CLARIN / BiG Grid Development Report

User Delegation in the CLARIN Metadata Infrastructure: connecting the component registry and ISO-DCR

Part II - Implementation

authors Willem van Engen & Mischa Sallé

4 July 2013

Table of Contents

List of Figures

1. Introduction

The European project CLARIN[1] is developing a research infrastructure for e-Humanities based on a service oriented architecture. The vision of CLARIN is to have a collection of web services, accessible to the user via a number of (web) portals. Web service owners may want to restrict usage based on who is using it, and web services may need to call other web services, which can be protected as well (see Figure 1). This means an authentication and authorisation infrastructure (AAI) is needed that supports this.

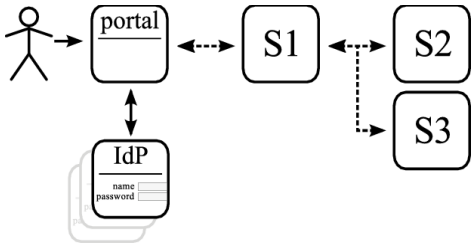

Figure 1: An example service invocation. After authenticating with an identity provider (IdP), the user invokes service S1 from within a portal. In turn S1 invokes S2 and S3 with the user's permissions.

The design of a security infrastructure has been the topic of the workshop "Security for web services"[2] organised by BiG Grid[3] in April 2010. The next step was to consider a simple but practical use-case to gain some experience. This was found in a CLARIN web application (component metadata registry/editor[4]), which needs to access private data present in another web application (the ISOcat data category registry[5]). BiG Grid has investigated a number of technologies that would enable this[6], leaving a setup based on OAuth 2.0 and X.509 certificates as promising options (see the full report[6] for details).

CLARIN has indicated a preference for an OAuth 2.0 based solution, which therefore will be tried first, an X.509 certificates based solution will only be investigated in case OAuth2.0 turns out not to satisfactory solve the use-case.

2. OAuth 2.0

OAuth[7] is an open protocol for secure API authorisation, used widely on the world wide web. Version two of the standard[8] provides the flexibility needed for our use-case.

When an OAuth client tries to access a service (resource) which is protected by OAuth, it needs to be authorised by the resource owner. Proof of authorisation is an OAuth access token. In OAuth2.0 this token is provided by an Authorization Service (AS). The standard specifies various ways (OAuth 2 flows) in which an access token can be obtained[9], depending on the use-case. In order for the AS to provide the client with an access token, the user (resource owner) needs to authenticate and grant permission, i.e. authorise the client.

Note that the authorisation being discussed here concerns the *user* authorising the *client* to access its data at the resource, while the authorisation decision made by the resource server itself is outside the scope of OAuth.

---

1 http://www.clarin.eu/
2 http://agenda.nikhef.nl/conferenceDisplay.py?confId=993
3 The Dutch national grid initiative, http://www.biggrid.nl/
4 http://www.clarin.eu/cmdi
5 http://www.isocat.org/
6 http://www.nikhef.nl/pub/projects/grid/gridwiki/images/6/66/Clarin-security_for_web_services-research-report010.pdf
7 http://oauth.net/
8 http://tools.ietf.org/html/rfc6749, already used in production for example by Facebook, GitHub and Google.
9 These are called "OAuth flows"; http://www.independentid.com/2011/03/oauth-flows-extended.html

The typical flow is as depicted in Figure 2.[10]

1. For the purpose of this document, we assume the *user* is accessing a web portal, called the *client* in OAuth terminology, using a web browser. The general OAuth specification allows for other clients too.
2. The client (portal) needs access to another web service (*resource* in OAuth terminology).
3. For this it needs to obtain authorisation from the resource owner, i.e. the user, and it initiates an OAuth 2.0 flow. The client can get the required authorisation directly from the user or via the *Authorization Server* (AS).
4. In the website context the latter could be a redirect to an *Authorization Endpoint* (AE) on the AS. Such an AE is an endpoint where the user authenticates (either directly or via a redirect to an IdP as in the Figure) and then gives its authorization. Note that because of the dual role it is also sometimes called *Authentication Endpoint*. As already noted above, authorisation concerns the user authorising the client.

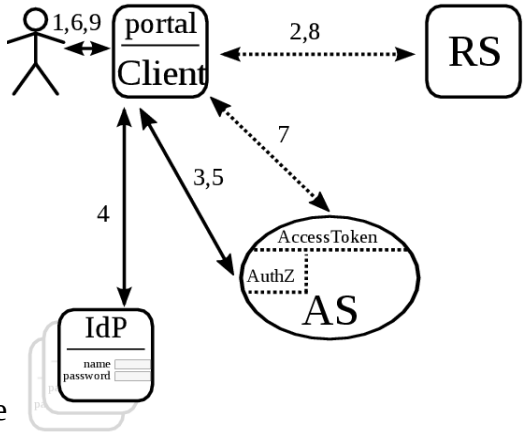

Figure 2: Typical OAuth2.0 flow, where the user obtains a authorisation code at an authorisation endpoint.

5. The client then receives...
6. ...via the user (which is redirected back) an *authorisation grant* (code) ...
7. ...which it can use to obtain an *access token* directly from the AS (the token endpoint).
8. This token is then used by the client to obtain the restricted contents from the resource server.

   The server then should check the token (how this should be done is left open in the OAuth2.0 specification) and returns the restricted data to the portal...
9. ...which returns everything to the user

In summary, there are three different stages in the OAuth2.0 flow: 1) the user authenticating and authorising the client at the AS, resulting in an *authorisation grant*. 2) the client using the *authorisation grant* to obtain an *access token*. 3) the client using the *access token* at the resource server to obtain the restricted data. Stages 1) and 2) will be discussed in the next two Sections.

2.1. Authorisation grants

Different types of authorisation grants have been defined but the specification leaves room for adding additional ones. The most common and preferred grant is using an explicit *authorisation code* obtained from an AE, this is the scenario sketched in Figure 2. Facebook[11], GitHub[12] and Google's web service profile[13] all support using such an approach.

Another is the *implicit code* where the AS directly returns an access token without an intermediate grant. The scenario is similar to OpenID Connect implicit Client profile[14], supported e.g. by

10 We use drawn lines for browser-based connections (incl redirections), dotted lines for server-server connections.
11 http://developers.facebook.com/docs/authentication/
12 http://developer.github.com/v3/oauth/
13 https://developers.google.com/accounts/docs/OAuth2#webserver
14 http://openid.net/specs/openid-connect-implicit-1_0.html

Google[15].

