QDL Scripts in OA4MP

Version 1.5

Table of Contents

Introduction....1
Configuration Format....2
Handlers....2
The XMD element in....2
  State....3
  Where does the QDL script go?....3
  Examples....4
    Running code directly....4
    Running scripts....4
  Loading multiple scripts for a handler....5
Accessing information in the runtime....6
  Detailed notes....8
    Access Control (access_control.)....8
    Authorization headers (auth_headers.)....8
    Claim sources (claim_sources.)....9
    Extended Attributes (xsa.)....9
    Flow State (flow_states.)....10
    Email (mail., message.)....10
    Returned Tokens (access_token., refresh_token., claims.)....11
    Initial Scope requests (audience., scopes.)....11
    Refresh/Exchange scope requests (tx_audience., tx_resource., tx_scopes.)....11
  Tip - A Note on variable visibility....12
  Error Handling....12
    New Way – using raise_error()....13
    Old Way – Set an error variable and return....13
  CILogon extension....15
General Examples....16
  Example: Checking arguments....16
  Example: Checking groups for memberships....16
  A full example....17
    Setting Extended attributes in the CLI....17

Introduction

This blurb refers to extension to QDL server scripting system that is used on OA4MP. The basic system is described in detail at QDL server scripts

Configuration Format

There is one extension to QDL's anaphors, to wit, the addition of an **`xmd`** for e***x***ecution ***m***eta***d***ata block. This contains phase information which tells the server when to execute the script. In this case, the only change to QDL's basic anaphor is here

```
BLOCK :
{
  CODE
   , XMD
   [, ARGS]

}

XMD:
  "xmd":{
    "exec_phase" : PHASES | PHASES
  }
PHASE:
   ("pre" | "post") _ ("auth" | "token" | "refresh" | "exchange" | "user_info")

PHASES:
   PHASE | [PHASE+] // single phase or array of them. Array means execute each.
```

Note that the XMD (from e**X**ecution **M**eta**D**ata) is *required* for running QDL on a server. QDL anaphors are embedded in handlers. OA4MP executes anaphors as needed.

Handlers

You should read on configuring handlers for the specifics of each handler. This article is concerned with the QDL element in that configuration (if present),the QDL code executed by it and in particular, the runtime environment for any scripts.

The XMD element. Phases.

The XMD element has the information OA4MP needs to determine when and how QDL should be invoked[1]. At this point it contains the *phases* where the anaphor is executed. These are denoted as ***`exec_phase`*** in the XMD.

Scripts are executed in one of 10 execution phases. There are pre- and post- phases for each of authorization, token, refresh, exchange and user_info. If you specify that the phase is pre_*X* then it is run before that endpoint is run (and before any system-wide claim sources are processed), allowing you to *e.g.* do some initialization. This is typically where you set a claim source(s) or do some type of

1 The name QDL was chosen for a couple of reasons. One of which is that it is used in aircraft navigation (QDL is one of several so-called Q-codes) when a plane cannot actually see where it is going (flying in dense fog, whiteout conditions, etc). so location, speed and heading are sent at regular intervals and the aircraft uses these in conjunction with its instruments to determine what course corrections it sets. The logo for QDL incidentally has the morse code for Q-D-L. Nobody is sure why Q-D-L was chosen for the Q-code, but it probably because the Morse code is easy to key. In any case, for OA4MP, the analogy is very good, with the server calling QDL which gets the flow back on course.

setup. The post_*X* phase allows you to do anything you need to right before the results are handed back to the user.

List of phases

- auth - authorization phase
- token - first token exchange, when the grant is presented and an access and refresh token are gotten.
- refresh - token refresh, i.e., for grant type “refresh_token”
- exchange - for exchanges as per RFC 8693
- user_info - for queries to the user info endpoint.
- all - all of the above

These are prepended with either **pre_** or **post_** (e.g., “pre_auth”, “post_all”) to denote if they are invoked before system processing or after system processing of that phase. If you use the unqualified **all** phase, then the script will be run every time.

Certain claims can only be gotten at certain times. For instance, claims that rely on the http headers from the identity provider are only available during the authorization phases, so these claims are gotten and stored. These can never be re-gotten until the user logs in again.

