Maarten Smeets

AMIS

Securing Oracle Service Bus REST services with OAuth2

Contents

[Securing Service Bus REST services using OAuth2 2](#_Toc528332650)

[Deploy the token service 2](#_Toc528332651)

[Create a group and user 2](#_Toc528332652)

[Create a stripe, keystore, keypair 4](#_Toc528332653)

[Deploy the token service WAR 8](#_Toc528332654)

[Grant the token service access to the keystore. 10](#_Toc528332655)

[Confirm it is working as expected 12](#_Toc528332656)

[Deploy the custom OWSM policy for role based access 18](#_Toc528332657)

[Patch your environment 18](#_Toc528332658)

[Deploy the policy JAR file 18](#_Toc528332659)

[Import the policy description XML 18](#_Toc528332660)

[Secure your services 21](#_Toc528332661)

[Add and configure OWSM policies 21](#_Toc528332662)

[Confirming the service is protected 25](#_Toc528332663)

[Good to know 28](#_Toc528332664)

[Additional logging 28](#_Toc528332665)

[Which setting is located where 30](#_Toc528332666)

[The stripe, keystore name, key alias, expiration time (token validity) 30](#_Toc528332667)

[Who can access the token service 31](#_Toc528332668)

[Custom OWSM policy configuration 31](#_Toc528332669)

# Securing Service Bus REST services using OAuth2

This manual described how you can secure Service Bus REST services with OAuth2 and configure role based authorization without having to install/license additional products (such as Oracle API Platform, Oracle Entitlements Server, Oracle Access Manager). Of course this solution does not provide all the features of the mentioned products but if you do not need them, this might be a viable alternative.

This manual describes the installation, usage and test process. It does not explain the choices made and code written. For this check out my blog posts at: <https://technology.amis.nl/author/maarten-smeets/>

The required deployables mentioned in this manual can be downloaded at: <https://github.com/MaartenSmeets/oauth2/tree/master/tokenservice/TokenApplication>

This is a JDeveloper 12.2.1.3 application. Just do recursive search/replace in files (Notepad++ or JDev can do this) of 12.2.1.3 with a previous version if you require a jpr/jws for a 12.2.1 version older than 12.2.1.3. For ease of use, also jars, wars, zips have been committed which can be used as-is (no build process required if the default configuration is sufficient).

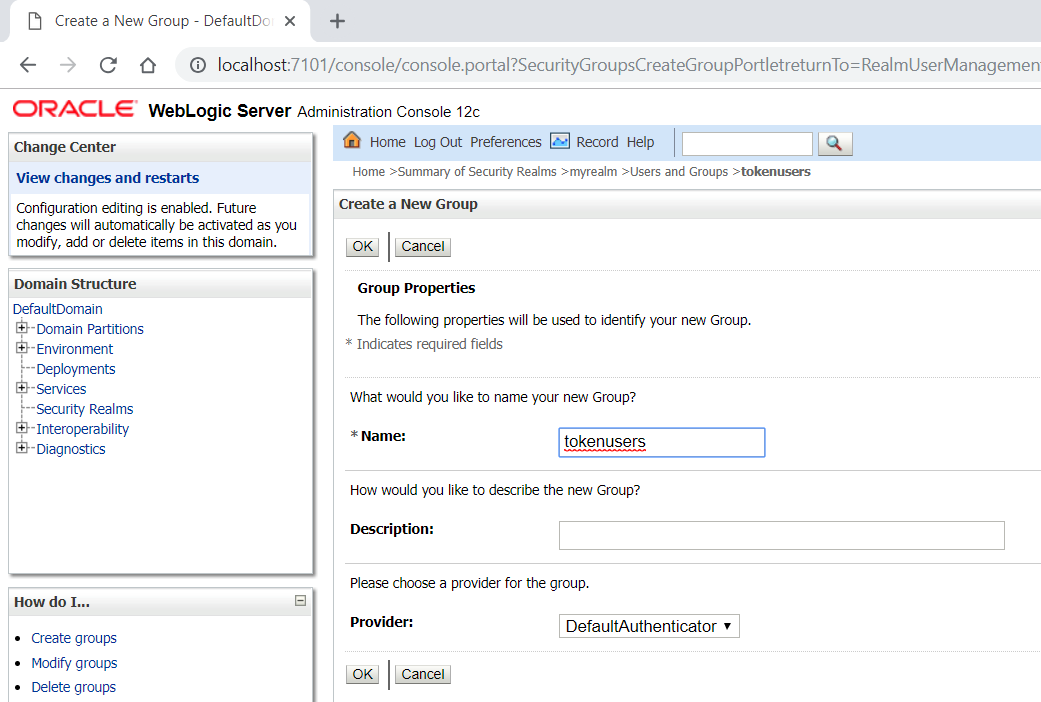
## Deploy the token service

### Create a group and user

Login into the WebLogic Console

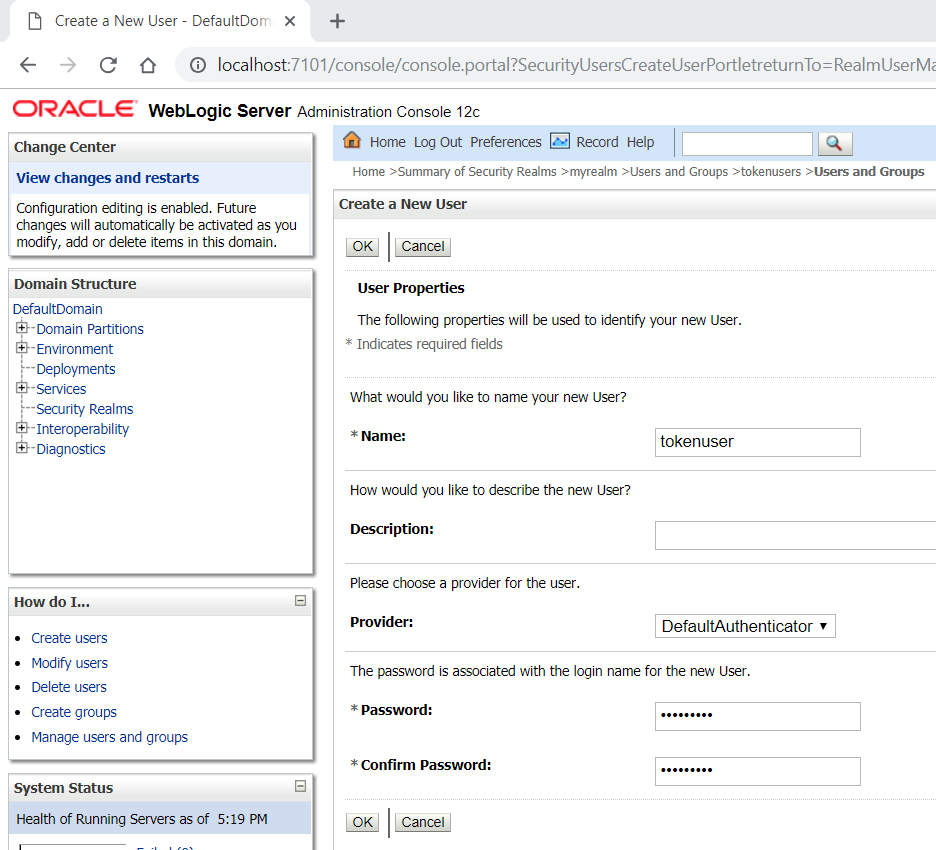
Create a group tokenusers and put a user in this group. This user is allowed to obtain tokens.

Security Realms, myrealm, Users and Groups, Groups, New

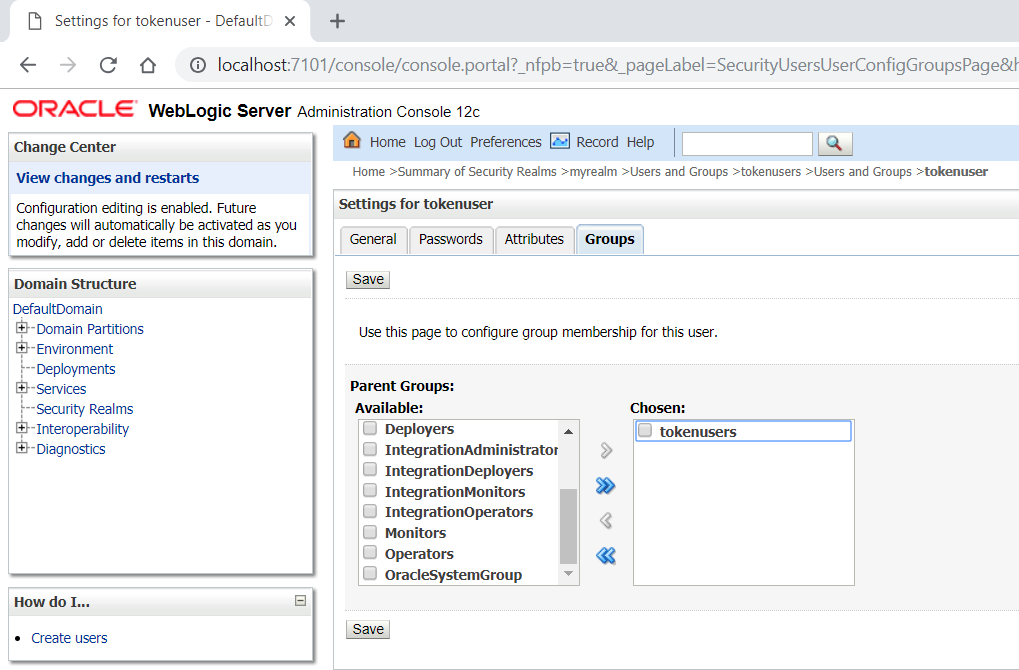


Create a new user

Security Realms, myrealm, Users and Groups, Users, New



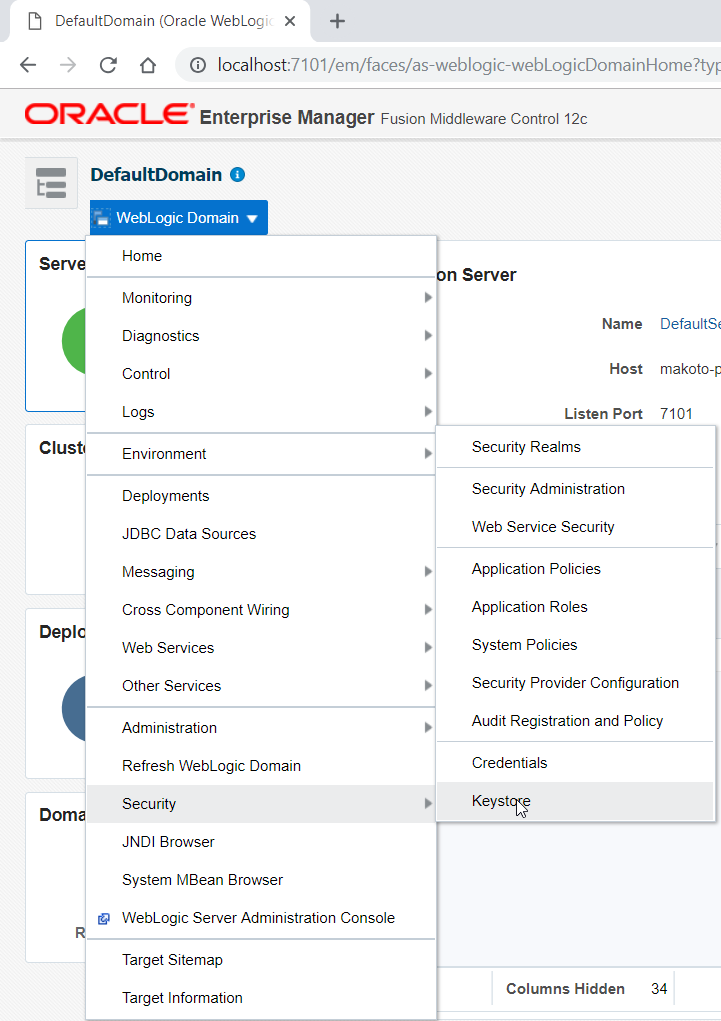
Open the user and add it to the tokenusers group



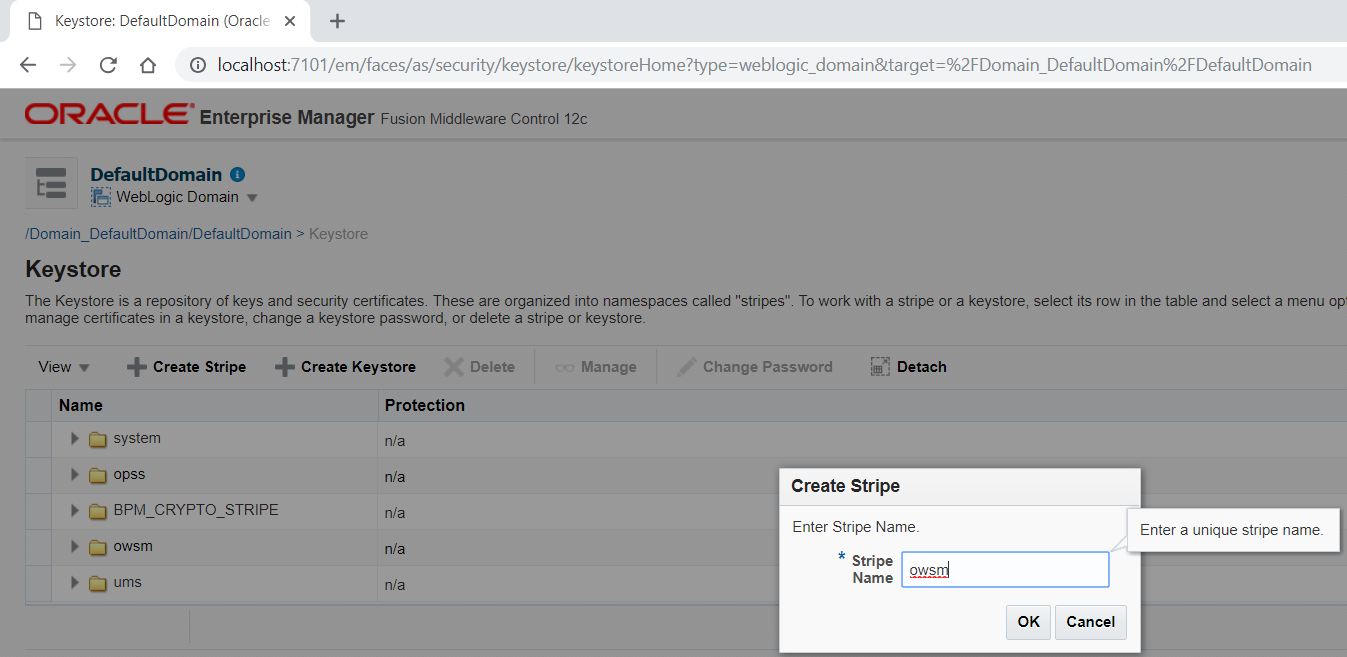
### Create a stripe, keystore, keypair

Next create a keypair in the owsm/keystore called oauth2keypair.

