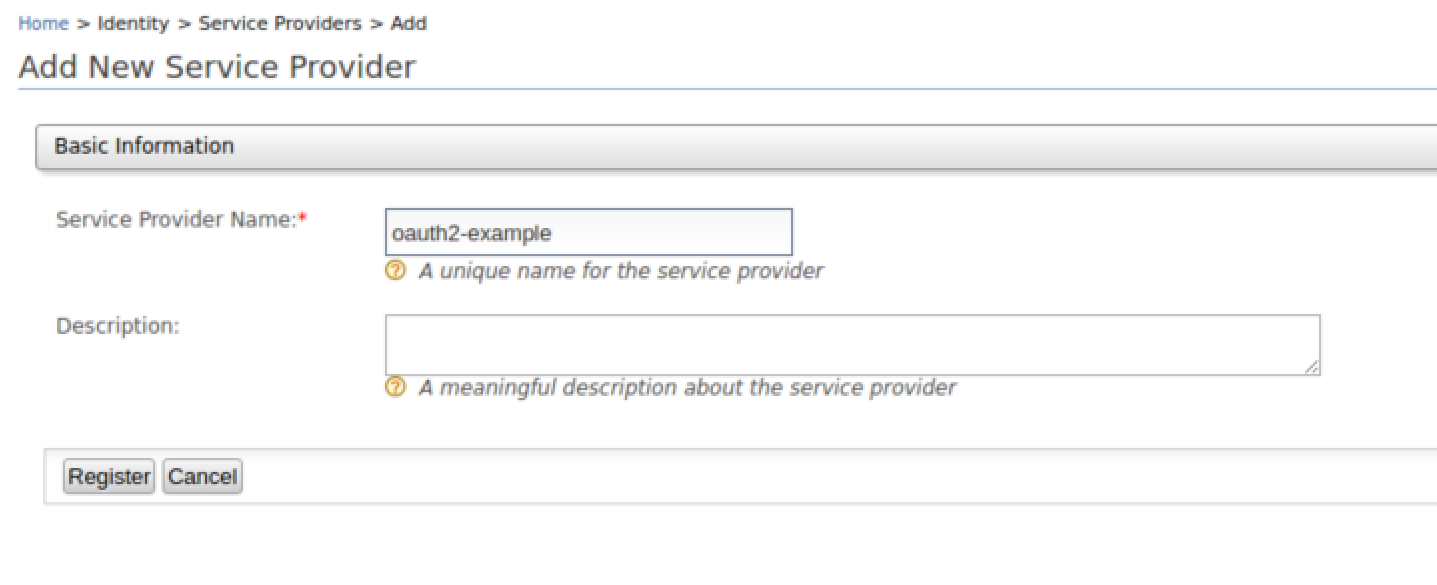
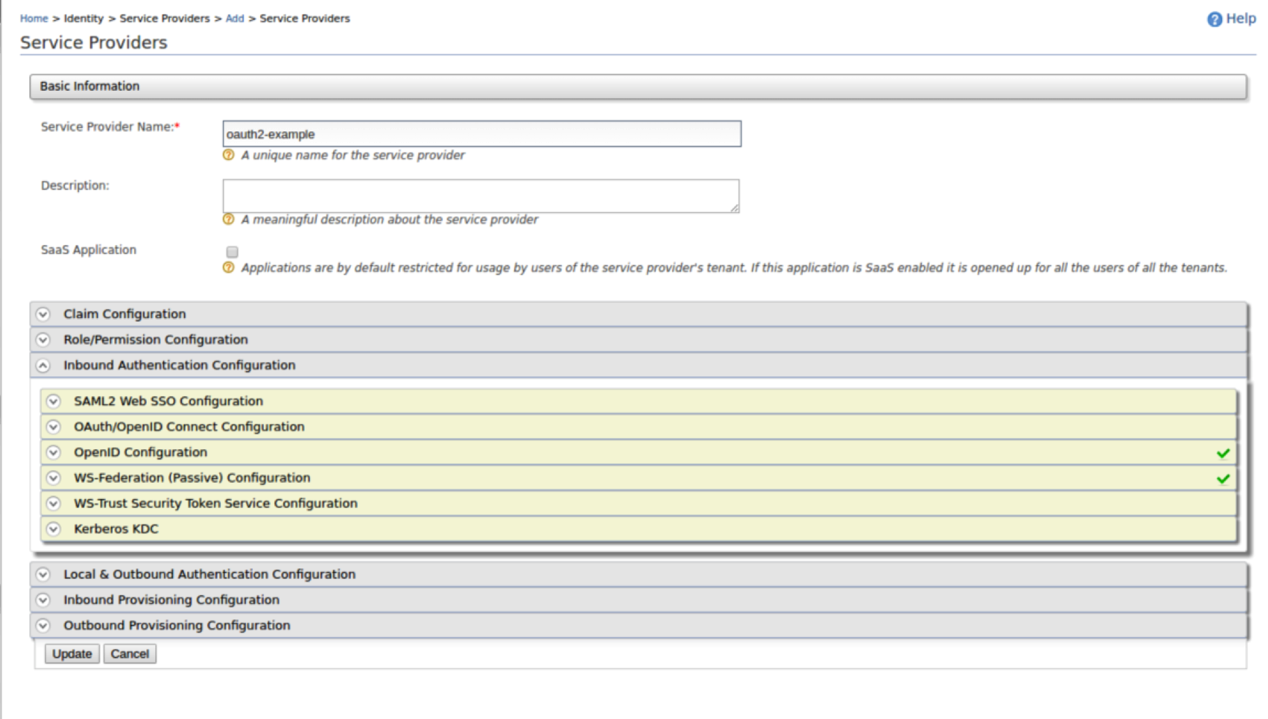
headers = dict(Authorization="Bearer "+bearer)

