Open Stack Federated Middleware Services

# Versioning

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Comment | Author |
| 0.1 | 10/07/2012 | Initial draft | Matteo |
| 0.2 | 10/07/2012 | Minor changes | Kristy |
| 0.3 | 18 July 2012 | Minor edits | David |
| 0.4 | 18 July 2012 | Major changes | Matteo |
| 0.5 | 19 July 2012 | New changes in the design | Matteo |
| 0.6 | 19 July 2012 | Checked and updated schema in appendix | Kristy |
| 0.7 | 20 July | Minor | David |
| 0.8 | 23/07/2012 | Minor changes in messages | Damien, Yann |
| 0.9 | 24 July 2012 | Minor edits | Kristy |
| 0.91 | 24 July 2012 | Updated sequence diagram and minor changes | Matteo |
| 0.92 | 25 July 2012 | Change in the config file | Matteo |
| 0.93 | 27 July 2012 | Removed inconsistencies and updated the schema | David |
| 0.94 | 31 July 2012 | Minor edits and updated schema to include SpName for request building | Kristy |
| 0.95 | 1 August 2012 | Updated sequence diagram | Kristy |
| 1.0 | 27 August 2012 | First public release | Kristy |

# Introduction

According to (1), the Federated Middleware Services (FMS) are in charge of preparing the request to the IdP for the client, validating the IdP’s assertion sent back by the client, using the PERMIS CVS, and managing the users and the tenants for the federated access.

Using the validated attributes and according to the configuration file, FMS retrieves the available tenants from Keystone. If the permitted tenants are not in keystone they will be created and linked to the user. If the user has provided a tenant (tenant friendly name), and it is in this list, then a scoped token is returned to the client along with the list of available services endpoints, otherwise the list of tenants is sent to the client as well as an “unscoped” token in order for the user to select the desired one.

Keystone receives the “unscoped” token and the selected tenant back from the client and it validates the “unscoped” token in order to return the corresponding “scoped” token for the tenant. Finally the scoped token is sent to the client along with the list of available services endpoints.

# Configuration file

In the configuration file the necessary information for the middleware is specified such as the key and the certificate to sign the request to the IdP. It also defines the attributes that are needed to access the system.

The attributes are divided into sets and each set corresponds to a tenant. The attributes in the set can be specified either as a type or type and value[[1]](#footnote-1).

The following code represents an example of a configuration file.

*<Config>*

*<CertificateFile>cvmiddleware.crt</CertificateFile>*

*<KeyFile>cvmmiddleware.key</KeyFile>*

*<KeyPassword>password</KeyPassword>*

*<SpName>KeystoneClientSecure</SpName>*

*<IdPpid IdP="secrete" PID="eduPersonTargettedID" />*

*<SetOfAccounts>*

*<Account DisplayName="Kent Personal Account">*

*<!-- Kent personal accounts -->*

*<Attribute Name="idp” Value=”kent.ac.uk" FN="Account" />*

*<Attribute Name="uid" FN="ID" />*

*</Account>*

*<Account DisplayName="Facebook Personal Account">*

*<!-- Facebook personal accounts -->*

*<Attribute Name="idp" FN="Account" />*

*<Attribute Name="uidFB" FN="ID" />*

*<MappedAttribute>*

*<UserAttribute Name=”role” Value=”staff” />*

*<RoleGranted>admin</RoleGranted>*

*</MappedAttribute>*

*</Account>*

*</SetOfAccounts>*

*</Config>*

A special statement (IdPpid) is used to indicate the identifier attribute an IdP issues to uniquely identify each user. Reading this statement the middleware can know which attribute to use to create the user uniquely in Keystone.

The configuration file is written in XML so that it can be easily configured by the administrator and easily parsed by the FMS. The Schema for the configuration file is presented in Appendix 1.

# Implementation

The FMS will be implemented in the different python classes:

* *XML Config Handler*: it parses the configuration file and creates an attribute configuration object. It also provides methods to retrieve information from the configuration object.
* *Middleware wrapper (auth\_federated)*: this class is the main class of the middleware. It wraps around the discovery, the validation and the engine. It is in charge of catching the requests and routing them to the respective handler.
* *Discovery Handler:* provides the methods to communicate with the DISCO service. It maps an IdP realm into an IdP endpoint
* *Request Handler*: It creates the IdP request for the client.
* *Validator Handler:* it catches the IdP response sent by the client and validates it to extract the valid attributes. This class uses the Validation interface to validate the IdP’s response.
* *Validation Interface:* provides the methods to validate the IdP’s response.
* *Disco Interface:* provides the methods to communicate with the discovery service.
* *Engine*: it is the core of the new middleware. It receives the validated attributes from the Validator and it interacts with the Keystone (SK) engine to create tenants and users.