Login into the EM Fusion Middleware Control



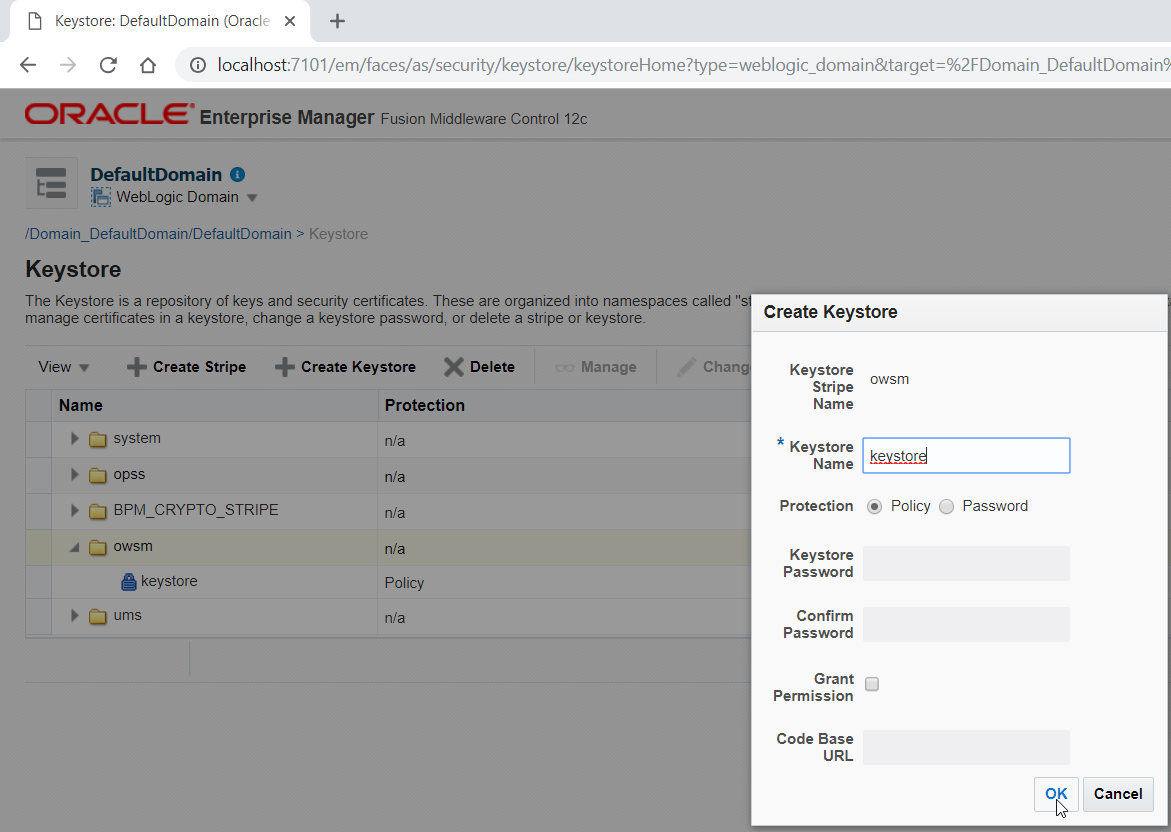
Create a stripe owsm if it is not already there



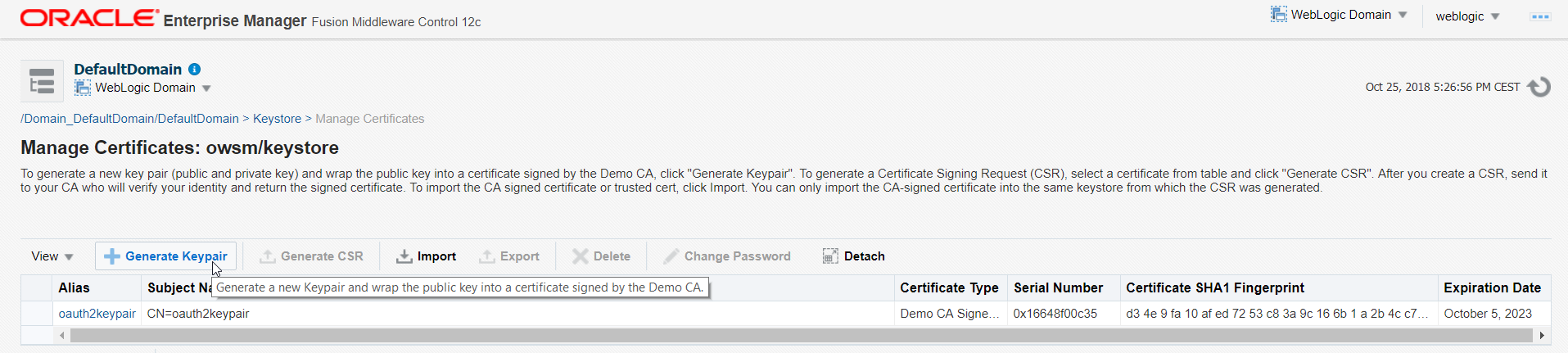
Open the stripe and create a new keystore called keystore



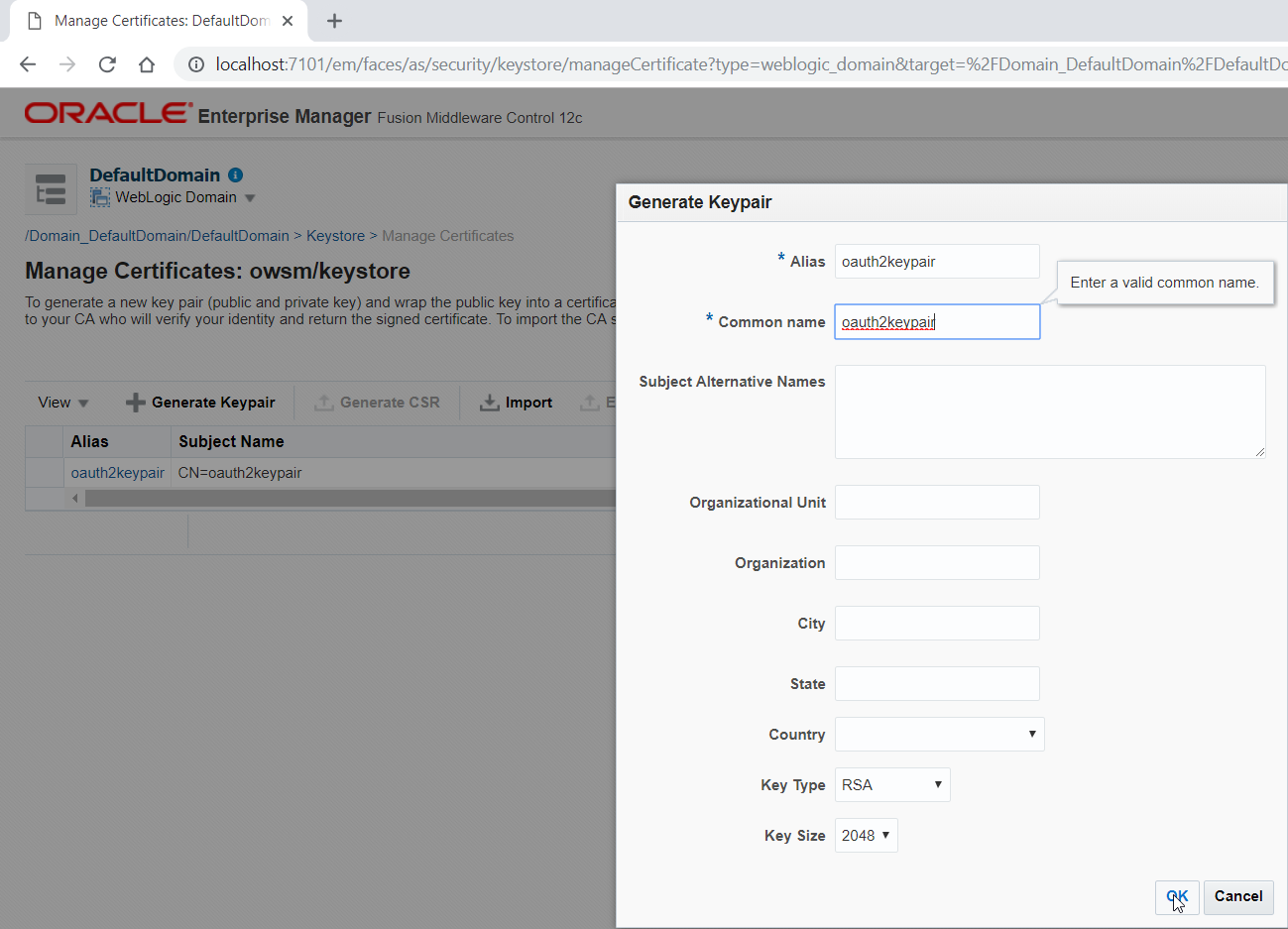
This keystore needs to be policy based since OWSM, and thus OWSM policies, do not support password based KSS keystores.



Open the keystore and click Manage. Next click Generate keypair. In a production situation you would want to import your own keypair signed with your own certificate authority here.