Effectively this implicit code flow is the setup as sketched in Figure 3. It is aimed at clients which are inside the user's web browser (e.g. in JavaScript) and makes no clear distinction between user and client. Hence in the typical flow the AS does not separately authenticate the client, since the user would have access to the client credentials, but relies on the user authenticating and granting authorisation, and then immediately returns an access-token (although it does send a client identifier and optionally a redirect URI to the AS, which could be used to trace the client). The flow has the advantage of being simpler, but has the disadvantage that the access token is directly reachable for the user (browser) and is more prone to attacks by malicious clients.

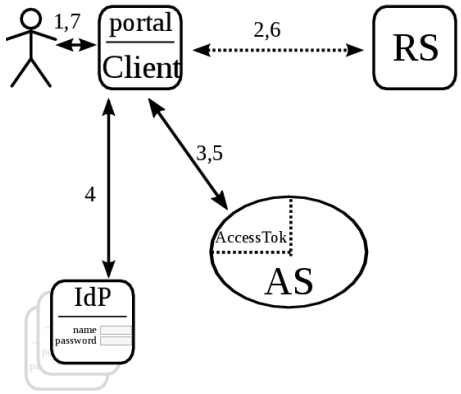

Figure 3: Effective OAuth2.0 flow with the implicit code.

Another interesting authorisation grant is the *SAML code*, where additional information can be passed in the authorisation flow.[16] It can be used either to transport the granting of authorisation by the user to the AS and/or authenticating the client.

These three different types of authorisation grants will be discussed in more detail, in the context of the CLARIN use-case, in Section 3.2.

The OAuth 2.0 specification leaves it open to further define custom ways to obtain an access token, but not conforming to standards makes it more difficult to build a modular setup where individual components can be replaced.

2.2. Access tokens

The standard only describes simple *bearer access tokens*[17]: resource servers need to retrieve any metadata, including user identity, from a special endpoint on the AS. Nevertheless, the standard does not to exclude the use of structured tokens and the authors of the internet-draft on JSON Web Tokens[16] seem to have in mind to also use them as access tokens.

Structured tokens would provide a solution to several problems, especially when using multiple ASes:

- Each service may need to know all ASes (to verify a token).
- Each service may need to be known to all ASes (for client authentication).

On the other hand, structured tokens also have a number of disadvantages such as

- Interoperability / interchangeability: when the token-consuming service needs to use the information inside the token, the consumer and AS need to agree on the format. An opaque token is typically sent to the same AS producing the ticket.
- Security: the information in the tokens must at least be signed in order to prevent the client from changing its content. It might also be necessary to encrypt it (for the resource/token consumer) which would mean the AS needs the public key/certificate of the token consumer.

15 http://code.google.com/apis/accounts/docs/OAuth2.html#clientside

16 http://tools.ietf.org/html/draft-ietf-oauth-saml2-bearer. A similar grant is based on JSON Web Tokens, see http://tools.ietf.org/html/draft-ietf-oauth-json-web-token

17 http://tools.ietf.org/html/rfc6750. Work is also ongoing for the use of Message Authentication Code tokens, see http://tools.ietf.org/html/draft-ietf-oauth-v2-http-mac

An important open question is how to verify the access token and obtain necessary information about the user, the client etc. This is something the OAuth2.0 specification does not define.
Is that information present in the token itself (structured tokens), in extra parameters in the OAuth request, or is it returned by a (REST) call to a certain endpoint? Where are these endpoints located? Who controls them?

A common solution is to define a *check token* or *userinfo* endpoint on the AS, where the resource service can verify the token and optionally obtain information about the user. Similarly, a so called *introspection endpoint*, which could be integrated with the check token endpoint, could also provide information about the client who requested the token. This can prevent a number of attack scenarios and is implemented by e.g. FaceBook.

Another solution is proposed by OpenID connect, by making use of the multiple-valued response type[18]: instead of returning only an access token, the AS can additionally return an *id token* which can then be used to obtain extra information, this approach is used by e.g. Google.

3. OAuth 2.0 scenarios in the CLARIN use-case

It is clear that OAuth 2.0 still gives a lot of flexibility in its use. Hence a number of solutions for the CLARIN situation is discussed here.

3.1. Role of the AS

The first choice is whether it is preferable and/or feasible to have only a single AS. Having multiple ASes could be an advantage: not having a central service is more in line with the general CLARIN setup, and could allow for more fine grained authorisation rules. On the other hand it is also a lot more complicated to administer as services should trust and keep track or discover multiple ASes, the ASes themselves should possibly communicate grants etc. There are no standards for this known to us. A single, high-availability endpoint is probably easier to implement and maintain while still providing the necessary features.

Another choice to make concerns the mechanism used for obtaining (extra) user information. As discussed above in Section 2.2, the most common solutions seems to be either the implementation of a *check token* endpoint on the AS or the use of an *id_token* in addition to the *access_toke*n. Which option to use depends on the choice of AS and client implementation.

3.2. Position of the IdP

Within the context of OAuth2.0 the position where the user authenticates, usually via an Identity Provider (IdP), is another point of choice and defines the type of authorisation grant being used. There are roughly three possible scenarios of integrating the IdP, which effectively correspond to the different grants discussed in Section 2.1, authorisation grant, implicit grant and SAML assertions. Although all three have already been discussed briefly, we will here discuss the flow, in the context of the CLARIN use-case, in more detail.

3.2.1. Authorization Service authenticates at IdP

In the *implicit grant* flow, the user would be redirected by the portal directly to the token endpoint

18 http://tools.ietf.org/html/rfc6749#section-8.4 and http://openid.net/specs/oauth-v2-multiple-response-types-1_0.html. Note that OpenID also makes use of a userinfo endpoint, as specified in the standard http://openid.net/specs/openid-connect-standard-1_0.html

of the AS and the access token would be returned through the browser to the portal via a HTTP redirect. Authentication of the user would typically be an additional browser redirect from the AS to the IdP. In this approach there would normally be a single, centrally managed Authorization Service.

The flow is as follows, see Figure 3:

1. User accesses portal
2. Portal redirects to AS *(providing client authentication + scope)*
3. AS redirects to IdP *(providing a SAML request)*
4. IdP authenticates user
5. IdP redirects back to AS *(returning a SAML assertion for the AS[19])*
6. AS asks user for confirmation, including the requested scopes
7. AS redirects back to portal *(returning an access token)*
8. Portal sends request to service *(providing access token)*
9. Service verifies access token & obtains user information:
    - If opaque access token: request info from userinfo endpoint *(providing access token)*
    - If structured access token: verify&parse access token
10. Service verifies authorisation (including scope)
11. Service returns result to portal

3.2.2. Authorization Endpoint authenticates at the IdP

This setup uses the *authorization grant* flow. Decoupling the authorisation and authentication steps from obtaining the token leads to greater flexibility. The two endpoints can physically be separated (e.g. used by FaceBook), while the token endpoint of the Authorization Service only needs to trust the Authorization Endpoint, instead of all clients separately. This makes it possible to have multiple Authorization Services, which could allow owners of a group of services to use their own. Furthermore it keeps the actual access tokens outside of web browser as discussed above.