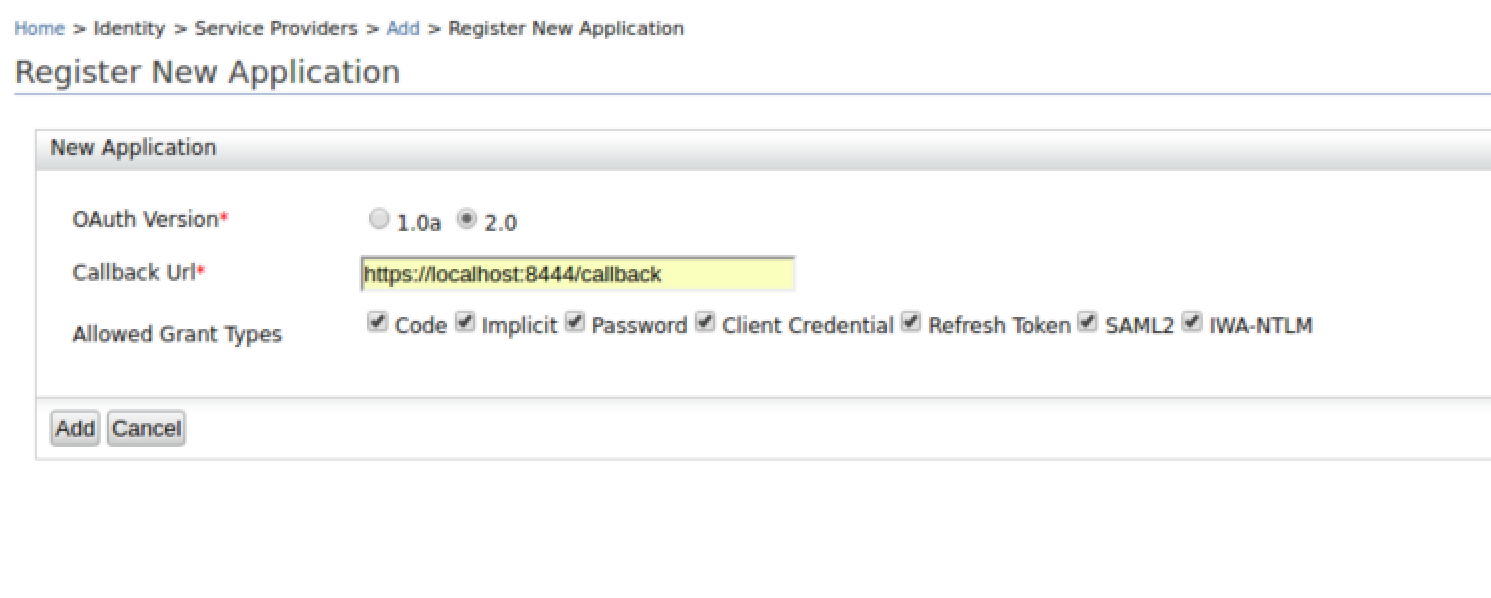
h = httplib2.Http(ca\_certs="./keys/ca.cert.pem")

resp, content = h.request(url, "GET", headers = headers)