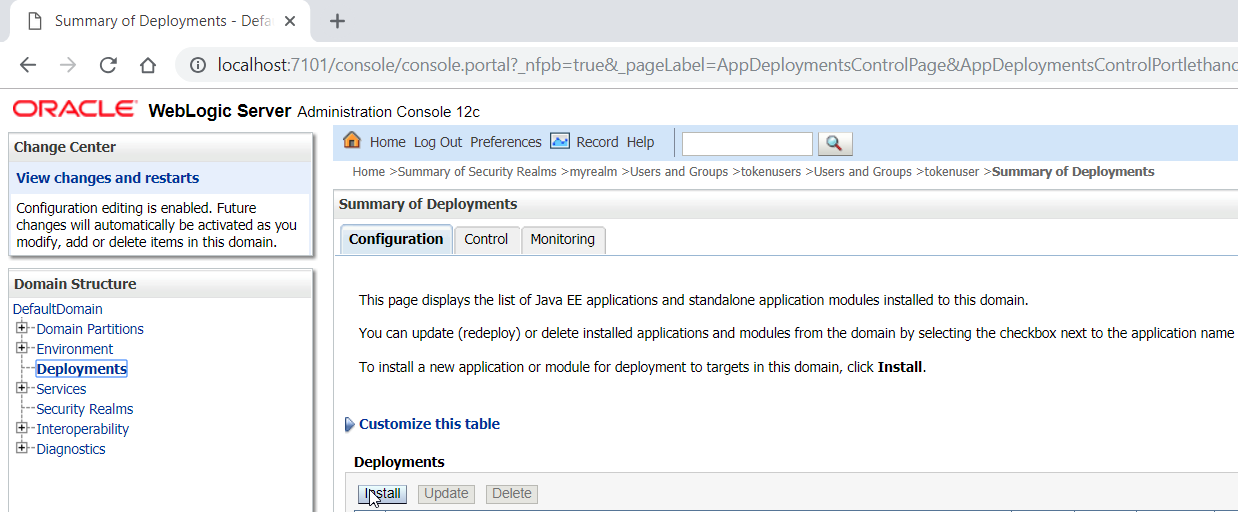


The name should be oauth2keypair for this example

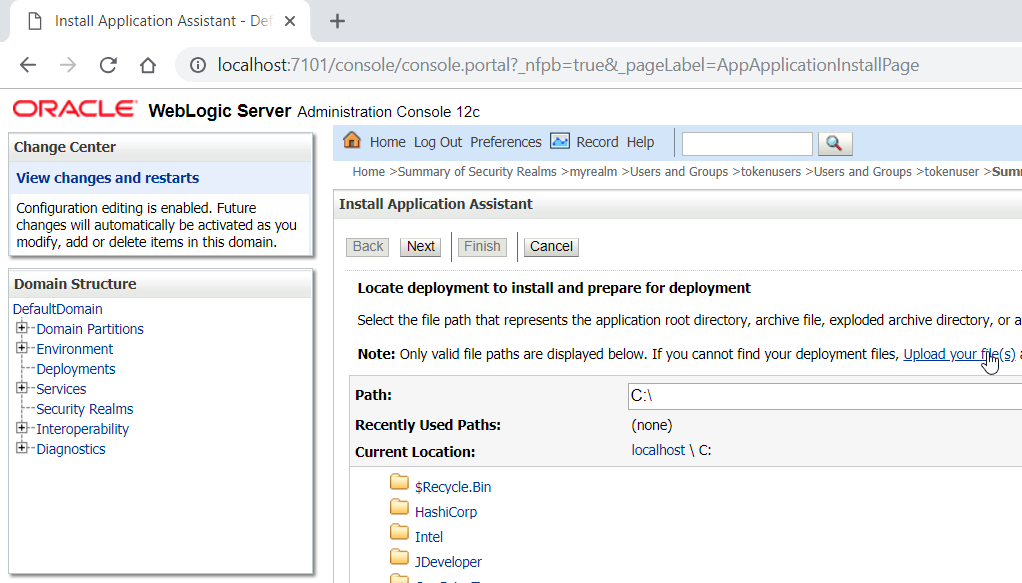


### Deploy the token service WAR

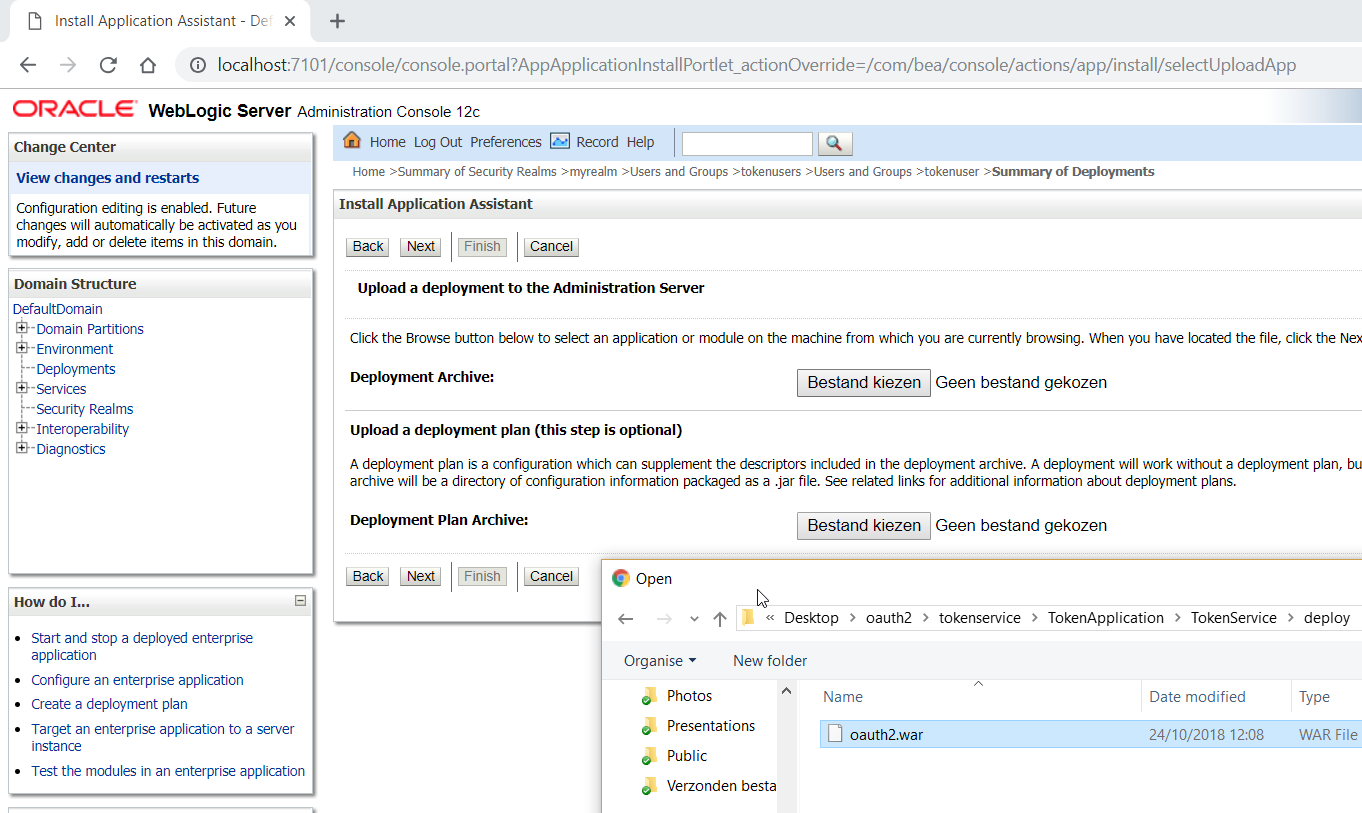
Deploy the token service WAR file from <https://github.com/MaartenSmeets/oauth2/tree/master/tokenservice/TokenApplication/TokenService/deploy> to your managed server. Login to the WebLogic Console. Go to deployments, Install.



Upload your file



Select the WAR file from the deploy folder from the token service

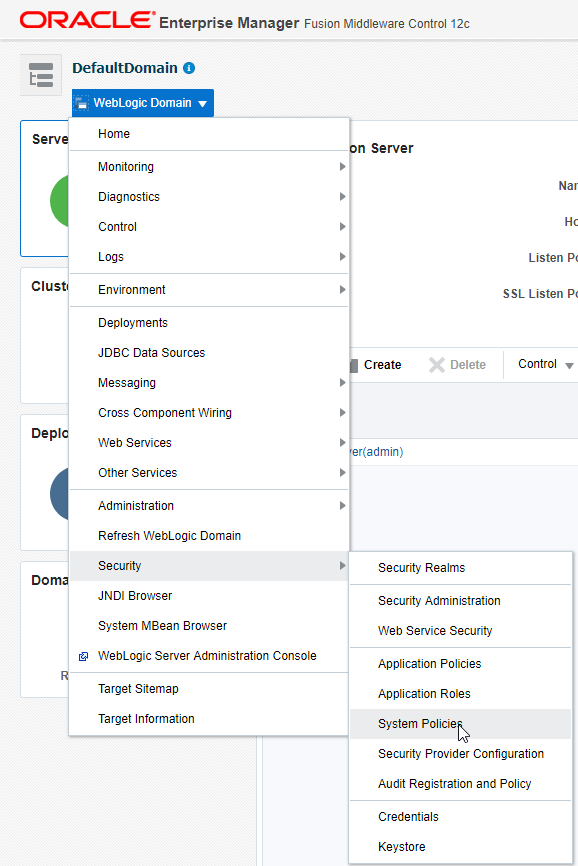


Make sure the managed server (or managed server cluster) is the only target! Default Next, Next, Next, Next, Finish.

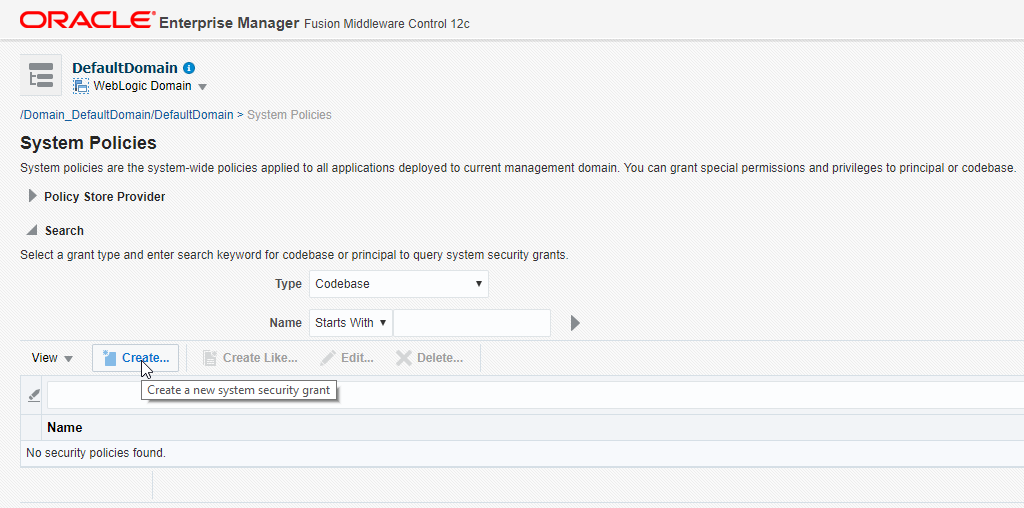
The WAR file contains a web.xml file and does not enforce TLS. Since sending basic authentication credentials and tokens over HTTP is dangerous (can easily be sniffed), you might want to update the web.xml file to enforce TLS if you can host it (<transport-guarantee>CONFIDENTIAL</transport-guarantee>). If you do TLS offloading before traffic reaches the WebLogic Server of course this is not needed.

### Grant the token service access to the keystore.

Login to the EM Fusion Middleware Control and open op the system policies page.



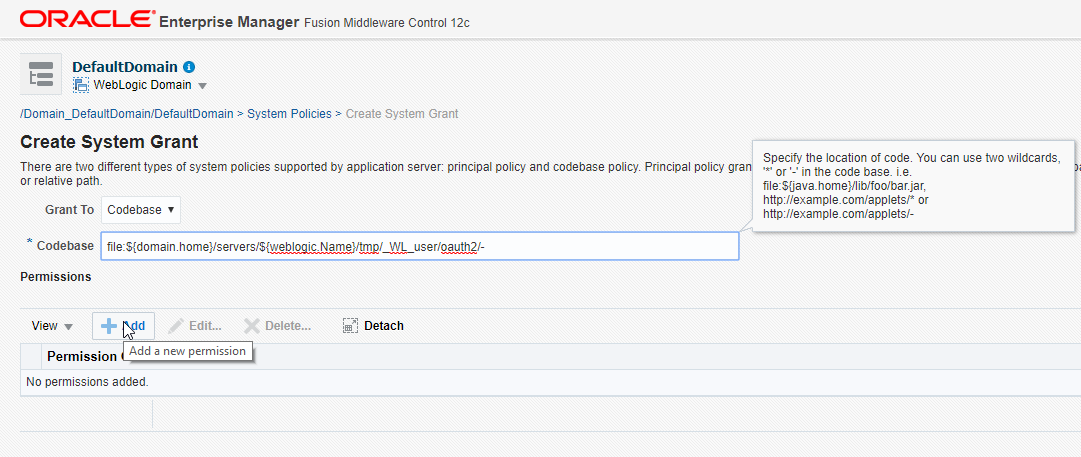
Create a new policy



Specify as codebase: file:${domain.home}/servers/${weblogic.Name}/tmp/\_WL\_user/oauth2/-

This is the location of the token service deployment. The parameters will be replaced by WebLogic.