The following figure graphically describes the relations between the different parts of the new middleware.

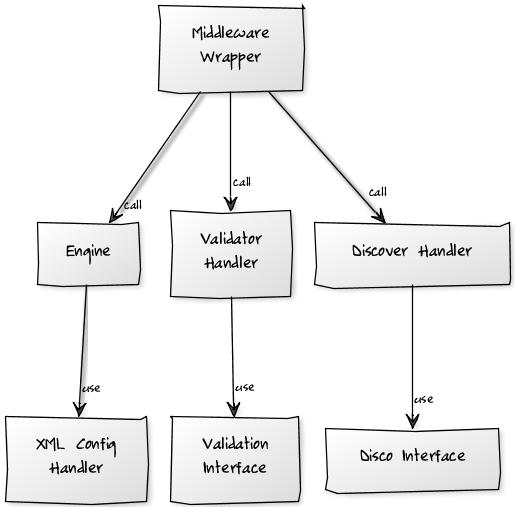
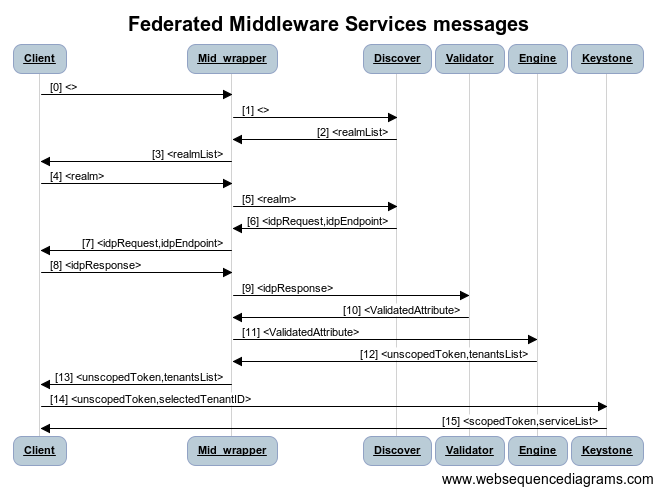


Figure 2 Middleware class diagram

The FMSs respect the steps described in the following sequence diagram.

Figure 3 FMS sequence diagram

As we can see from the previous image, the requests the FMS expects and the responses it sends are described in the following table. Particularly, these are the messages to and from the Middleware Wrapper.

* [0] <>: the client sends an empty request body in order to retrieve a list of available realms (IdPs)

{}

* [4] <realm> : the client sends a realm in order to retrieve the request for user authentication with the IDP.

{‘realm’ : ‘exampleOfRealm’}

* [7] <idpRequest, idpEndpoint> : the middleware returns the Request required for the authentication.

{ ‘idpRequest’ : ‘exampleOfRequest’, ‘idpEndpoint’: ’linktotheIDP’}

* [8] <idpReponse> : the client sends the Response returned from the IDP.

{ ‘idpResponse’ : ‘exampleOfResponse’ }

* [13] <unscopedToken, tenantsList> : the middleware return the brand-new unscoped Token and the list of available tenants.

{ ‘unscopedToken’ : ‘exampleOfToken’,

‘tenants’: [{‘name’: ‘tenantName1’ , ‘id’: ‘tenantID1’, ‘friendlyName’: ‘tenantFN1’}, {‘name’: ‘tenantName2’ , ‘id’: ‘tenantID2’, ‘friendlyName’: ‘tenantFN2’}]}[[2]](#footnote-2)

* [14] <unscopedToken, tenantId> : the client sends to keystone the unscoped token and the selected tenant\_ id in order to get the scoped token.
* [15] <scopedToken, services>: Keystone returns the scoped token and a list of services at which it can be used

The user can also provide a tenant friendly-name from the beginning, because he may already know which tenant he wishes to access. In this case, the client will use the matching ID of the tenant specified as tenantId in [10].