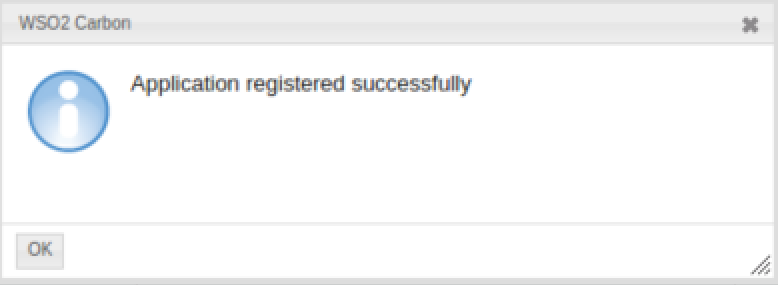
1. Start your server!
2. Now if you call:  
   python random-client.py fbas78734nerjkka233a1  
     
   this will add the following HTTP header:  
   Authorization: Bearer fbas78734nerjkka233a1  
     
   There is a further slight enhancement which is to also print out the full JSON response (so we can see the extra fields we are adding).
3. Ok, we have almost everything in place! We have an OAuth2 server. We have a client that can send a bearer token, and a server that can read the token, pass it over to the OAuth2 server and validate the response.
4. There are multiple ways defined in the OAuth2 specification to get hold of the OAuth2 token. The two ways we are going to look at are documented in this sequence diagram:  
   
5. In both approaches, before we create the token, we need to register our application to the WSO2 Identity Server.
6. Browse to:  
   <https://localhost:9443>   
     
   Sort out any SSL issues and then login as **admin/admin**  
   You should see:



1. In the left-hand bar, select **Service Providers -> Add**Add the Service Provider Name: **oauth2-example**Click **Register**  
   
2. Expand the Inbound Authentication Configuration  
   
3. Expand **OAuth/OpenID Connect Configuration**, and then click **Configure**Enter the callback URL: <https://localhost:8444/callback>

Click **Add**

1. You should see:

  
  
*Client credentials flow*

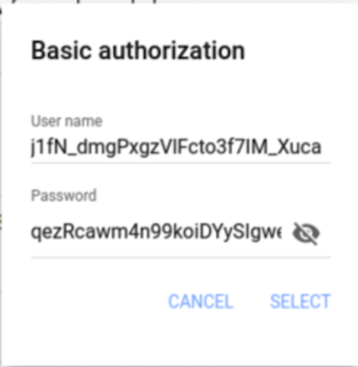
1. We now have an OAuth2 Client ID and Client Secret that identifies both the “Client Application” and the user (admin) to the Auth Server.
2. Pull up ARC and create the following request:

URL: <http://localhost:9763/oauth2/token>

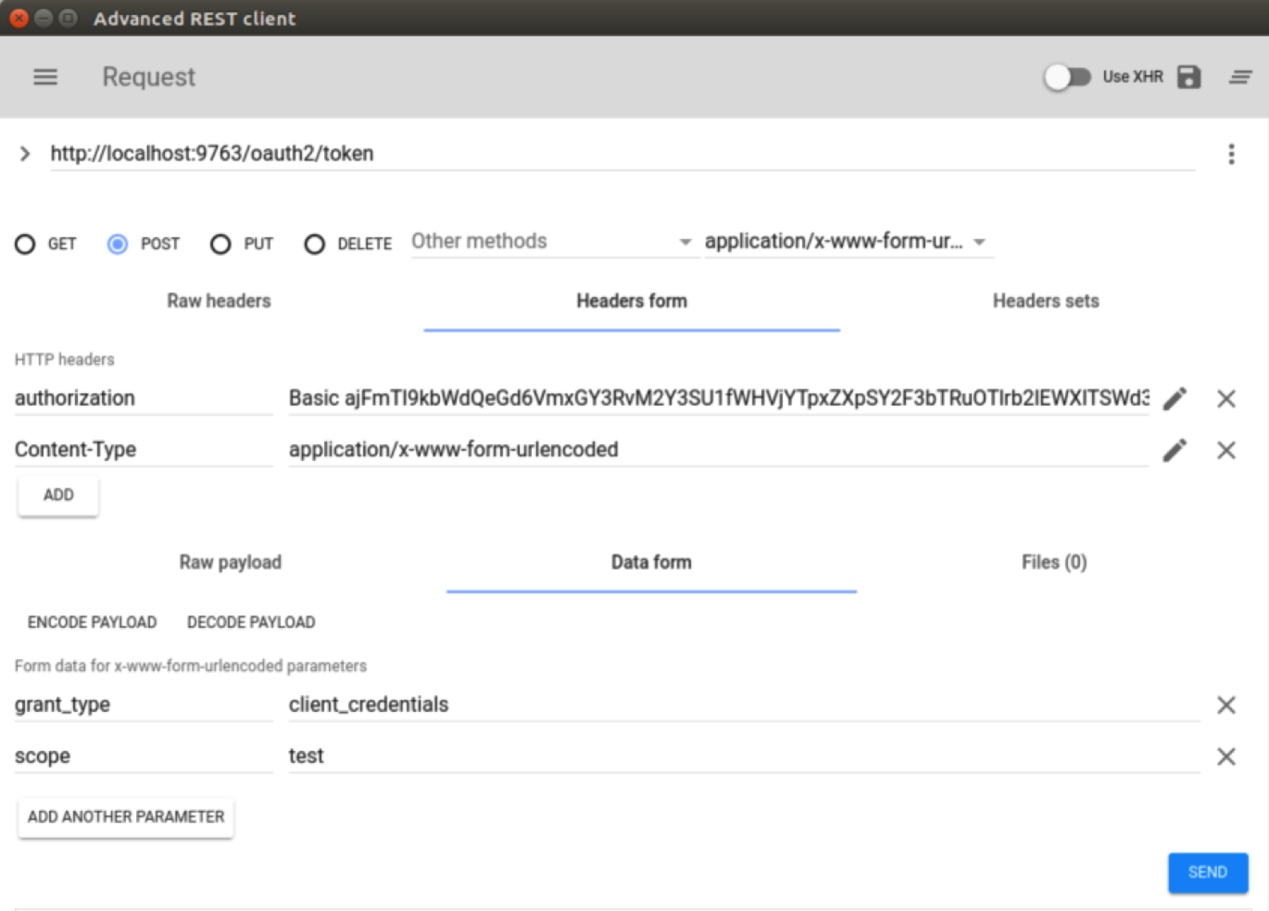
Hint: create the headers with the built in tools (**Headers Form** / then look for the pencil icon)

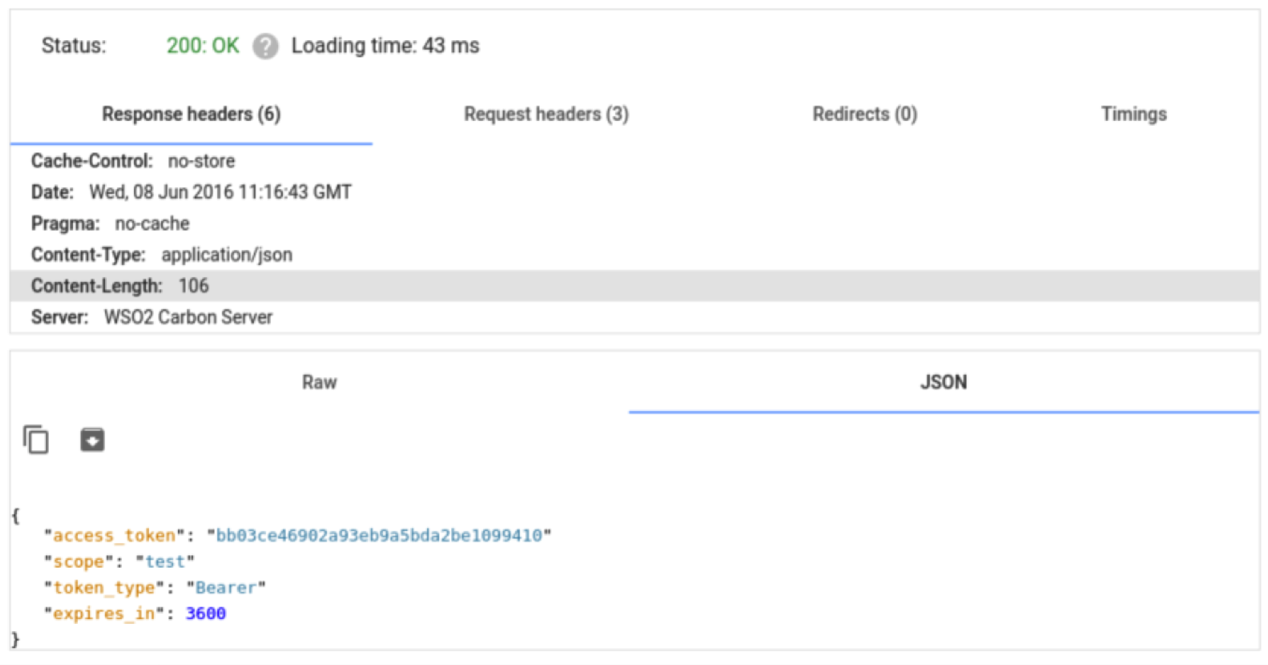
Headers:

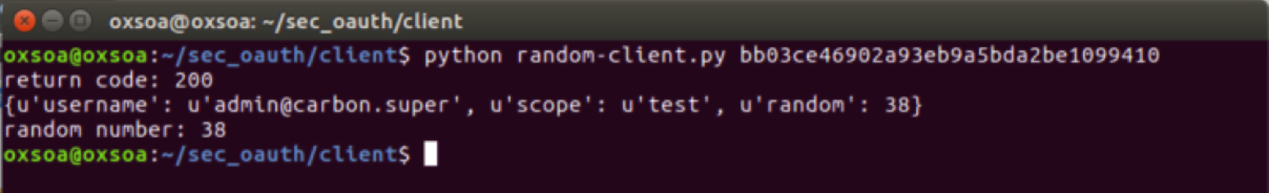
Authorization:   
Username**: <your client id copied from the IS web console>**

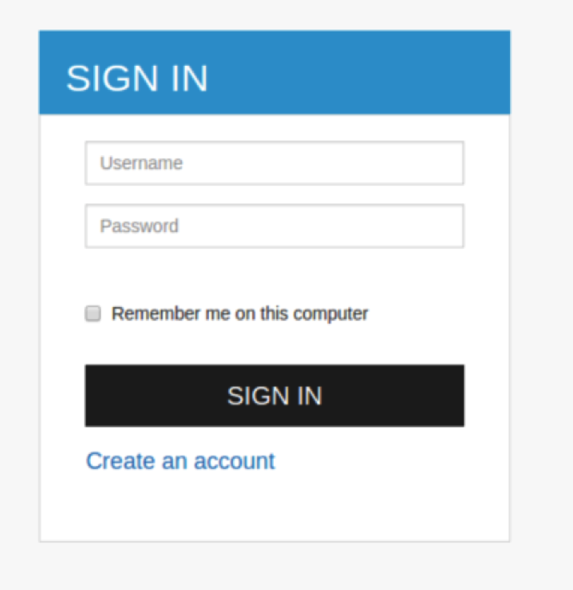
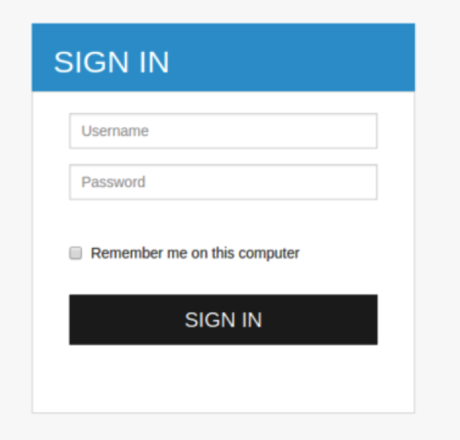
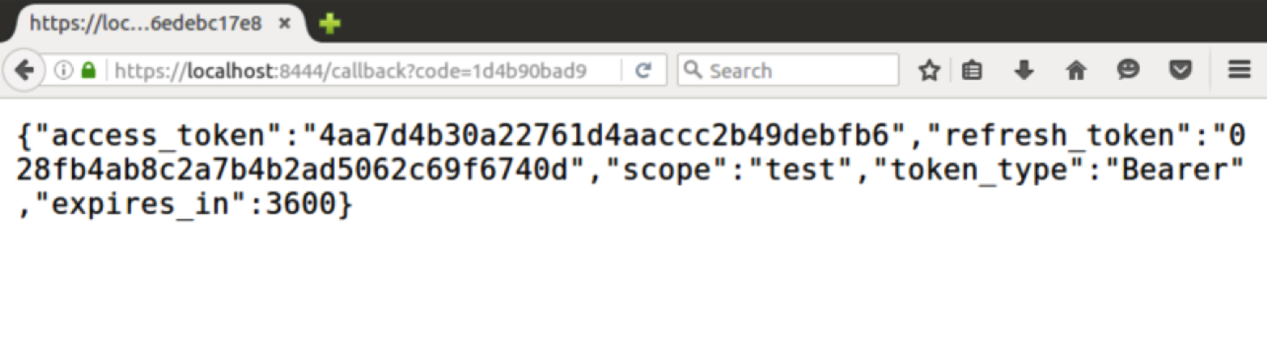
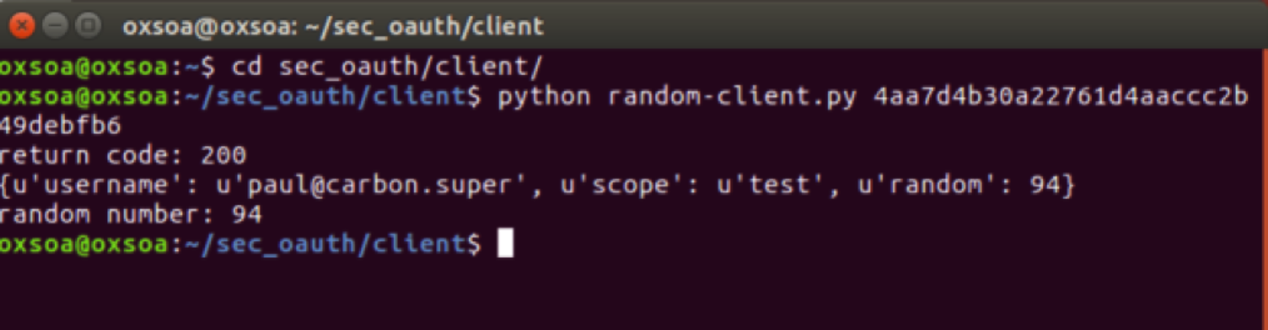
**** Password: **<your client secret from the IS web console>** Now use the ARC **Data Form** to add the following parameter. Chose the right Content-Type: **application/x-www-form-urlencoded**