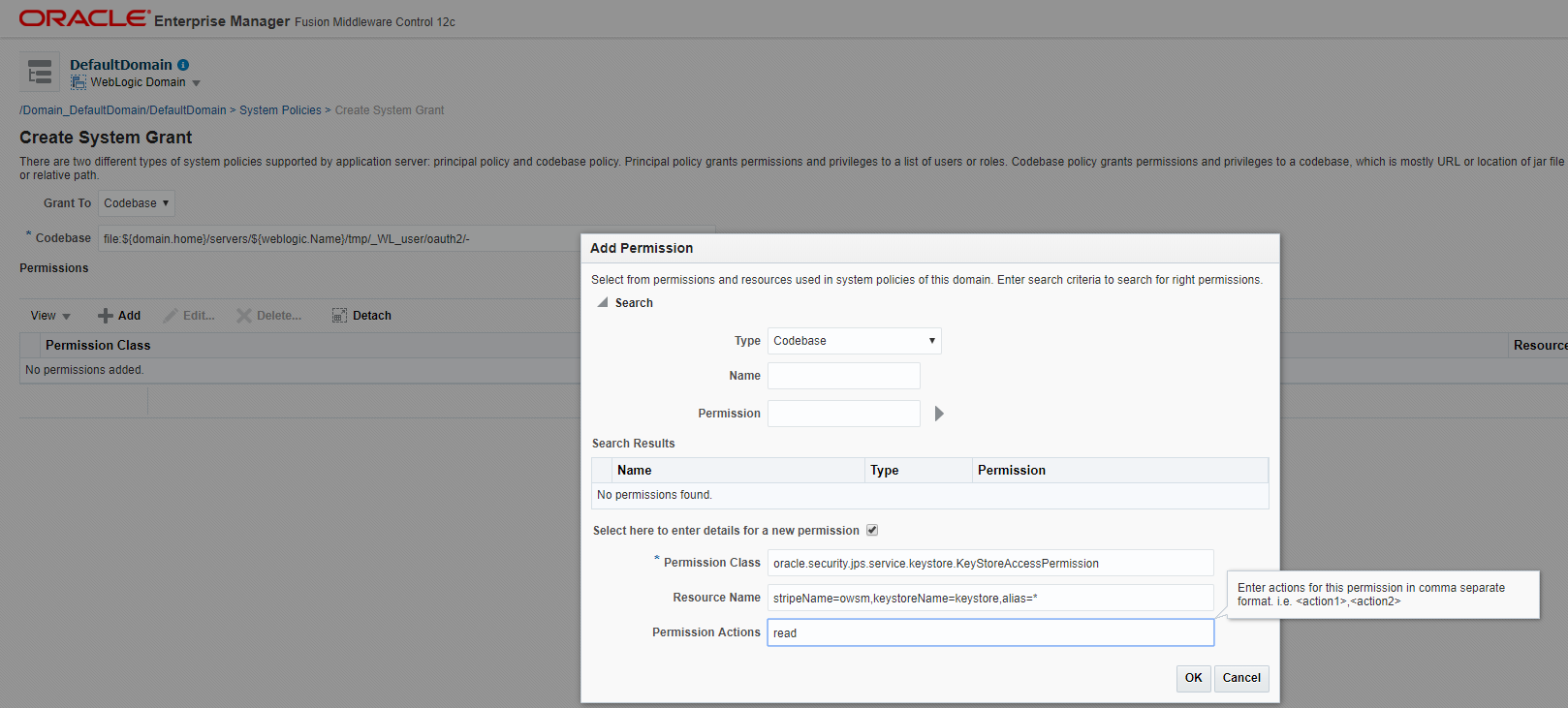
Add a new permission

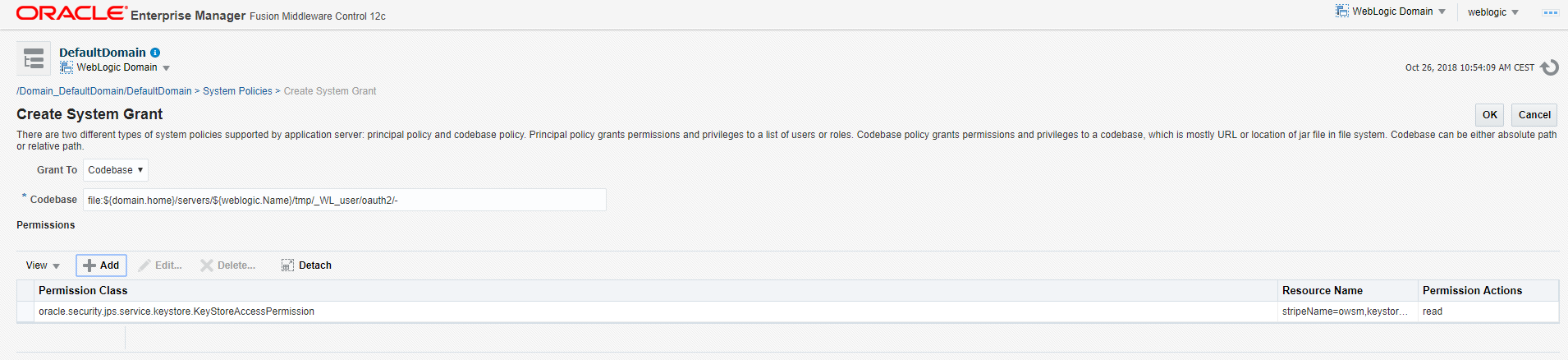


Permission Class: oracle.security.jps.service.keystore.KeyStoreAccessPermission

Resource Name: stripeName=owsm,keystoreName=keystore,alias=\*

Permission Actions: read





### Confirm it is working as expected

The token service after deployment on my Integrated WebLogic Server was available on: <http://localhost:7101/oauth2/resources/tokenservice>

When something goes wrong, no token is returned. You will then get a response like:

{

"access\_token": "",

"scope": "read write",

"token\_type": "Bearer",

"expires\_in": 0

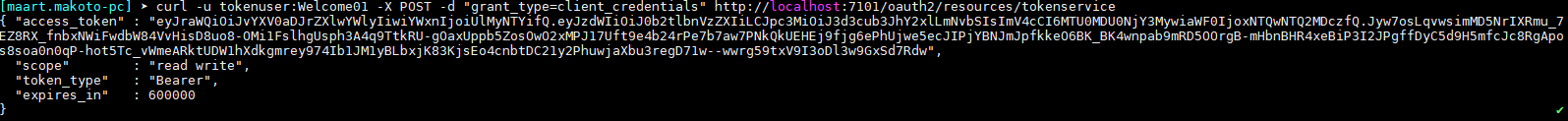
}

#### Curl

You can test the token service with curl:

curl -u tokenuser:Welcome01 -X POST -d "grant\_type=client\_credentials" http://localhost:7101/oauth2/resources/tokenservice

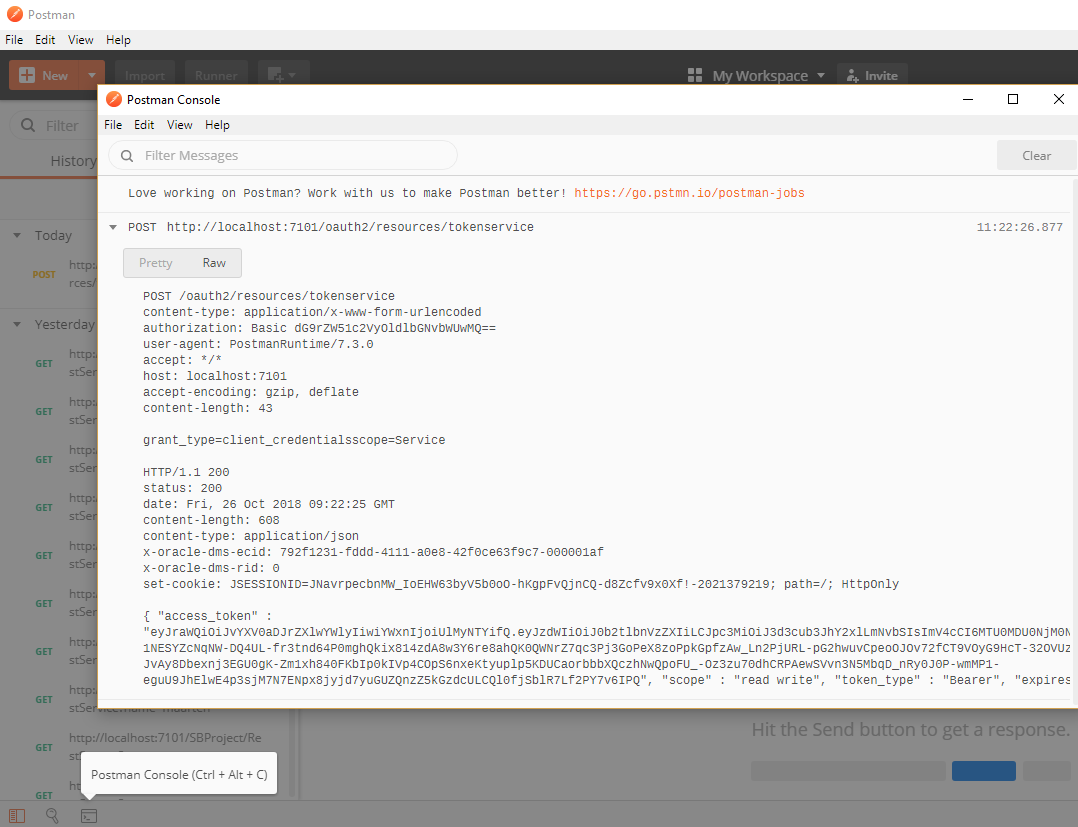
A valid response is shown below



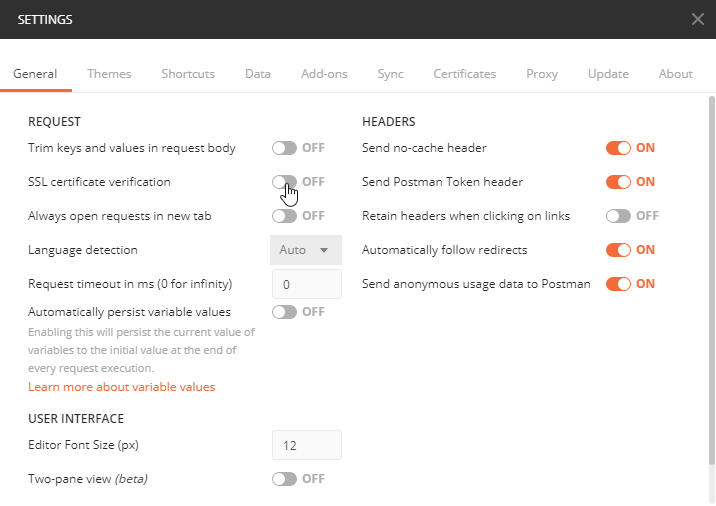
#### Postman tips/tricks

You can also test the token service with Postman (can be downloaded from: <https://www.getpostman.com/>)

Postman has a build-in console which allows you to analyse requests and responses in detail



When using Postman to access TLS secured services, you might not see responses in the console. Postman checks if (public) certificates are valid. This check can be disabled:

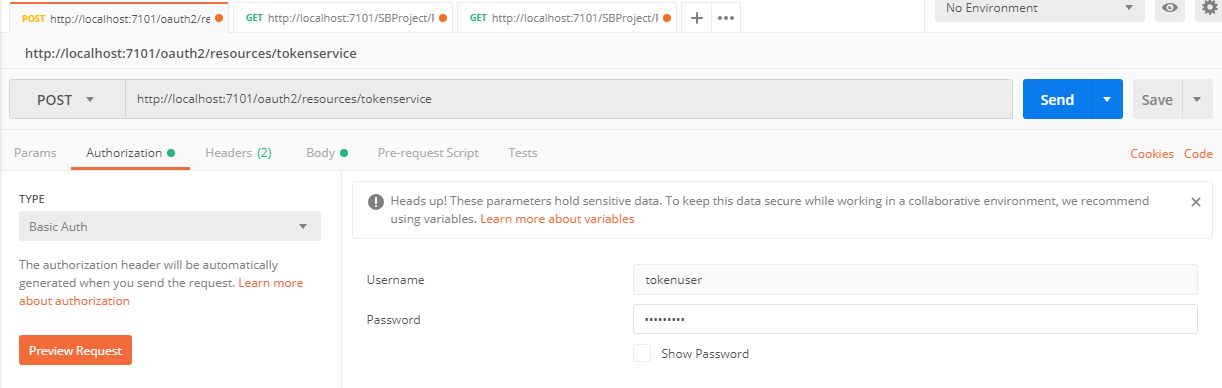


Postman has built-in support for OAuth2. You can also ‘manually’ do a request

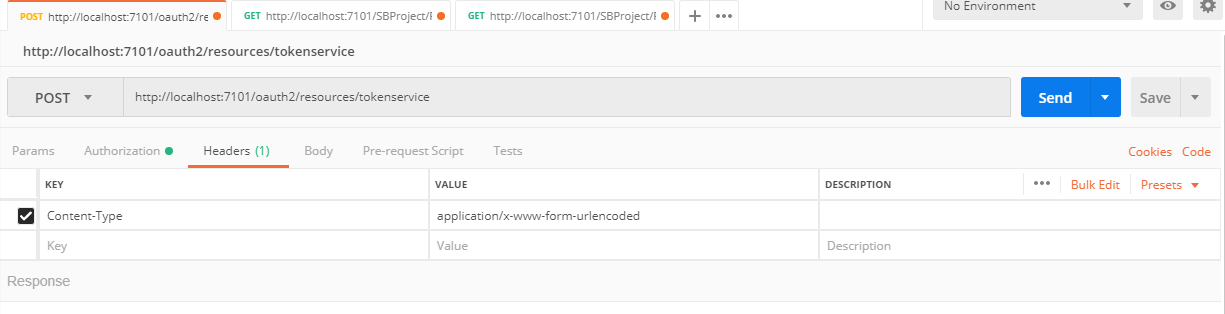
#### Postman Manual test

When testing the service manually with Postman you can do:

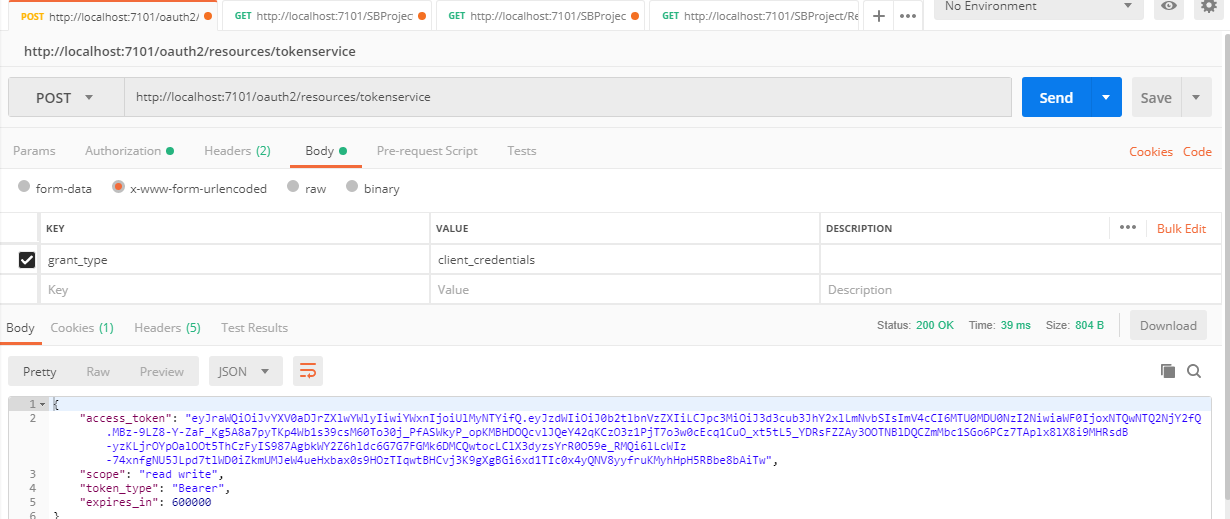
An HTTP POST request on the token service URL with Basic Authentication. The username and password used should be the same as for the user which was created and put in the group tokenusers. In this example tokenuser.