In this case the flow is as follows, see Figure 2:

1. User accesses portal
2. Portal redirects to AE *(opt. providing client authentication + scope)*
3. AE redirects to IdP *(providing SAML request)*
4. IdP authenticates user
5. IdP redirects back to AE *(returning SAML assertion)*
6. AE obtains user consent (for scopes)

19 There are different profiles defined in SAML for getting attributes from the IdP to the SP. The SAML2 HTTP POST binding returns them in the response through the browser, the HTTP Artifact binding uses a back channel, see e.g. http://en.wikipedia.org/wiki/SAML_2.0#SAML_2.0_Bindings and https://www.oasis-open.org/standards#samlv2.0. Shibboleth 2 by default uses the attribute push where the SAML goes through the user 's browser instead of via a back channel, Shibboleth 1.3 used SAML 1.1 which cannot encrypt assertions, making it unsafe for attribute push. See e.g. https://www.switch.ch/aai/support/metadata/saml1-attribute-push.html. SimpleSAMLphp when using SAML2 also supports both bindings. See e.g. http://simplesamlphp.org/docs/stable/simplesamlphp-ukaccess#section_6 and http://simplesamlphp.org/docs/1.6/simplesamlphp-artifact-sp

7. AE redirects back to portal *(providing code or optionally JWT assertion)*
8. Optional: Portal obtains user information by parsing structured code/assertion
9. Portal requests access token from AS *(providing client authn + code/assertion + scope)*
10. AS verifies code/assertion
11. AS responds to portal *(returning access token)*
12. Portal sends request to service *(providing access token)*
13. Service verifies access token & obtains user information:
    1. If opaque access token: request info from userinfo endpoint *(providing access token)*
    2. If structured access token: verify&parse access token
14. Service verifies authorization (including scope)
15. Service returns result to portal

3.2.3. Client/portal authenticates at IdP (SAML assertion pass-through)

The portal could act as a SAML service provider. If the portal can obtain a signed SAML assertion from the IdP it could use this as authorisation grant along the lines of the OAuth SAML bearer assertion draft[16]. This draft specifically mentions (in section 3) among other the following requirements:

1. The AS MUST check the validity of the assertion, including the signature
   - *This means that the SAML assertion must be signed.*
   - *This means that each AS must be able to parse and verify SAML.*
   - *This means that each AS must have a list of all IdP metadata (needed in any case).*
2. The SAML assertion MUST contain an audience restriction identifying either the AS directly (it MAY be its token url), or the authorisation endpoint (AE) SAML entity.
   - *Audience = AS: could be ok when each IdP includes the AS in the assertion.*
     - *This would either limit a session to a single AS, or each IdP should include all available ASes.*
     - *SAML2 assertions support specifying multiple audiences[20], where the relying party should be member of one or more of the specified audience: hence each AS could be a valid audience.*
   - *Audience = AE: one could, perhaps, see the portal as its own authorisation endpoint, so each portal should be known by all ASes to verify the audience. This effectively reduces it to the implicit code flow and seems to be quite a stretch from the the now published OAuth 2.0 specification.*

   In either case, it will still be necessary to have some metadata exchange between IdPs and AS, which would appear to be the main motivation of having the portal, instead of the AS, act as SAML SP.

The requirements further imply:

20 See paragraph 2.5.1.4 in the specification http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf. Also see the errata on the same section.

- Each portal is a SAML service provider
  - *The portal needs to be able to obtain the raw assertion; SimpleSAMLphp cannot*[21]*, mod_shib can*[22].
  - *Each portal must have a list of all IdP metadata.*
  - *Each portal must be known to all IdPs, which can be taken care of by means of a/the CLARIN service federation.*
- The identity providers must allow that the assertion is passed through to other services within the CLARIN domain.

The flow is as follows:

1. User accesses portal
2. Portal redirects to IdP *(providing SAML request)*
3. IdP authenticates user
4. IdP redirects back to portal *(returning SAML assertion)*
5. Portal obtains user information from SAML assertion *(perhaps not needed: pass-through)*
6. Portal requests access token from AS *(providing client authn + SAML assertion + scope)*
7. AS verifies assertion
8. AS verifies authorisation
9. AS returns access token to portal *(returning access token)*
10. Portal sends request to service *(providing access token)*
11. Service verifies access token & obtains user information:
    1. If opaque access token: request info from userinfo endpoint *(providing access token)*
    2. If structured access token: verify&parse access token
12. Service verifies authorisation (including scope)
13. Service returns result to portal

In this flow, it is the portal's responsibility to request user consent for certain services and scopes. When this is not desired, it could be moved to the AS. In that case the portal needs to redirect the user to the AS, which then asks for consent, before it hands out an access token.

4. Available software implementations

For each of the three components in the use-case, client/portal, resource server and AS, we will shortly go over the different options. The number of implementations of OAuth2.0 client and server software is increasing rapidly hence it is difficult to give an exhaustive list of available solutions.

4.1. Client

There are already many existing OAuth2 client implementations for various programming

21 The assertion is not stored; http://groups.google.com/group/simplesamlphp/msg/45046408d1667ebd
22 https://wiki.shibboleth.net/confluence/display/SHIB2/NativeSPAssertionExport

languages. Good documentation of the settings needed is very important. It would also be useful to have a number of examples in the most important programming languages (at least Java and PHP). Finally, a way to configure a client using only an Apache configuration, without touching the portal code, would be a bonus. User and token information would then be supplied via environment variables.

4.1.1. Use-case: CMDI Component Registry

The CMDI Component Registry is Java-based (Tomcat servlet) using mod_jk[23]. It uses servlet security (SHHAA[24]), with mod_shib[25] for single sign-on. The servlet filters extract a user identity from headers, server variables etc. and put this into the REMOTE_USER variable (for Shibboleth usually the ePPN or ePTID).