Usually the requirements for the exchange are the same as for the token phases, so the most common use pattern is to just specify exchange, refresh and access phases for a single script. Note that, as always, the current phase is set in the state, so your scripts can check.

State

When a QDL script is run on the server, its state is stored and then recovered for each subsequent call.

You have a single environment from start to finish for your scripts.

E.g., If you set something in, say, the pre_auth phase, it will be there in the post_exchange phase . See below for the table of system-managed constants. See “Accessing information in the runtime” below. These are injected into the state every time the system loads allowing you to have current state in sync with the flow. If you need something for later, store it in another variable.

Where does the QDL script go?

There are two places

1. Inside a handler. This means that it is only invoked for that handler in the specified phase(s).

2. At the top-most level in the server configuration. This is outside of the current domain of this document and is used for setting up server-wide scripts that *e.g.,* run for every client.

Understanding anaphors

The main documentation for anaphors – the JSON serialization of calls to the QDL runtime engine is intentionally simple and should be read before going any further.

Examples

Running code directly

Running a single line of code.

In this case, a token handler for identity tokens is created and a single line of code is run to assert a claim:

```
tokens{
  identity{
    type=identity
      "qdl":{
        "code":"claims.foo:='arf';",
        "xmd":{"exec_phase":["post_token"]}
      } //end QDL
    } //end identity token
}// end tokens
```

This will assert a single claim of **`foo`** has the constant value of **`arf`**. Note this runs exactly once right after the access token is created and that claim will then persists through refreshes, exchanges and such.

Running multiple lines of code.

If you wanted a more complex anaphor with several lines of code, just pass an array of lines. The anaphor in the handler would then be

```
{"qdl":{"code":[
          "x:=to_uri(claims.uid).path;",
          "claims.my_id:=x-'/server'-'/users/';"
        ],
       "xmd":{"exec_phase":"pre_token"}}}
```

(This takes a **`claims.uid`** (like `'http://cilogon.org/serverA/users/12345'`) parses it and asserts a new claim, having the value **`my_id == 'A12345'`**.

Running scripts

Loading a simple script that has no arguments

```
{"qdl":{"load":"x.qdl", "xmd":{"exec_phase":"pre_token"}}}
```

Note that if there were arguments, they would be included in the arg_list. While arguments to the script are optional (at least as far as the handler goes), some execution phase is always required so the handler knows when to run it. If you omit the execution phase, your code will never run.

Loading a script and passing it a list of arguments.

```
{"qdl":
   {
     "load":"y.qdl",
     "xmd":{"exec_phase":"pre_auth","token_type":"access"},
     "args":[4,true,{"server":"localhost","port":443}]
   }
}
```

This would create and run script like (spaces added)

```
script_load('y.qdl',
     4,true,from_json('{"server":"localhost","port":443}')
   );
```

Note that the arguments in the configuration file (which is JSON/HOCON) are respectively an integer, a boolean and a JSON object. These are faithfully converted to number, boolean and stem in the arguments the script gets.

Loading a script with a single argument

```
{"qdl": {
  "load": "y.qdl",
  "xmd":  {"exec_phase": "pre_auth","token_type": "access"},
  "args": {"port": 9443,"verbose": true,"x0": -47.5,"ssl":[3.5,true]},
 }
}
```

In this case, a script is loaded and a single argument is passed. This is converted to

```
script_load('y.qdl',
   from_json('{"port":9443,"verbose":true,"x0":-47.5,"ssl":[3.5,true]}')
  );
```

Loading multiple scripts for a handler

The handler identifies what sort of state you want exposed to the QDL scripts. Again (because it is important), the id token handler does not supply the access token and if you create on there, it will be ignored. Partly this is because in the control flow it makes no sense to be populating the access token at that point. If you have want to run multiple scripts in a single handler, they should have disjoint phases and simply be passed as an array of scripts:

```
{"tokens":
 {"access": {
 "lifetime": 1200000,
 "type": "scitoken"
 "qdl":  [
 {"load": "ga4gh/ga4gh.qdl","xmd": {"exec_phase": ["post_user_info"]}},
 {"load": "ga4gh/at.qdl","xmd":
```

```
    {"exec_phase":["post_token","post_refresh","post_exchange"]}}
 ]
}}}}
```