The request should have a Content-Type HTTP header of application/x-www-form-urlencoded



The body should contain a x-www-form-urlencoded grant\_type key with value client\_credentials



Above you can see a valid response. The access token is a JWT token consisting of 3 parts. A header, a body and a signature. The header and body are Base64 encoded and the signature is encrypted with the previously created keypair. The parts are separated with a ‘.’. Header and body can be Base64 decoded with for example <https://www.base64decode.org/>.

In this case the header is:

{"kid":"oauth2keypair","alg":"RS256"}

kid is a hint to the party receiving the token on which key alias they can use to decode the token. Alg is the algorithm which was used to create the signature.

And the body is:

{"sub":"tokenuser","iss":"www.oracle.com","exp":1540547266,"iat":1540546666}

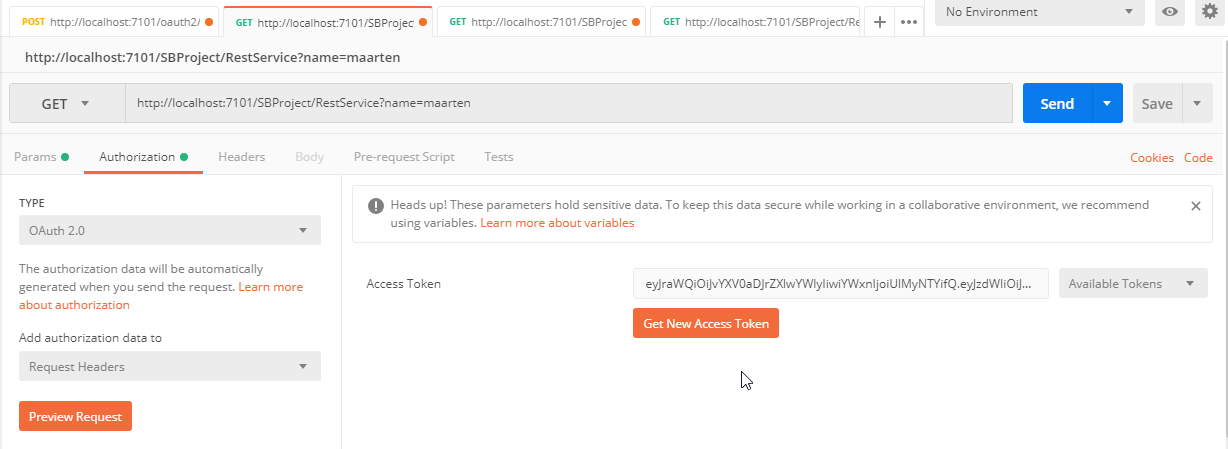
‘sub’ indicated the subject for who the token was created. You should check that this is the same as the user used in the Basic Authentication request. ‘iss’ indicates the token issuer. ‘www.oracle.com’ is trusted by default by the predefined policies. ‘iat’ indicates the issue date (epoch) of the token. You can transform this to a date with for example [www.epochconverter.com](http://www.epochconverter.com)

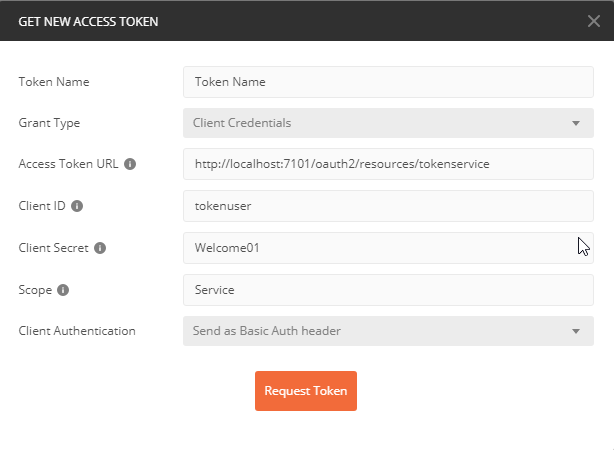


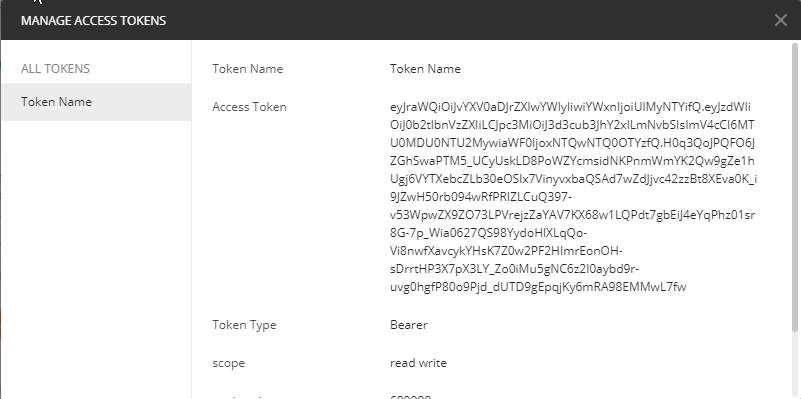
‘exp’ indicates expiry time. When the current time is past the expiry time, the token can no longer be used.

#### Postman Built-in OAuth2 support

When you have configured a service to require a valid JWT token to be accessed, you can use the build-in Postman support for OAuth2. Go to the Authorization tab, Specify OAuth 2.0 and authorization data in the Request Headers. Now you can click ‘Get New Access Token’



Indicate Grant Type Client Credentials, specify the Access Token URL and use the previously created username and password as Client ID and Client Secret respectively. The token service does not do anything with the scope yet. The Client Authentication should be ‘Send as Basic Auth header’. When requesting a token, a valid response would be something like is shown below. When you scroll down you have a ‘Use Token’ button to add the token to the request of a service call (HTTP header Authorization content: Bearer token (in which token is replaced with the actual token))



## Deploy the custom OWSM policy for role based access

### Patch your environment

In order to use the custom OWSM policy on your environment, it needs to be 12.2.1.3 or a previous version but with the following patch installed:

Patch 24669800: Unable to configure Custom OWSM policy for OSB REST Services

### Deploy the policy JAR file

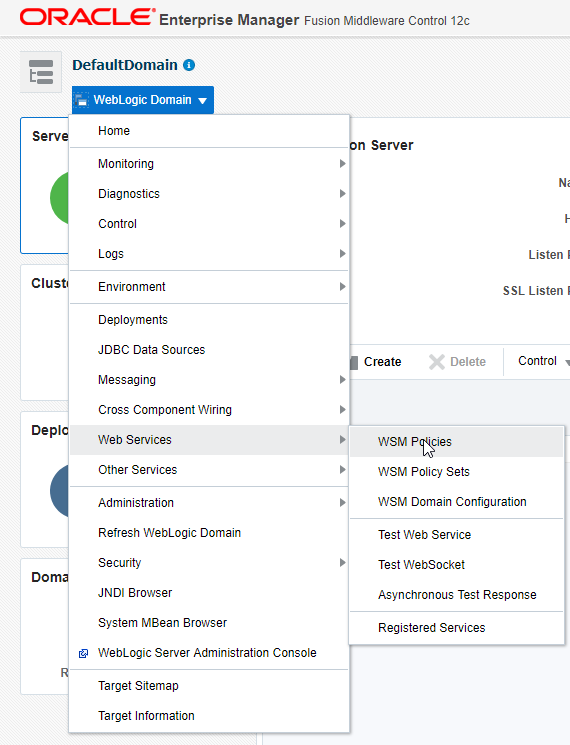
You can find the policy JAR file at: <https://github.com/MaartenSmeets/oauth2/tree/master/tokenservice/TokenApplication/CustomUserPermissionPolicy/Deployables>

This file needs to be put in the WLS\_DOMAIN/lib folder. In my case with the Integrated WebLogic server this was: C:\Users\maart\AppData\Roaming\JDeveloper\system12.2.1.3.42.170820.0914\DefaultDomain\lib

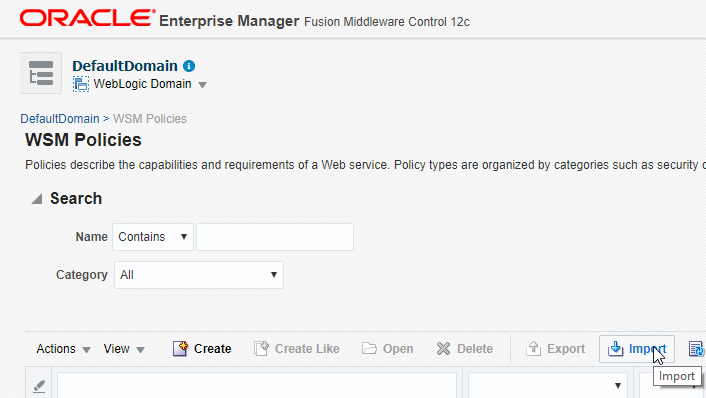
After having done this, restart the domain to make the policy available.

### Import the policy description XML

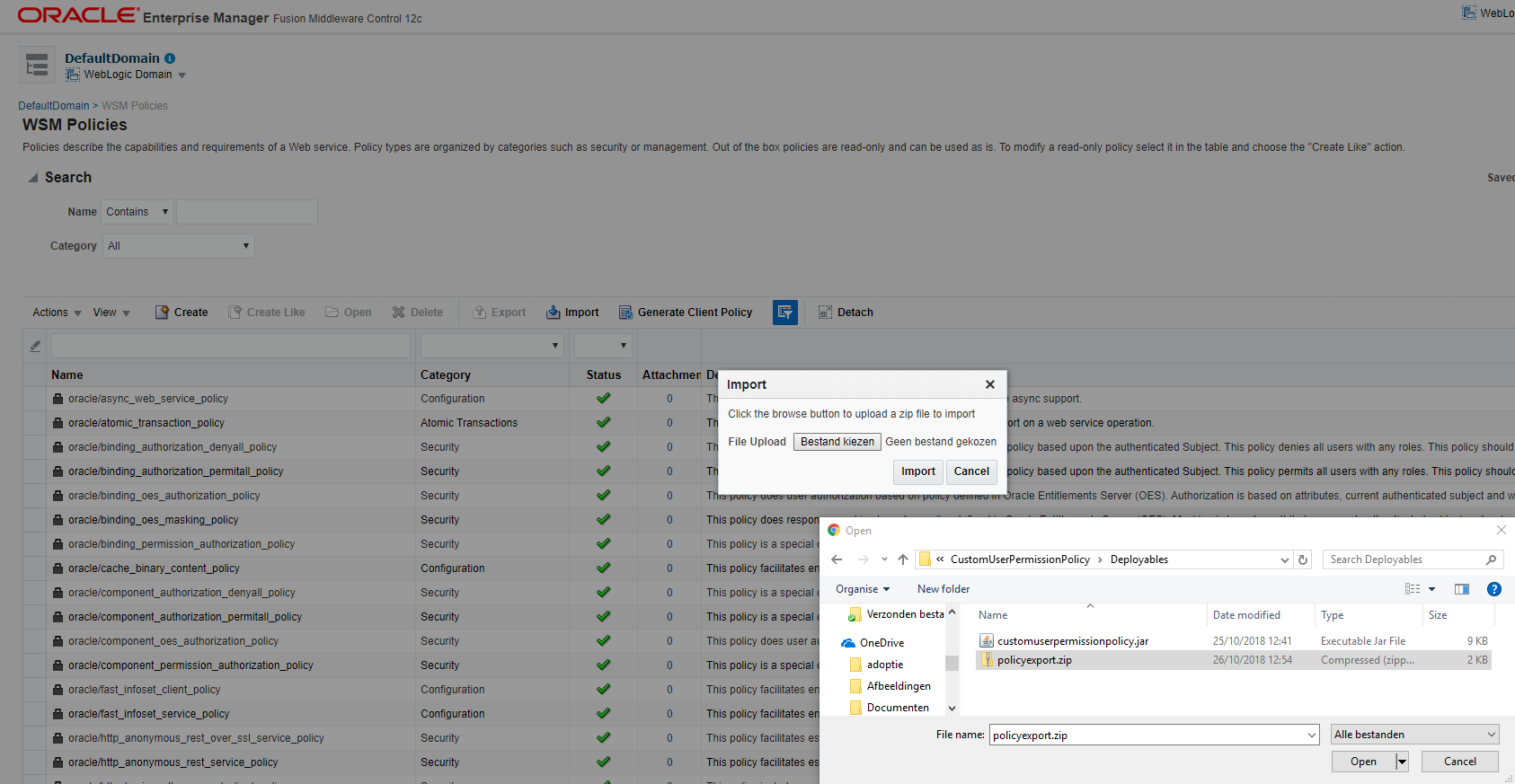
Login to the EM Fusion Middleware Control and go to the OWSM policies page;