Available options:

- *client library:* a Java client library to integrate within the portal for obtaining a token
    - access to portal needs to be web-based, because obtaining a token requires the user's web browser
- *reverse proxy/servlet* that always requests an access token after logging into the portal.
    - pro: portal doesn't need to change much
    - con: token not always needed

The former is cleaner, since it only fetches a token when it is needed. It is probably also easier to implement, and there is quite a number of existing client-side OAuth 2.0 libraries:

- oauth2lib's client[26]
- google-oauth-java-client[27]
- OAuth for Spring Security[28]
- Apache Amber, renamed into Apache Oltu[29]
- perhaps at some point it would be convenient to write a spring-social[30] provider.
- A potentially interesting commercial solution could be PingFederate[31]. This is outside the scope of this project.

4.2. Authorization server

The OAuth2 AS (Authorization Server) is the most complex component of the three, which handles user authentication using SAML SSO, consent, possibly some authorisation, and token management. It does not need to integrate directly with an existing application (unlike the client or RS) which makes it easier to replace one implementation with another. At the time this project started, there was not that much choice between existing OAuth2.0 AS implementations and their maturity varied. Options:

- RedIRIS's oauth2lib's Authorization Server[32]

23 http://tomcat.apache.org/connectors-doc-archive/jk2/jk/aphowto.html
24 https://aai2.rzg.mpg.de/secure/shhaa/site/
25 https://wiki.shibboleth.net/confluence/display/SHIB2/NativeSPApacheConfig
26 http://www.rediris.es/oauth2/
27 http://code.google.com/p/google-oauth-java-client/wiki/OAuth2
28 https://github.com/SpringSource/spring-security-oauth/wiki and https://github.com/SpringSource/spring-security-oauth/wiki/oAuth2
29 https://cwiki.apache.org/confluence/display/OLTU/Documentation
30 http://www.springsource.org/spring-social
31 https://www.pingidentity.com/resource-center/oauth-essentials.cfm
32 http://www.rediris.es/oauth2/ SVN codebase at http://forja.rediris.es/scm/?group_id=801

- OpenAM OAuth2 module[33]
- Songkick's OAuth2 provider[34]
    - easy to setup
    - no split AS/RS (no *check_token* endpoint)
- ndg_oauth server[35]
    - split AS/RS possible, *check_token* endpoint present
        - *check_token* only returns decision, not user identity
    - uses client-certificates for client authentication (instead of HTTP basic authentication)
- CloudFoundry User Account and Authentication Server[36]
    - based on OAuth for Spring Security
    - split AS/RS (with *check_token* endpoint)
    - seems pretty complete

Some potentially interesting newer and hence for this use-case not yet considered AS implementations include

- Apache Amber/Oltu Authorization Server[29]
- PHP OAuth 2.0 Authorization Server[37]
- OpenConext APIS OAuth2.0 Authorization Server[38]

4.3. Resource server

The resource server needs to verify the access token supplied by the client, and also needs to retrieve user information, since one of the requirements is that resources ultimately make the authorisation decision. This is nothing more than a simple REST call to a *check_token* or *user_information* endpoint on the Authorization Server.

The process should be made easy for service developers, so at a minimum clear documentation is needed on how to do this. It would be useful to have code samples in a number of programming languages.

To complete it, a way to run the service directly on Apache, with just some extra configuration for doing the check and returning information in (environment) variables, would be very beneficial. This may include running a reverse proxy that does the authorisation and passes the request onto the real service upon success.

4.3.1. Use-case: ISOcat

In our use-case, ISOcat is the resource server. While many services use servlet filters for security, ISOcat is a series of modules for Netkernel[39] with an embedded Jetty HTTP server and running behind Apache using mod_proxy[40]. Recently this has been combined with mod_shib[25] for Shibboleth single sign-on. Since it does not use servlets, ISOcat looks directly at the headers set by mod_shib to find the user information.

Available options include:

33 http://openam.forgerock.org/doc/admin-guide/index.html#oauth2-module-conf-hints
34 https://github.com/songkick/oauth2-provider
35 https://github.com/cedadev/ndg_oauth and http://ndg-security.ceda.ac.uk/browser/trunk/ndg_oauth/README.txt
36 https://github.com/cloudfoundry/uaa
37 https://github.com/fkooman/php-oauth
38 https://github.com/OpenConextApps/apis
39 http://www.netkernel.org/
40 http://httpd.apache.org/docs/2.2/mod/mod_proxy.html

- RedIRIS's oauth2lib's resource server[26]
- using a *library*
    - Apache Amber/Oltu resource service[29]
- *servlet-filter* (after migration to servlets)
    - RESThub example[41]
    - JAX-RS, Protecting resources with OAuth filters, a RESTful Java API[42]
    - OAuth for Spring Security[28]
- *reverse proxy* that checks the OAuth2 token, optionally fetches user info, and hands it over to the real application
    - mod_perl based reverse proxy with remote user authorization[43]
    - Passing REMOTE_USER from Apache as a reverse proxy to web application servers[44]
- *Apache httpd module* or *option* that does the same as the proxy directly
    - Apache module does not exist, and would be quite some work (not new idea[45])
    - if there is a config option to do authentication or set headers from a script, this may be a possibility (something like WSGIAccessScript[46], or another way of running a script plus mod_setenvif[47])

The use of a *reverse proxy* would be the most flexible. If it also integrates well with servlets (which often already use a reverse proxy), then this would be a fit solution.

5. Demonstrators

5.1. First demonstrator: OAuth2lib

One of the first seemingly complete available solutions is the OAuth2lib[26]: a PHP software package that provides all the different components for building such a security infrastructure, combined with SAML single sign-on. Hence the first attempt to solve the use-case has been based on this library.

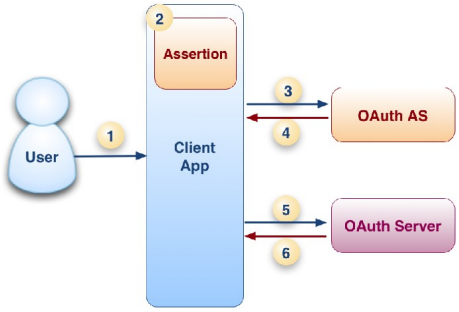

Figure 4: OAuth2lib components and their interaction.

Figure 4 shows the flow that OAuth2lib implements, it is essentially the SAML code flow as described in Section 3.2.3.

1. The user accesses the portal and logs in using a SAML identity provider (not shown),
2. ... resulting in a security assertion about who the user is.
3. The assertion is exchanged ...
4. ... for an access token at the AS,
5. which is used to (5) access the service
6. and (6)obtain the result.