cgrant\_type: client\_credentials  
scope: test

Your screen should look like:  


1. Now **Send**
2. You should see a response like:   
   
3. Copy that access token. Now open a new terminal window.   
   cd ~/sec\_oauth/client  
   python random-client.py *ce6c89211ac41044e17c7012726d583e*  
     
   *(but use your token!)*

You should see:  


1. Congrats, we have made the first type work.
2. This client\_credentials model doesn’t really implement what we would properly like, which is that a user *delegates authority* to a client to do something with a scope. The reason is that we expected the User to somehow issue this HTTP request and get the credential.
3. The better model is documented in the second flow in the sequence diagram. Please take another look.
4. To support this, we need logic running in a server[[1]](#footnote-1). Normally this would be coded into the same system as the client is running as. In other words, our client is a website so it can host this logic. Since we just have a Python command-line client, I have coded a simple server that does this. You can look at the code for this in the **callbackserver**.**js** and **oauth2token**.**js** code in the **~/sec\_oauth/server** directory.
5. Edit the callbackserver.js and replace the existing client\_id and client\_secret with yours copied from the IS Web Console.
6. Start a new terminal and run the callback server:  
   cd ~/sec\_oauth/server  
   node callbackbackserver.js
7. Now let’s create a “real” userid instead of admin/admin.
8. Log out of the Admin console and go to <https://localhost:9443/dashboard>
9. You should see:
10. Click Create an account and follow the process.
11. Now browse: <https://localhost:8444/gettoken>
12. You should be redirected to the Auth server to login.
13. Sign in using the ID that you created.
14. You are asked if you want to approve this.
15. Click **Approve**
16. You should see something like:
17. Copy the access token and try your random-client again with this token.
18. You should see something like:  
    
19. Notice that we now have a “real” userid in this response. For example, we could evaluate the scope and the username in our server logic to implement more fine grained logic.
20. Congratulations. The lab is complete!
21. **Extension**:  
    Add the refresh client to the python code and code a refresh flow.

Here is the sample refresh flow from the OAuth2 spec:  
<https://tools.ietf.org/html/rfc6749#page-47>

POST /token HTTP/1.1   
Host: server.example.com   
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW   
Content-Type: application/x-www-form-urlencoded grant\_type=refresh\_token&refresh\_token=tGzv3JOkF0XG5Qx2TlKWIA

1. This isn’t strictly true. You can google urn:ietf:wg:oauth:2.0:oob to find out more. This would have probably been the right way to code the python client, but the server based approach we have implemented is more likely to be of use to you in real life. [↑](#footnote-ref-1)