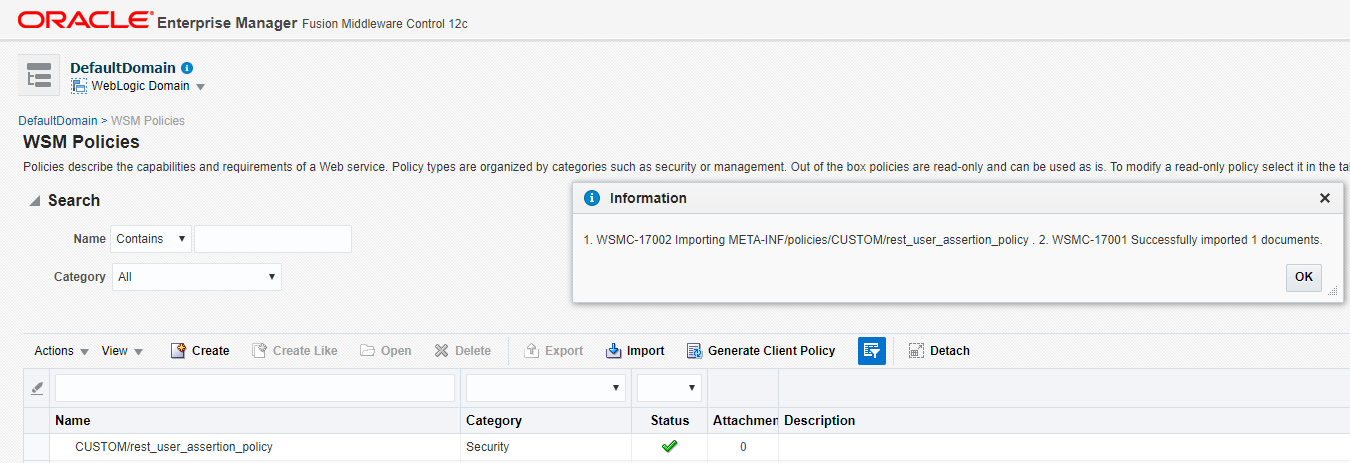


Import the custom policy



Select policyexport.zip from the Deployables folder



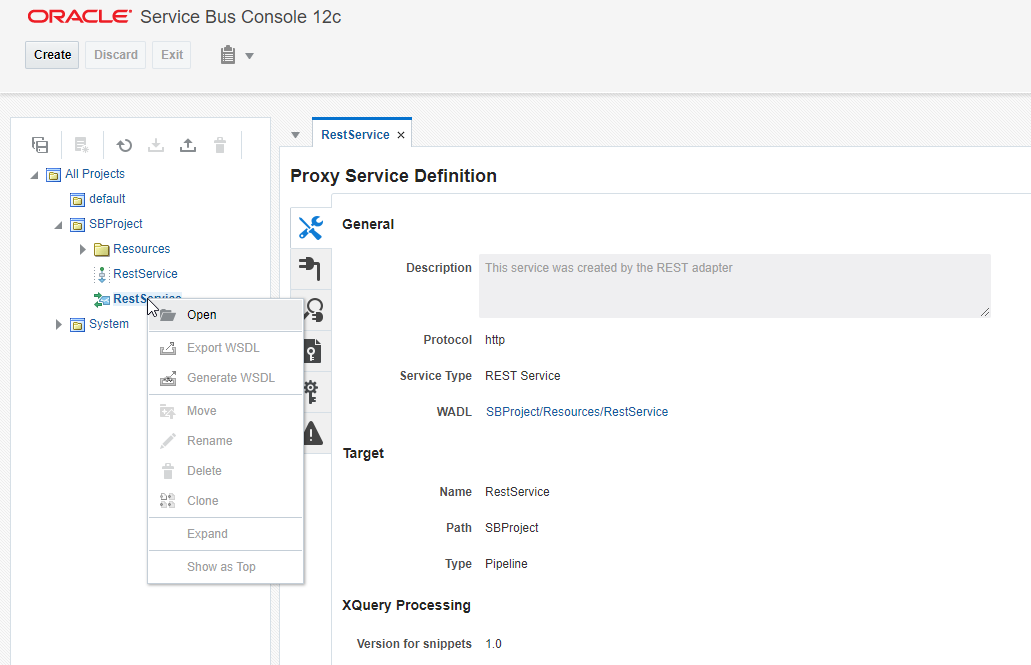


## Secure your services

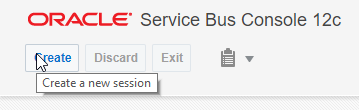
These steps are required to secure services and provide role based access. The service in this example is an Oracle Service Bus REST service.

### Add and configure OWSM policies

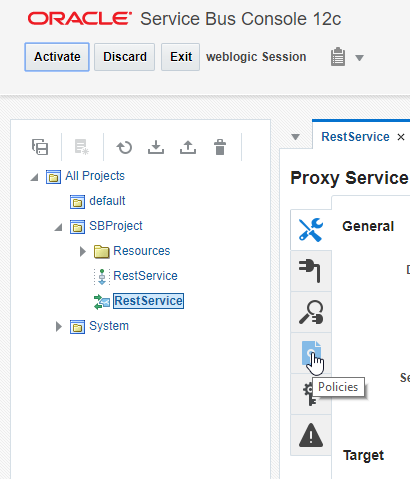
Open up the Service Bus console and navigate to the Proxy you want to protect