As part of the implementation phase, an OAuth2lib test setup was created. There were some initial

41 http://resthub.org/refdoc/java/oauth2.html#secured-your-resource-servers
42 http://cxf.apache.org/docs/jax-rs-oauth2.html#JAX-RSOAuth2-ProtectingresourceswithOAuthfilter
43 http://blog.rot13.org/2009/06/reverse-proxy-with-remote-user-authorization.html
44 http://www.allgoodbits.org/articles/view/28
45 http://code.google.com/p/mod-auth-oauth/wiki/mod_auth_oauth
46 http://code.google.com/p/modwsgi/wiki/AccessControlMechanisms#Host_Access_Controls
47 http://httpd.apache.org/docs/2.2/howto/access.html#env

problems, but with some help of its developers and debugging, a functional test setup was realised.

5.1.1. Issues

During the installation and configuration, more details about OAuth2lib's design became clear. It appeared that the Authorization Service accepts authentication information from the client (portal) without any verification. Technically, the SAML assertion is converted to a simple web token[48] without a signature. Whatever the client provides to the Authorization Service, is accepted. This means that the portal is fully trusted by the AS, and subsequently by the service. Full client trust is not what was envisioned for our use-case.

Another issue is that all clients need to communicate with the identity providers using SAML, support of which is not present by default in most OAuth 2.0 client libraries. While this is provided by OAuth2lib for PHP-based web portals, it would be a barrier to the use of other OAuth 2.0 flows. This would affect, for example, in-browser web applications, and software written in other programming languages, see also the discussion in Section 3.2.3.

Finally it seems the development of OAuth2lib has stagnated.

5.1.2. Possible solutions

Authorization service also does authentication

Both architectural problems can be solved by moving authentication from the portal to the Authorization Service, either directly or via an Authorization Endpoint, i.e. changing to one of the scenarios in Section 3.2.1 or Section 3.2.2. When the portal needs to know who the user is, it redirects the user to the Authorization Service, which handles the SAML login, and asks for user consent to use the required services.

SAML assertion pass-through

The security issue could also be solved by passing signed SAML assertions to the AS.

This is the scenario of Section 3.2.3. When the user logs into the client, it can obtain a *signed* SAML assertion from the identity provider. When this is passed to the AS, it can check that the assertion was signed by a trusted identity provider. See Section 3.2.3 for further requirements on the use of assertions.

This solution still requires SAML support at each client. OAuth2lib uses SimpleSAMLphp, which does not provide access to the raw assertion[21].

5.1.3. OAuth2lib conclusion

Even apart from the support question, the implementation of OAuth2lib currently has problems. Considerable development work is needed to come to a clean solution. This involves moving the authentication away from the client, either directly to the Authorization Service (solution A), or to a to-be-added Authorization Endpoint (solution B). The latter allows for more flexibility and is more secure.

5.2. Second demonstrator: using NDG OAuth and Spring Security.

Based on the experiences of the first demonstrator, the needed effort to implement a secure solution

48 Explained at http://msdn.microsoft.com/en-us/library/windowsazure/gg185950.aspx

and the lack of effort being put into further development of the library, it was decided to look for alternative solutions.

From the available implementations for an Authorization Server the *ndg_oauth server*, developed by CEDA (Centre for Environmental Data Archival, STFC), seemed the most suitable and complete option. Additionally, the EUDAT project[49], in which CLARIN is a partner, will probably use it as a core component of their AAI, it can easily work behind Apache, is maintained and has most of the necessary features.

For the client (CMDI Component Registry) and resource server (ISOcat), it has been decided to use OAuth for Spring Security, as this will be relatively easy to integrate with the Java-based web services.

Installation of the demonstrator is described in the README.md on the GitHub pages[50]. We will here give some background information on the chosen implementations and some additional details not included in the README.md.

5.2.1. Authorization Server: ndg_oauth server

The *ndg_oauth* was developed at CEDA[51] with ideas of an online CA in mind (NDG stands for NERC DataGrid, which is no longer in existence, but the name is still used) and finds its origins in the MashMyData project[52]. This project looked into similar use-cases as the CLARIN use-case discussed here. This included looking at solutions similar to CILogon[53], i.e. using a MyProxy[54] based online CA. For the MashMyData project a solution was developed with a MyProxy based CA hidden behind a portal.

This was followed by the development of an OAuth2.0 based implementation (client, RS and AS), where the MyProxy CA was instead being put behind an OAuth2.0 Resource Server. After further refactoring, both service interfaces (AS and RS) have become generic WSGI filters. Secondly, for the Contrail project[55], a pure Python implementation for the online CA has been written. This online CA is typically functioning as a resource server using the *ndg_oauth server* codebase and will probably, together with the AS, also be used by the EUDAT project. However, the NDG OAuth AS is generic in the sense that it can also function standalone, without online CA.

The OAuth flow implemented by *ndg_oauth server* is the authorization code flow[56], meant for web applications/portals, Section 3.2.2/ Figure 2, with the addition of a /check_token endpoint. All endpoint urls specified are defaults at the Authorization Server and can be changed in its configuration file. The server can be run as a service using *paster*[57] but also behind Apache via *mod_wsgi*[58]. When running using Apache, *mod_shib* can be used for SAML single sign-on. When running without Apache, authentication is done using *repoze_who*[59], currently via basic authentication and a htpasswd file.

49 http://www.eudat.eu/
50 https://github.com/wvengen/oauth2-demo
51 http://www.ceda.ac.uk/
52 http://ndg-security.ceda.ac.uk/wiki/MashMyData
53 http://www.cilogon.org/, see also previous report.[6]
54 http://grid.ncsa.illinois.edu/myproxy/
55 http://contrail-project.eu/
56 http://tools.ietf.org/html/rfc6749#page-24
57 http://pythonpaste.org/script/
58 https://code.google.com/p/modwsgi/, see http://ndg-security.ceda.ac.uk/browser/trunk/ndg_oauth/README.txt#L266 and https://github.com/wvengen/oauth2-demo/blob/master/README.md#running-behind-apache
59 http://docs.repoze.org/who/2.0/

The CEDA code can be found on GitHub[60]. The version used for this use-case contains a few adaptations[61], see also Subsection 'Adaptations' below.

Running the AS involves two steps. The first part is installing the generic ndg_oauth Python library (the adapted version[61]). It can be installed with the easy_install command either using a GitHub checkout or from a prepared tar ball[62]. The second part, contained in the demonstrator GitHub, contains mostly configuration parameters such as login credentials for the client and resource, the web frontend for the AS as used in the demonstrator and the repoze_who configuration which is used for the authentication of the user.