### Engine implementation

The engine is in charge of managing the users and the tenants until their unscoped token has been issued. It expects to receive a set of valid attributes. It will act in the following way:

1. Using the attributes configured in the configuration file it identifies the user uniquely and it checks if the user exists in KS otherwise it creates a new user.
2. It reads the configuration file to retrieve the permitted accounts. They are identified by a set of attributes that the user has to own in order to access that account. A tenant in KS is identified by the concatenation of these attributes.
3. It checks if each permitted account (each permitted tenant) already exist, otherwise it creates a new one.
4. It checks if the user and the tenant are linked otherwise it links them together.
5. The middleware creates an unscoped token and returns it to the client as well as the list of available tenants. These will then be used by the client to get the scoped token directly using the KS API with its existing swapToken method call.

# References

(1) Chadwick, David, Damien Germonville, and Yann Fouillat. "Adding federated access to OpenStack." 2012.

## Appendix 1 Schema for configuration file

<?xml version="1.0" encoding="ISO-8859-1" ?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<!-- Simple Types -->

<xs:element name="RoleGranted" type="xs:string" />

<xs:element name="CertificateFile" type="xs:string"/>

<xs:element name="KeyFile" type="xs:string"/>

<xs:element name="KeyPassword" type="xs:string"/>

<xs:element name="SpName" type="xs:string"/>

<!-- Attributes -->

<xs:attribute name="DisplayName" type="xs:string" />

<xs:attribute name="Name" type="xs:string" />

<xs:attribute name="IdP" type="xs:string" />

<xs:attribute name="FN" type="xs:string" />

<xs:attribute name="PID" type="xs:string" />

<xs:attribute name="Value" type="xs:string" />

<!-- Complex Types -->

<!—Identifies the attribute used by an Identity Provider as a unique Persistent Identifier for each user. If the IdP attribute is missing then the PID is the default PID that applies to all unspecified IdPs -->

<xs:element name="IdPpid">

<xs:complexType>

<xs:attribute ref="PID" />

<xs:attribute ref="IdP" use=”optional” />

</xs:complexType>

</xs:element>

<!-- Element representing one attribute with a type (Name), an optional value, and a friendly identifier (FN)-->

<xs:element name="Attribute">

<xs:complexType>

<xs:attribute ref="Name" />

<xs:attribute ref="FN" />

<xs:attribute ref="Value" use=”optional” />

</xs:complexType>

</xs:element>

<!-- Element representing an optional set of attributes that grant a role on Openstack services

UserAttribute - specifies zero to many user attributes that are required in order for the user to be granted the role. If zero attributes are specified this means that the role is granted to all authenticated users

RoleGranted - the Openstack role that is granted if the user possesses all the UserAttribute(s) -->

<xs:element name="MappedAttribute">

<xs:complexType>

<xs:sequence>

<xs:element ref="UserAttribute" minOccurs = “0” maxOccurs="unbounded" />

<xs:element ref="RoleGranted" minOccurs = “1” maxOccurs="1" />

</xs:sequence>

</xs:complexType>

</xs:element>

<!-- Element representing a user attribute. It comprises a Name and an optional value -->

<xs:element name="UserAttribute">

<xs:complexType>

<xs:attribute ref="Name" />

<xs:attribute ref=”Value” use=”optional” />

</xs:complexType>

</xs:element>

<!-- Element representing a set of attributes required for access to an account

DisplayName - friendly account identifier -->

<xs:element name="Account">

<xs:complexType>

<xs:sequence>

<xs:element ref="Attribute" maxOccurs="unbounded" />

<xs:element ref="MappedAttribute" maxOccurs="unbounded" />

</xs:sequence>

<xs:attribute ref="DisplayName" use="required" />

</xs:complexType>

</xs:element>

<!-- Element representing a set of accounts -->

<xs:element name="SetofAccounts">

<xs:complexType>

<xs:sequence>

<xs:element ref="Account" maxOccurs="unbounded" />

</xs:sequence>

</xs:complexType>

</xs:element>

<!-- Element representing the configuration element for the Federated Services Middleware

CertificateFile / KeyFile - the private key/certificate pair used for signing the request to the IdP -->

<xs:element name="Config">

<xs:complexType>

<xs:sequence>

<xs:element ref="CertificateFile" maxOccurs="1" />

<xs:element ref="KeyFile" maxOccurs="1" />

<xs:element ref="KeyPassword" maxOccurs="1" />

<xs:element ref="SpName" maxOccurs="1" />

<xs:element ref="IdPpid" minOccurs="1" maxOccurs="unbounded" />

<xs:element ref="SetofAccounts" maxOccurs="1" />

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:schema>

1. The combination of type and value defines a fixed account and so a fixed tenant. [↑](#footnote-ref-1)
2. In this example, the square brackets stand for an array definition. In the specific case, it defines a list of tenants. [↑](#footnote-ref-2)