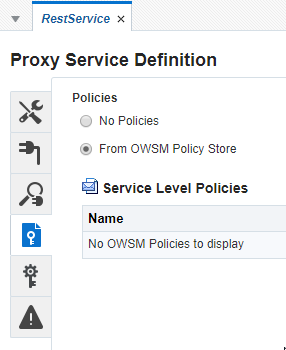
Create a session



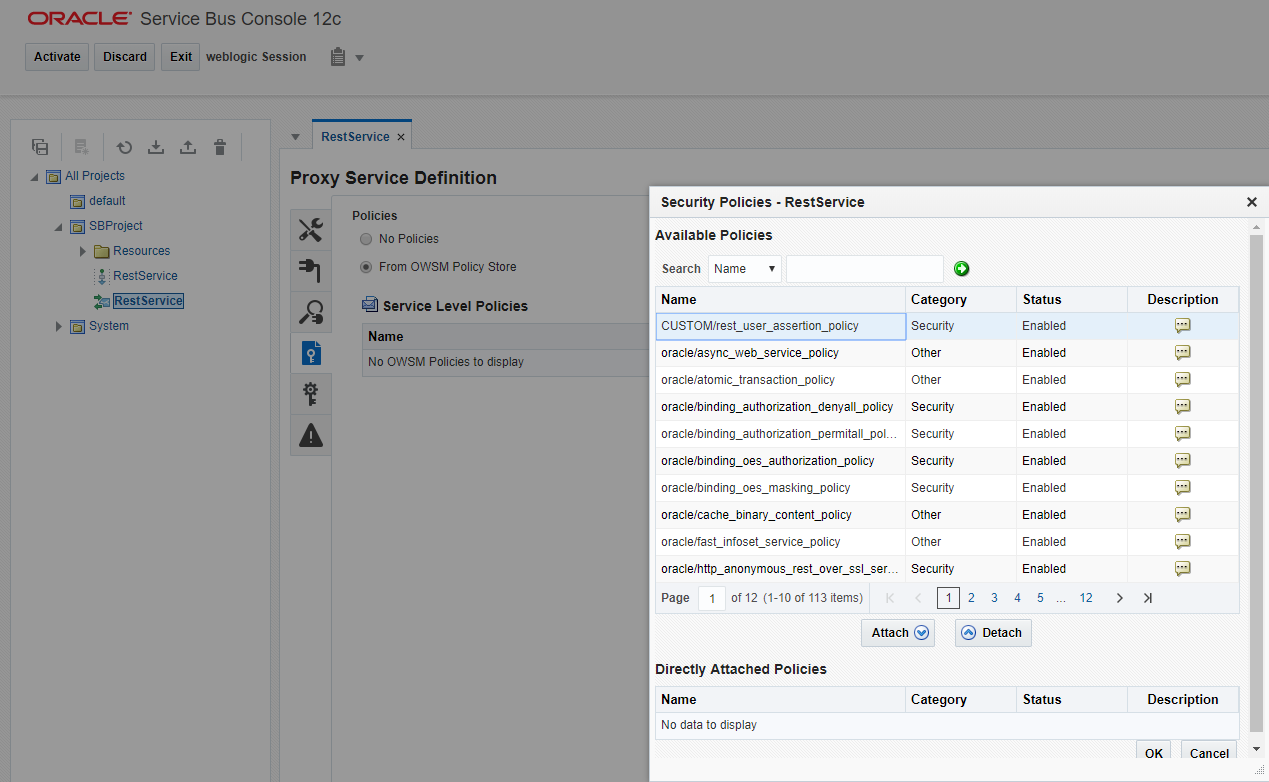
Go to the policies page



Indicate you want to use the OWSM policy store

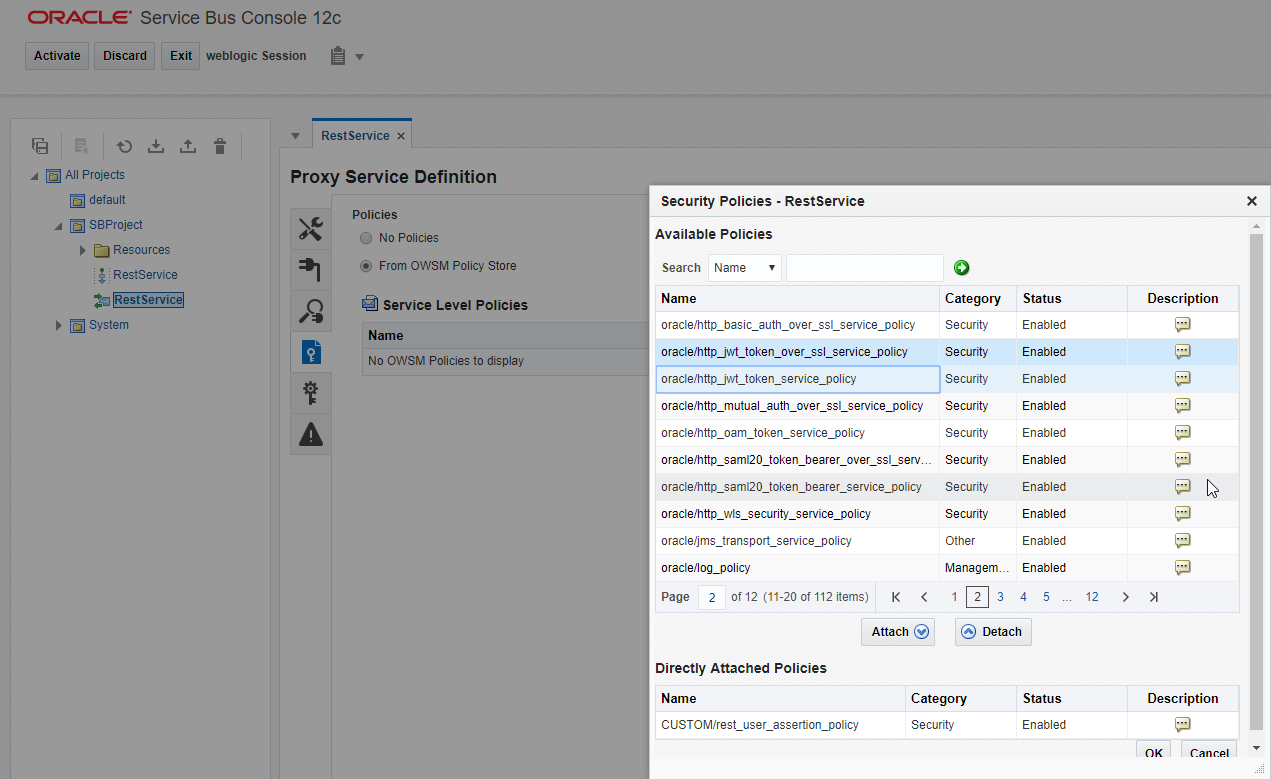


Click the attach icon

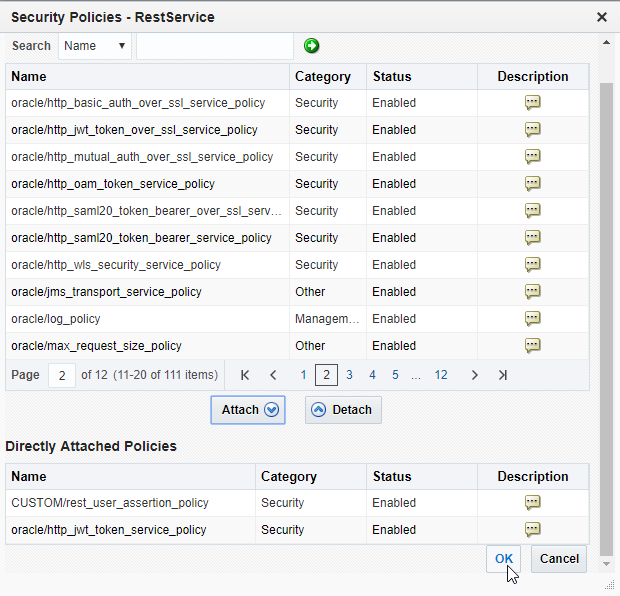


Select CUSTOM/rest\_user\_assertion\_policy and click attach

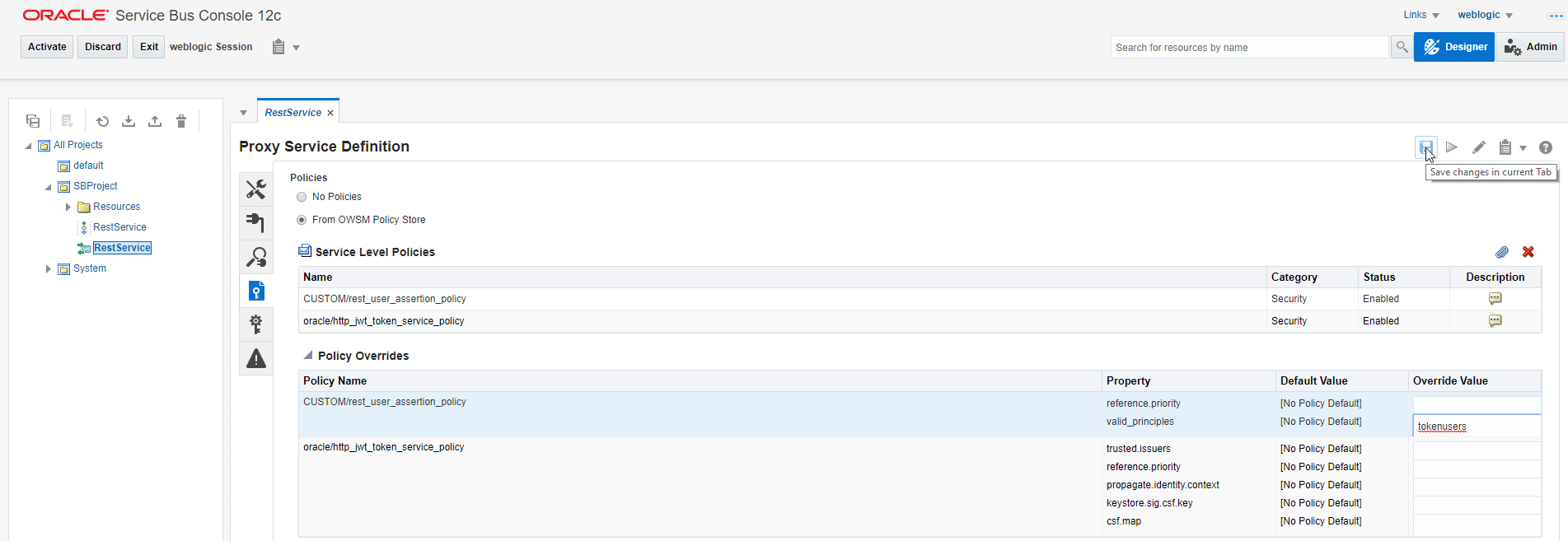
When using TLS for incoming messages on the Service Bus attach oracle/http\_jwt\_token\_over\_ssl\_service\_policy else attach oracle/http\_jwt\_token\_service\_policy



After attaching click Ok.



Indicate which principles are allowed to access the service. Principles can be WebLogic authentication provider users or groups. We configured the group tokenusers before. This can be a comma separated list of principles. Only one needs to be valid for the request to pass.



Activate your change

### Confirming the service is protected

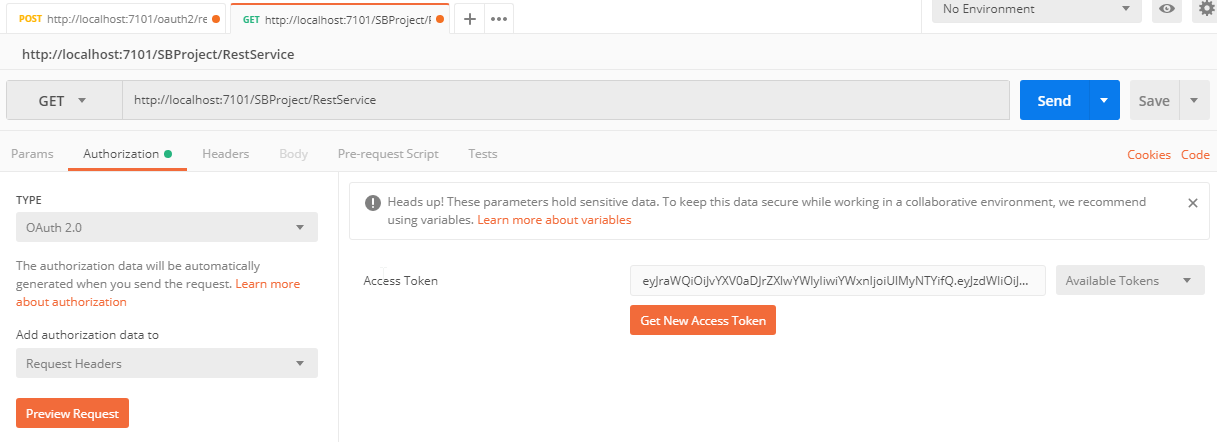
The testprocess has been described briefly when deploying the token service.

In this test situation my protected service is: <http://localhost:7101/SBProject/RestService?name=maarten>

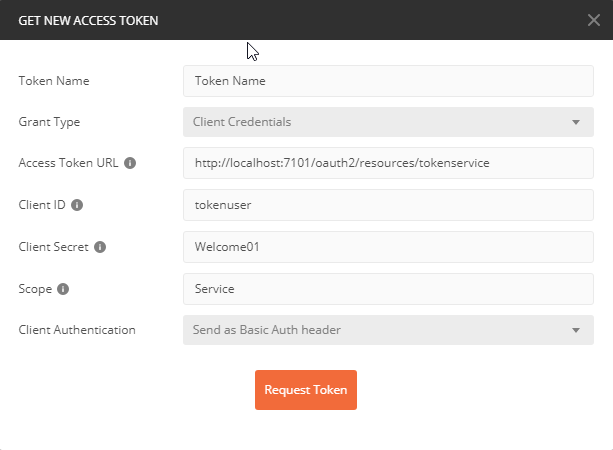
Open Postman

Configure it to do a GET request to <http://localhost:7101/SBProject/RestService?name=maarten>

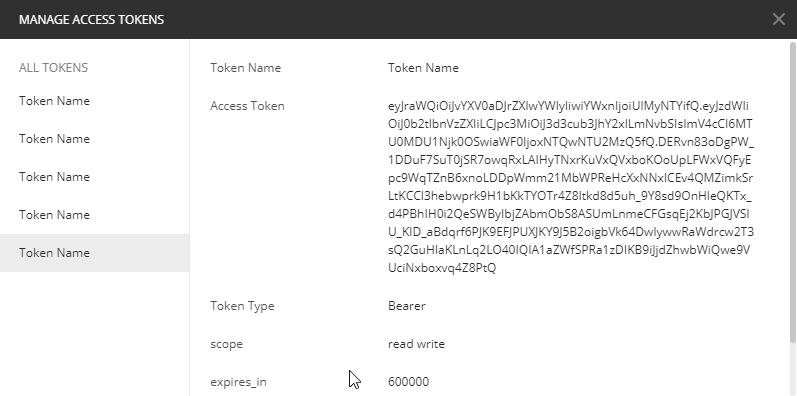
Go to the Authorization tab. Type OAuth 2, ‘Add Authorization data to Request Headers’



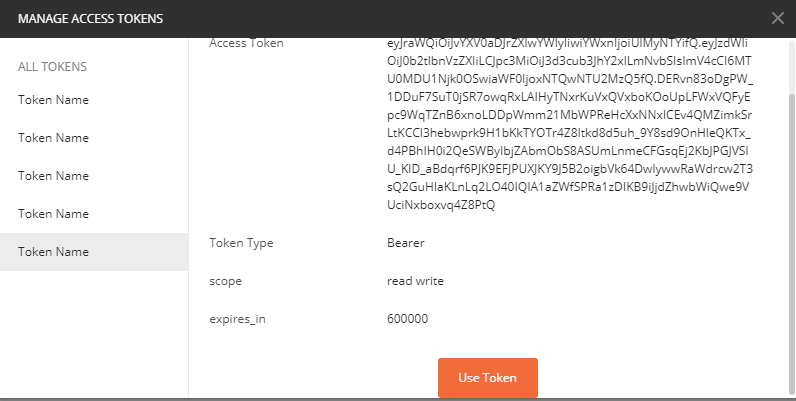
Get New Access Token



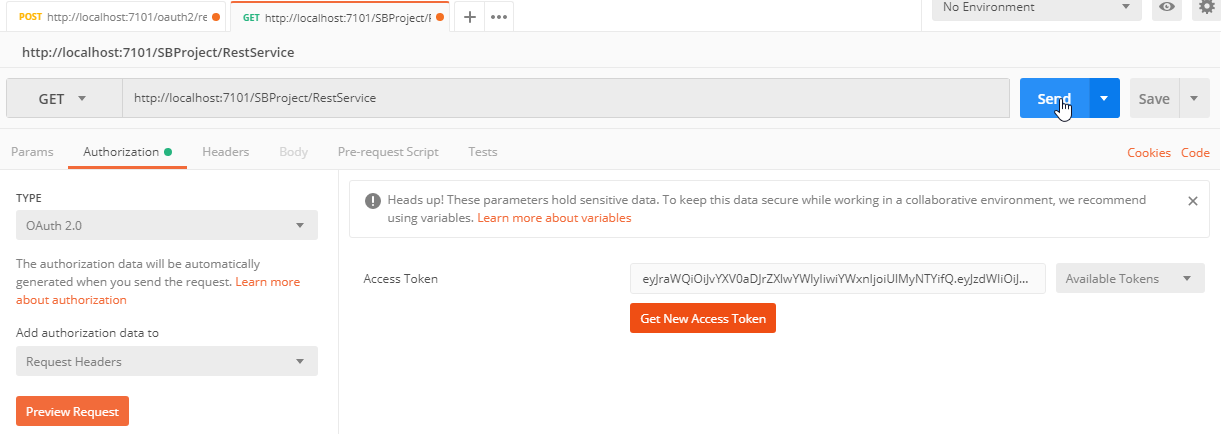
Request Token



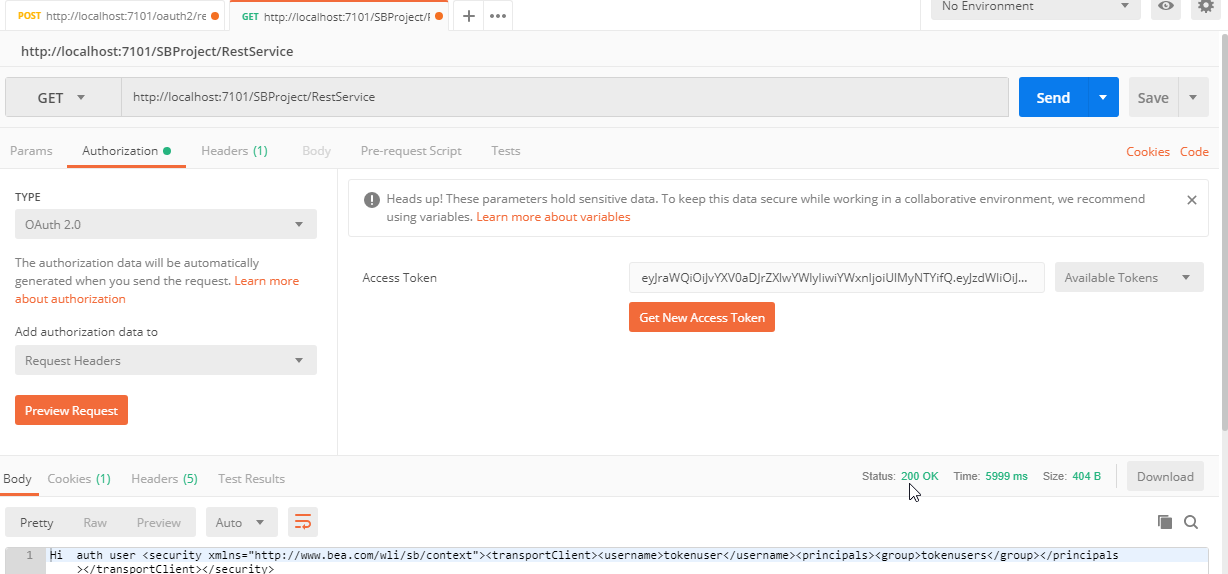
Use Token



Send a request



Confirm the response gives the expected result. HTTP 200 = OK



In the Service Bus you can access the authenticated user at $inbound/ctx:security/ctx:transportClient/ctx:username

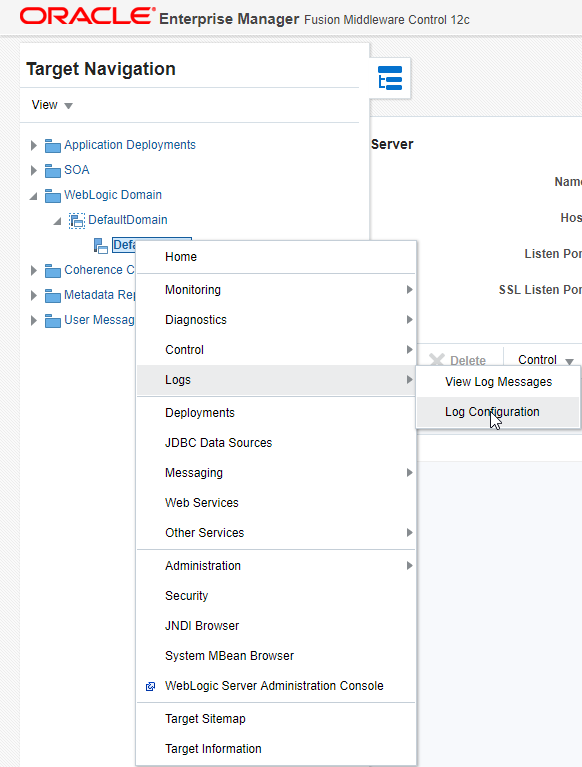
You can confirm several things during testing:

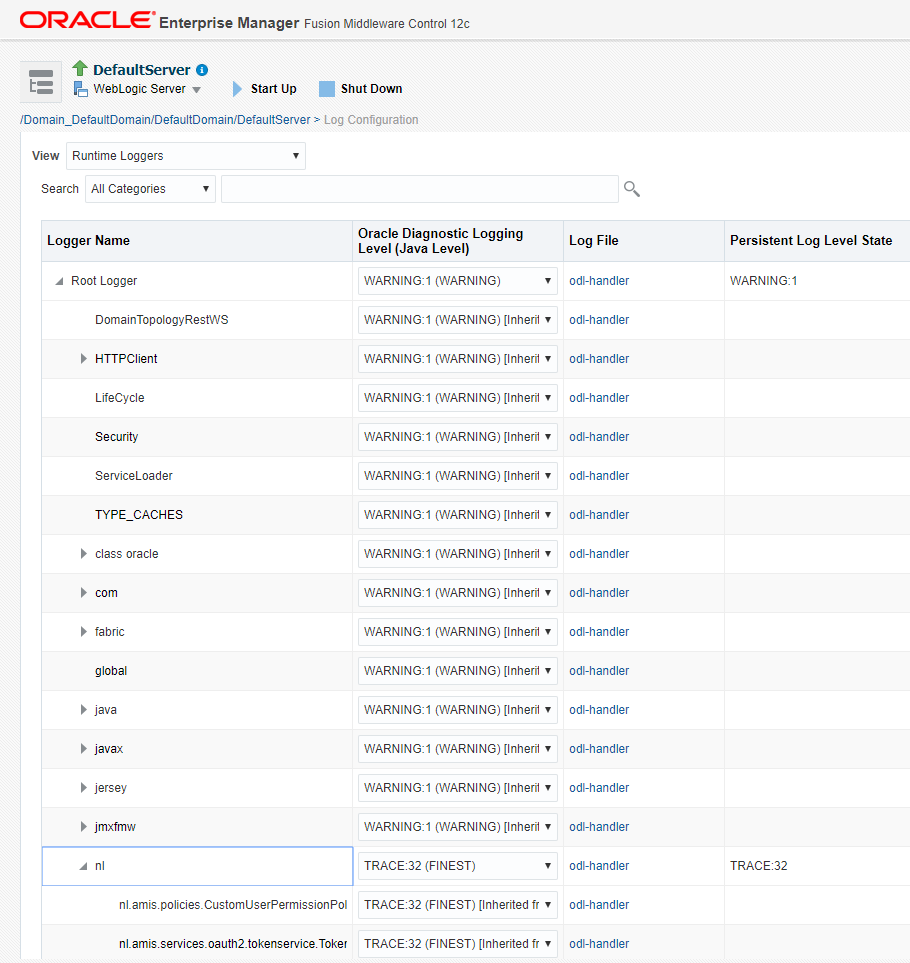
* Try to request a token for a user not in the tokenusers group. The tokenservice should not generate a token for such a user.
* Try to use a token of a user which is not in a role specified as configuration parameter of the custom policy. The reply should indicate Unauthorized. HTTP status code 401
* Try specifying several principles and a user having for example only the last one. The user should be able to access the service with the token.
* Try to manipulate the token. E.g. decode the body, replace the subject, encode the body. The JWT token validation should fail.

# Good to know

## Additional logging

The OAuth2 token service and custom OWSM policy both use the ADF logger. Log messages will appear in the servers diagnostics log files. You can increase the log level using the normal EM log configuration settings:





## Which setting is located where

### The stripe, keystore name, key alias, expiration time (token validity)

These can be configured in TokenService\src\nl\amis\services\oauth2\tokenservice\TokenApplication.properties

Stripe owsm with keystore keystore are used by default by the predefined OWSM policies oracle/http\_jwt\_token\_over\_ssl\_service\_policy and oracle/http\_jwt\_token\_service\_policy. Should you wish to change them, you need to override the policy configuration and define credential store framework entries. Mind that the meaning of the parameters and credential store framework entries differ for KSS and JKS keystores. OWSM does not support KSS keystores with passwords. Detail on how to do the is out of scope for this manual. See for more information: <https://technology.amis.nl/2017/09/24/oracle-soa-suite-and-weblogic-overview-of-key-and-keystore-configuration/>.

The key alias is also the kid in the header of the JWT token and will be used by the party validating the token to obtain the public key in order to use it to verify the signature.

After you change the properties file and do a redeploy, sometimes the new properties are not picked up. Restart the

### Who can access the token service

This is defined in the web.xml and weblogic.xml of the token service

web.xml

* The token service is protected with basic authentication
* The token service can be accessed with a POST request over HTTP
* Someone in the security role tokenusers is allowed to access the service

weblogic.xml of the token service

* The security role tokenusers maps to the WebLogic principle tokenusers. This is the name of a group defined in the WebLogic security realm.

The token will be generated for the user who did the Basic Authentication request. This user is visible as sub (Subject) in the body of the token.

### Custom OWSM policy configuration

The custom OWSM policy consists of 2 deployments.

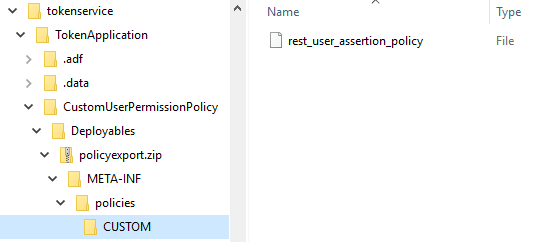
A JAR file

* This JAR contains the Java code of the policy. The Java code uses the parameters defined in the file below.
* a policy-config.xml file. This file indicates which class is implementing the policy. Important part of this file is the reference to restUserAssertion. This maps to an entry in the file below

A policy description ZIP file

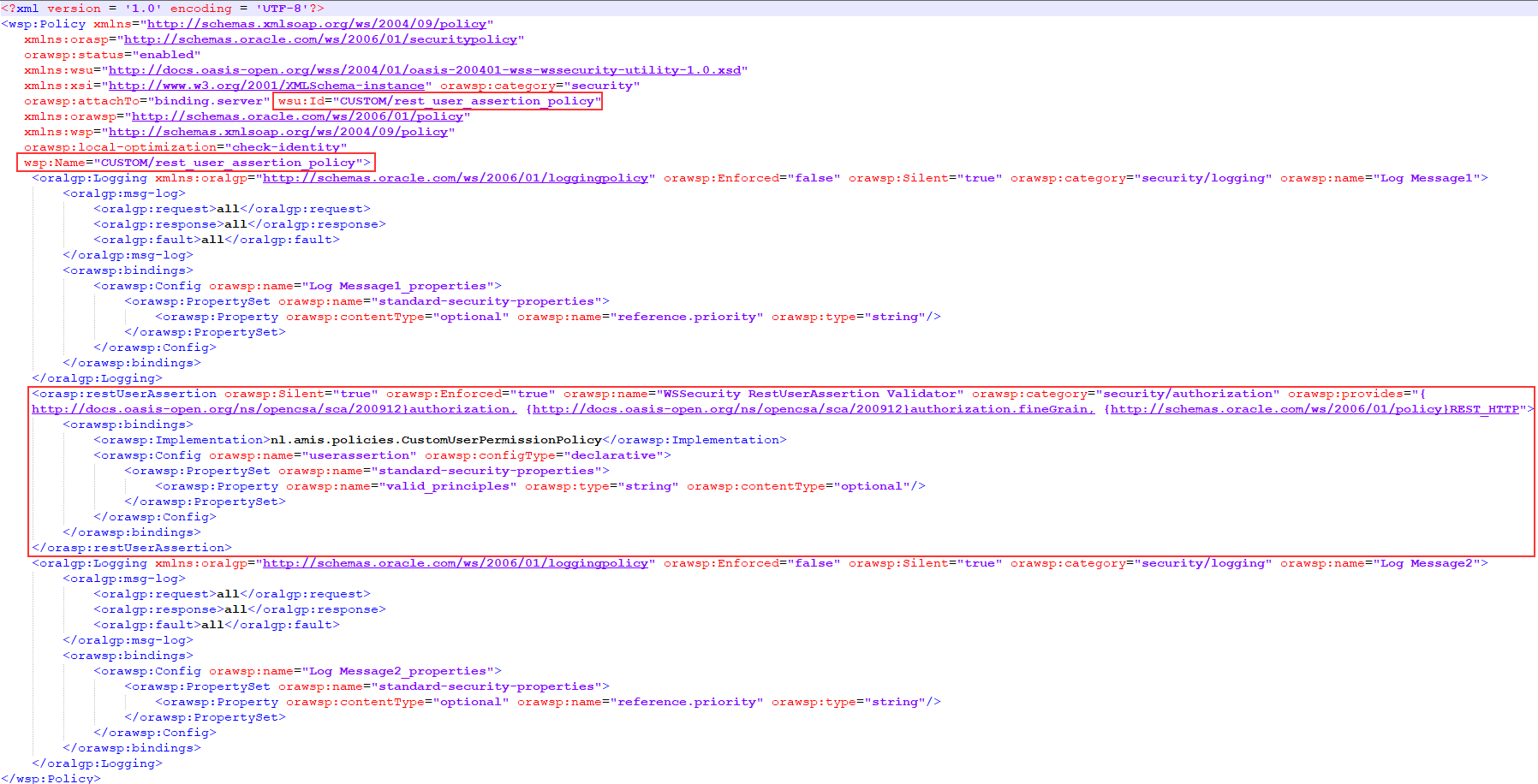
* This contains a policy description file.   
  Which parameters can be set for the policy?   
  Of which type are the parameters?   
  What are the default values of the parameters?  
  Is it an authentication or authorization policy?  
  Which bindings are supported by the policy?

The policy description file contains an element which maps to the entry in the policy-config.xml file. Also the ZIP file has a structure which is in line with the name and Id of the policy. It is like;

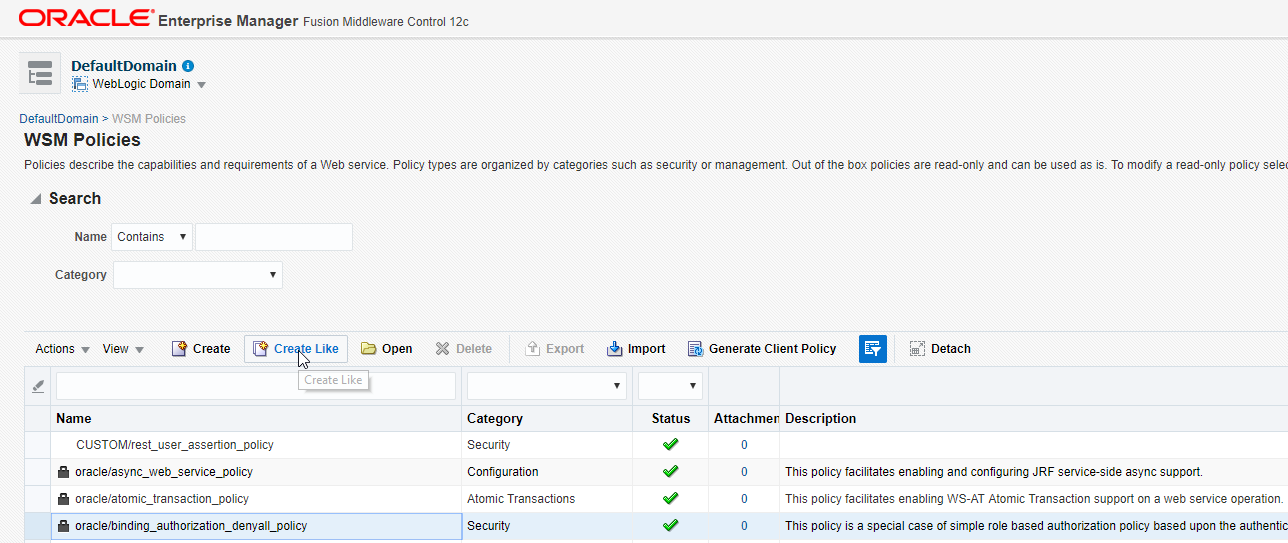


Thus the name of the policy is CUSTOM/rest\_user\_assertion\_policy

This name is also part of the contents of the rest\_user\_assertion\_policy file. You can also see there is again a reference to the implementation class and the restUserAssertion element which is in the policy-config.xml file is also there. The capabilities of the policy are mentioned in the restUserAssertion attributes.



Suppose you want to expand the capabilities of the policy, you can use the WSM Policies screen to create a policy like one of the standard policies which has the capability you want.



Next you can export that newly created policy. It is not possible to Export the predefined policies as you can see in the above screenshot. This only works for policies in the Security category. This will give you the policy description file which contains the policies capabilities. You can then copy these (attributes) into your own custom policy.