AS Endpoints

The ndg_oauth server defines three endpoints:

- ***authorization endpoint***[63]
  The authorization endpoint is used to obtain an authorization code. The user's web browser is typically redirected to this endpoint. Following succesful authentication and user consent the user's web browser is then redirected back to the client. The ndg_oauth server uses an integrated authorisation endpoint, living on the same server.
  - *endpoint url*: `/oauth/authorize`
  - *parameters*: `response_type`, `client_id`, `redirect_uri`, `scope`, `state`
    see description in RFC6749 Section 4.1.1.
  - *authentication*: indirectly
  - *response*: HTTP GET parameters in redirect: `code`, `state`
    see description in RFC6749 Section 4.1.2.
- ***access token endpoint***[64]
  The access token endpoint is used to obtain an access code at a server-side web application, after obtaining a code from the authorization server.
  - *endpoint url*: `/oauth/access_token`
  - *parameters*: `grant_type`, `code`, `redirect_uri`, `client_id`
    see description in RFC6749 Section 4.1.3.
  - *authentication*: client secret (default) or client certificate
    defined in the appropriate ndg_oauth server configuration .ini and client_register.ini files (client secret only part of fork for this use case).
  - *response*: JSON structure with access token
    see description in RFC6749 Section 4.1.4.

- ***check token endpoint***
  When the client accesses the resource providing an access token, the resource needs to check that the token is valid. This is done by contacting the Authorization Server's check token endpoint. The user identity is returned.

  This endpoint is not specified in the OAuth 2.0 RFC. There are other implementations using this endpoint and it would be useful to to keep them interoperable where possible, like CloudFoundry's RemoteTokenServices[65].

60 https://github.com/cedadev/ndg_oauth, see also http://ndg-security.ceda.ac.uk/browser/trunk/ndg_oauth
61 https://github.com/wvengen/ndg_oauth
62 http://www.nikhef.nl/~wvengen/misc/ndg_oauth_server-0.4.0.tar.gz
63 http://tools.ietf.org/html/rfc6749#section-3.1
64 http://tools.ietf.org/html/rfc6749#section-3.2
65 https://github.com/cloudfoundry/uaa/blob/master/common/src/main/java/org/cloudfoundry/identity/uaa/oauth/Remo

- *endpoint url*: `/oauth/check_token`
- *parameters*:
  - `access_token`: the access token to check for validity
  - `scope`: scope to require for the access token **(optional)**
- *authentication*: resource secret (default) or resource certificate
  defined in the appropriate ndg_oauth server configuration .ini and resource_register.ini files (only in the fork for this use case).
- *response*: JSON structure with at least a `user_name` field.

Adaptations

The *ndg_oauth server* from CEDA has a few limitations, which have been partially lifted. A GitHub pull request is currently open[66]:

- The /check_token endpoint only returns a decision, not a user identity. Since the CLARIN use-case demands that the resource server can ultimately authorise, it needs this information. The adapted endpoint returns this.
- The client authentication is done using client-side certificates. The lack of HTTP basic auth over https makes deployment more difficult. This is adapted to also support HTTP basic auth for the client.
- Implemented resource authentication for the /check_token endpoint, to avoid brute-force attacks on token check; also provides a starting point for audience-restricted tokens and resource-restricted attribute release.

Open questions or issues:

Although the AS generally provides all functionality needed, a few improvements could be useful:

- Need to implement a resource registry
  - would be nice to specify which attributes may be returned to which resource.
- Tokens are currently not bound to a specific resource (=no audience restriction). One improvement would be an adaptation of the /check_token endpoint to additionally return scope and/or client identifier.
- The ndg_oauth server does not implement renewal tokens. It is not clear whether this is needed for the use-case.
- It seems the ndg_oauth server accepts reusing the authorisation grant. The OAuth2.0 specification does not approve of that[67].

EUDAT

One of the advantages of using the *ndg_oauth server* implementation of an AS is the potential for integration with the contrail online CA. The EUDAT version of the Authorization Server only uses different configuration files but has an unmodified code base. See further Chapter 7.

---

teTokenServices.java

66 https://github.com/cedadev/ndg_oauth/pull/1

67 http://tools.ietf.org/html/rfc6749#section-10.5

5.2.2. Client library: using Spring Security

The demonstrator client has been implemented using Spring Security and OAuth for Spring Security[28].

The demonstrator code itself is a very small but functional example. It acts as a portal where the user logs first logs in, it then obtains an OAuth2 token and uses that to 'serve' restricted content from a Resource Server back to the user. In the CLARIN use-case this corresponds with the CMDI Component Registry accessing restricted content from the ISOcat.

It is build using maven and works as a Spring Security servlet running in Tomcat. Configuration of the servlet is done using the web.xml and spring-servlet.xml files. The latter contains all the details such as the AS endpoints, client-id and secret, and also the URLs where to get the content from the resource service.

5.2.3. Resource service library: using Spring Security

The demonstrator resource service is also implemented using Spring Security and OAuth for Spring Security [28].

The demonstrator code again is a very small but functional example, using an adaptation of CloudFoundry's RemoteTokenServices[65]. It is a service that can return either public or private content depending on whether credentials are provided. In the CLARIN use-case this corresponds with the ISOcat.

Just as the client, the code is build via maven and works as a Spring Security servlet running in tomcat. Configuration of the servlet is done using the web.xml and spring-servlet.xml files. The latter contains all the details such as the AS check token endpoint and the necessary resource credentials for that.

5.2.4. Flow within the demonstrator

For clarity we describe here the detailed flow from the demonstrator, it follows the general setup from Section 3.2.2. What is described here is what is followed in the basic demonstrator setup. The names in the figure follow the actual CLARIN use-case, see Figure 5.

1. The flow starts with a user pointing a web browser to the address of the client (portal/"CMDI Component Registry").

   When the user tries to click on the secured link, the client will first request authentication for the link itself, via HTTP basic authentication.
2. Upon successful login, it will try to also 'serve' the user protected information from the Resource Server ("ISOcat"). This will trigger the need for an OAuth2.0 token.
3. If there is no token available, nor an authorisation grant, the user will be redirected to the authorisation endpoint on the AS.
4. The user needs to log in: in the most basic setup, this works using basic auth at the AS itself, in a shibbolized setup it will involve a redirect to the IdP.

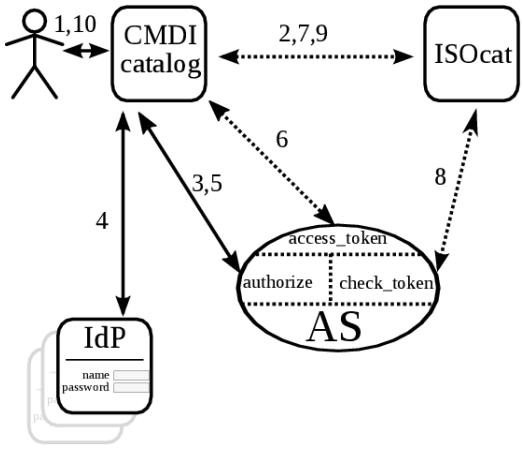

Figure 5: The demonstrator setup, with CMDI Component Registry, ISOcat and AS including its endpoints

The following few steps are hidden for the user

5. If the user logs in correctly, the AS returns via a browser-redirect an *authorization grant* to the client (portal/"CMDI Component Registry").
6. The portal obtains, using the grant, an *access token* (it hereby authenticates using its *client_id* and *client_secret* via basic auth). This goes via a direct channel between client and AS.
7. It provides this access token via the standard *Authorization* headers[68] to the Resource Server ("ISOcat").
8. The resource server checks, using its own username/password, the received token at the *check_token* endpoint.

   When valid, the AS returns a HTTP 200 with a JSON containing the user name.
9. The resource server decides whether it accepts the user name (the demonstrator server accepts all users, but this could be easily changed) and if so returns to the portal a JSON with the secured content.

Only the last step is again visible for the user

10. The portal uses the secure content to return a secured webpage to the user.

The next section will describe how this demonstrator setup was used to solve the actual CLARIN use-case.

6. Implementation by CLARIN

The second demonstrator described above in Section 5.2 was used as a basis for an implementation of the CLARIN delegation scenario on top of the actual components:

- ComponentRegistry consists of a REST service implemented in Java using Jersey[69] and depending heavily on the Spring Framework[70] as well as a graphical user interface implemented in Adobe Flex[71]. The latter runs inside the user's web browser and communicates directly with the REST service only. As explained in Section 4.1.1, the service runs inside a Tomcat instance that is protected through Shibboleth via the Apache HTTP server. The REST application contains a separate servlet that 'proxies' the ISOcat search service. This servlet is the subcomponent that will need to be extended so that it can access ISOcat's query service on behalf of the user currently logged in to the Component Registry.
- ISOcat is also a REST service, but implemented in Java using NetKernel[72]. The service runs behind an Apache HTTP server with mod_shib with lazy sessions[73].

6.1. Client: Component Registry

The ComponentRegistry REST service was already configured to use a number of the features provided by the Spring Framework, and therefore it was relatively straightforward to include OAuth for Spring Security (see Section 5.2.2). At the functional level, the following functionality needed to

68 http://tools.ietf.org/html/rfc6750#section-2.1
69 https://jersey.java.net/
70 http://www.springsource.org/spring-framework
71 http://www.adobe.com/products/flex.html
72 http://www.netkernel.org/
73 https://aai-demo.switch.ch/lazy/ and https://wiki.shibboleth.net/confluence/display/SHIB2/NativeSPProtectContent

be added to the service:

1. A method for the UI (acting as a client to the REST application) to *check* whether a token is available in the current session. If so, the ISOcat query dialog should show the text 'Connected to ISOcat'; if not, a link should be provided to the authentication endpoint allowing the user to obtain an access token.
2. A method for the UI to initiate the token request mentioned above. This should redirect the user to the Authorization Endpoint using methods available in spring-security-oauth with the option to return to the ComponentRegistry UI in the same state as it was left. The result of this 'handshake' should be reflected in the UI by updating the connection state.
3. A method for the UI to query ISOcat while passing on the access token to get access to the user's private ISOcat resources.

For these steps three servlets were added to the REST application respectively. All three made use of a shared instance of OAuth2RestTemplate[74], Spring Security's OAuth2 enabled extension of Spring's standard RestTemplate, which in turn is the class of choice in Spring for accessing a REST service over HTTP. It gets configured with the required OAuth2 parameters such as client id, secret and AS endpoints. The configuration and instantiation of this template takes place in a newly added Spring configuration file.

The first servlet, **OAuthCheckServlet** (made available at the /oauth/check endpoint) uses the OAuth2ClientContext object provided by the OAuth2RestTemplate to check whether the AccessToken property has been set. It then returns this state to the client. Note: in the initial setup the access token, when present, would be returned as a string to the web client. This is not a good idea, the web client does not need the actual token, and for security the token should be kept secret as much as possible. Hence it was decided to have it simply return a boolean indication of the presence of an access token.

The second servlet is called **OAuthHandshakeServlet** (mapped to /oauth/connect) and uses the OAuth2RestTemplate to obtain an access token. Through a single call from the servlet, all necessary redirects get carried out by the components of the spring-security-oauth2 library. In case of success, the handshake servlet redirects the web client to a success page. This in fact is an html page that, via JavaScript, activates a callback method in the Flex client application that in turn updates its state (by polling the check servlet). In case of failure a similar procedure takes place. In addition, the main error message gets passed onto the Flex application. In the initial implementation an internal frame (*iframe*) was added to the page containing the Flex application in which the response of the handshake was presented. However, it was pointed out that the use of such a frame hides the URL of the authentication end point, which makes it hard for the user to verify whether they are actually providing their credentials to a trusted party and are not, for example, being spoofed. The web client will be adapted in such a way that the authentication endpoint URL will become visible to the end user.

The third and final servlet is **OAuthIsocatServlet** (at /oauth/isocat). This servlet can be used to query ISOcat both with and without the availability of an access token. Therefore the web client can use this as a single gateway for ISOcat searches. The servlet chooses between a standard Spring RestTemplate and the OAuth2RestTemplate mentioned above depending on the presence of an access token in the session. On the same grounds it chooses between ISOcat's public search interface (/user/guest/search.dcif) or the authenticated search access point (/search.dcif). These two endpoints function identically in terms of parameters, so only the URI path depends on the OAuth2 authentication state. The presentation of the search results returned by the ISOcat servlet has not

74 http://static.springsource.org/spring-security/oauth/apidocs/index.html?org/springframework/security/oauth2/client/OAuth2RestTemplate.html

changed in comparison to the current production version except for the addition of a 'scope' column which indicates whether the data category represented by each row is public or if it originates from the user's private collection.

In order to get the OAuth2RestTemplate functioning, Spring Security had to be enabled. To prevent conflicts with the already present Shibboleth based authentication, a custom security filter had to be implemented to transfer the required aspects of the authentication state to the Spring Security models. The chosen strategy required the implementation of a custom 'pre-authentication filter' passing the session's *REMOTE_USER* as the authentication principal as well as implementing a dummy *UserDetailsService* returning no actual information since no such details are required in the present use case. All of this required some relatively complicated mapping. Despite the fact that we managed to obtain a working solution, it could be worthwhile to investigate whether this could be simplified.

6.2. Resource: ISOcat

The changes in ISOcat were relatively simple. ISOcat already offers the user the possibility to bind her local user account to a Shibboleth principal. User delegation in this case study only works for users who have done this binding. There is a fallback chain looking at different possible ways user credentials can get passed onto the application. This fallback chain was extended to look for the bearer token in the standard HTTP Authorization header: If found, the token is checked at the check_token endpoint on the AS. The AS returns user information, i.e. the principal, when the token is valid. If this principal is known, i.e. bound to a local user account, the secure content, i.e., information from the private or shared ISOcat workspace, is returned to the caller. On the other hand, if the bearer token is invalid or the principal is unknown either an unauthorized error code is returned or, depending on the context of the original, ISOcat falls back to the public workspace.

This failure process should be revisited and streamlined better. For example it would be good to return the information that the principal is not bound to any ISOcat user account in order to encourage the user to do so. Also, returning public information only, without warning the user that delegation has failed, might give a false impression that there was no matching information in the private or shared workspace, while in practice it is unknown.

7. Solving the use-case using X.509 certificates

Although the original report[6] suggested to also create a CILogon demonstrator, the results so far are successful enough to continue along the lines of OAuth2.0. Furthermore, the possibility of using the ndg_oauth server in combination with an online CA allows for a smooth transition to an X.509 certificates based solution when desired.

7.1. EUDAT

The EUDAT project seems to move towards this hybrid solution. In the context of that project, the current demonstrator has been adapted to allow a flow where the client (portal) connects to resource servers using certificates from an online CA instead of using OAuth2.0 tokens. The demonstrator for the EUDAT scenario can be found in the same GitHub repository[75].

The flow here is as follows, see also Figure 6. For clarity the names follow the CLARIN use-case. It is important to note that the current setup has the CA integrated on the same server, only providing a separate endpoint.

1. The flow starts with a user pointing a web browser to the address of the client/CMDI

75 https://github.com/wvengen/oauth2-demo/tree/eudat-master

Component Registry (portal).

2. The portal will try to obtain a client certificate by first redirecting to the AS *authorise endpoint*.
3. The AS will typically redirect to the IdP.
4. Once the user is logged in, the AS will return an *authorisation grant* via a redirect to the portal.

The following few steps are hidden for the user

5. The portal obtains, using the authorisation grant, an *access token* (it hereby authenticates using its client id and client secret via basic auth). This goes via a direct channel between client and AS.
6. The portal creates a key-pair and certificate signing request (CSR). It posts the CSR to the Online CA endpoint and presents hereby the OAuth2.0 token in an Authorization header. The Online CA functions as an OAuth2.0 resource server.

   The Online CA returns a signed certificate where the CN field contains the username as presented at login at the AS.

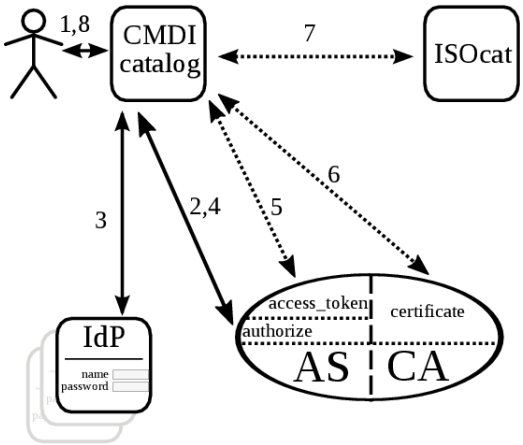

Figure 6: Demonstrator setup using EUDAT scenario/Online CA.

7. The portal uses the certificate as a client-side certificate to connect to the Resource Server.

   The resource server checks the certificate chain and makes an authorisation decision based on the username in the CN. It then returns the protected content to the portal.

Only the last step is again visible for the user

8. The portal returns the total page, including protected content.

When comparing the flow here with the pure OAuth2.0 flow as described in Section 5.2.4 one notices that the client certificates play a role which is very similar to the OAuth2.0 tokens. In some way they could be seen as signed structured tokens, containing explicitly the username and having a signature which can be checked without direct interaction with the Authorization Server. This latter feature, the absence of direct interaction between the Resource Server and the AS, could be quite a performance benefit. On the other hand, more actions need to be taken by the portal: creating a key-pair and certificate signing request plus an extra interaction with the online CA.

When creating a more generic multi-step delegation scenario, it needs to be worked out how the different services should obtain client certificates containing the right information. This might be more involved than for the pure OAuth2.0 scenario, although the latter also has not yet been worked out.

8. Conclusion and outlook

After investigating a range of OAuth2.0 implementations a first demonstrator was made, based on OAuth2lib. Getting it to work securely turned out to take up too much effort. This, combined with the lack of support and maintenance of OAuth2lib, lead to the decision to setup a second demonstrator using the ndg_oauth Authorization Server and OAuth for Spring Security. Although this second setup still needs further work to get it fully production ready, it looks like a very promising solution. For production use, a careful examination of the security requirements between the different services is needed as the simple setup with only the username as authorisation attribute

does not provide enough information to the resource servers to properly base an authorisation decision on. One of the missing features is proper audience-restriction. Another feature which might need to be implemented before moving to production is the lack of a refresh token flow.

For more extended use cases, including server to server delegation it is to be seen whether it becomes preferable to move to the EUDAT setup of the ndg_oauth server working together with an online CA which effectively uses X509 certificates instead of OAuth2.0 access tokens. When staying with OAuth2.0 tokens for these extended use cases, it will be necessary to implement scoping and audience restriction; typically the first resource server will itself need to request a token from the AS to access a second resource server. It probably should thereby present the first token in addition to proper client credentials. Even though there is discussion going on about these topics on the OAuth2.0 mailing lists, this is a relatively new area and lacking standards. At the same time, also in the EUDAT scenario it is not yet decided how to implement such delegation scenarios.

Other extensions which could become interesting are multiple-AS setups, either as fail-over or high-availability servers or as really separate servers. Even for fail-over systems, adaptations to the code will be needed.

For truly multi-AS setups all kinds of issues need to be dealt with. This includes discovery (which AS to contact), trust, synchronisation etc.. On the other hand, finding ways to deal with these type of issues would also be interesting for cross-federation contexts.

9. Acknowledgements

This work has been funded by BiGGrid (until 31 december 2012) and SURFnet.