In this case, two scripts are run by the handler. The first is **`at.qdl`** for setting up access tokens, and the second, **`ga4gh.qdl`**, is run in the user info phase. In this case, QDL needs to know about the access token to construct various bits of new information, a GA 4 GH passport. (As to the advisability of doing it in the user info endpoint, I demur.) The point is that you don't need to drop everything in one massive QDL script and deal with phases – let the system do that. You may also have multiple scripts per phase, but that might mean you should have a driver script that calls them. There is a strong suggestion that the phases be disjoint.

State in the runtime.

Loaded modules

Usually OA4MP loads several modules for your convenience to use. These are available without any specific action on your part.

Variable	Namespace	Module Description
acl	oa2:/qdl/acl	Access control
claims	a2:/qdl/oidc/claims	Claim utilities
jwt	oa2:/qdl/jwt	JSON web token utilities

Managed Variables

When a script is invoked, the QDLRuntimeEngine will set the following in the state:

Variable	U	Component	Description	Comment
`access_control.`	**`-`**	all	ACL info	This stem contains the client id in **`client_id`** and a possibly empty list of **`admins.`** See below.
`access_token.`	**`+`**	access token	claims	The current set of claims used to create the access token (if that token requires them).
`at_original_scopes.`		access_token, refresh, exchange	original access token scopes	This is a list of the scopes that were returned in the first token exchange. It cannot be altered. The intent is that you have it for reference in scripting for checking up or down scoping.
`audience.`	**`-`**	id token	requested audience	Requested audiences in the initial request. This may impact multiple tokens, such as the id token and the access token. Again, the spec. allows this to be overloaded.

auth_headers.	-	all	auth headers	In the pre and post auth phase, the HTTP headers will be converter to this stem. In all other phases, this exists, but is empty. This allows an IDP to assert user information (e.g. SAML does this) at authorization.
claims.	+	id token	claims	The current set of user claims that will be used to create the ID token.
claims_sources.	+	id token	list of claim sources for id token	A list of claim sources that will be processed in order. If you add one, be sure it is in the right place if needed. You may add/remove as needed.
exec_phase	-	all	current phase	This is the phase the script is being invoked in. It may be the case that a script is invoked in several phases (e.g. if there is a lot of initial state to set up) and blocks of code are executed based on the current phase. Only one phase at a time is active.
flow_states.	+	all	Flow states	The various states that control execution. Generally you only need to use these if you need to change the control flow, typically, there is an access violation and you terminate the request.
mail.	-	all	The mail configuration	This is the server configuration for email which allows for QDL scripts to send notifications. See the section on email.
message	-	all	The configured message	Note that this follows QDLMail format, so the first line is the subject and the rest of the stem is the body.
oa4mp_error	-	all	1000	A reserved value for error handling. See below.
proxy_claims.	-	all	claims from the proxy	If you have enabled authorization by proxy, the list of claims with their values allowed to the client is put here
refresh_token.	+	refresh token	claims	Claims used to create the refresh token, if supported.
scopes.	-	all	requested scopes	The scopes in the initial request. This may include scopes for access tokens too since the spec. allows this to be drastically overloaded. Setting this is ignored – you cannot change the scopes the user requested (though you sure can ignore them).
sys_err.	+	all	Errors	This is a stem you set in order to have the runtime engine generate an exception outside QDL. See below, Errors

tx_audience.	-	"	TX audience	requested audience for TX. These are strings that identify the service using the token.
tx_resource.	-	"	TX resources	requested resources for TX. Similar to audience but these are URIs.
tx_scopes.	-	refresh, access token	TX scopes	requested scopes for TX
xas.	-	all	extended attributes	Extra attributes (namespace qualified) that may be sent by a client.

U = updateable.

+ = y,

- = no.

If it is not updateable, then any changes to the values are ignored by the system.

TX = Token Exchange (RFC 8693). These are sent in the request. They may or may not be sent and in that case, but they always exist inside QDL during the **pre_exchange** and **post_exchange** phases (not at other times, since they come from the request itself). You can check with a call to **size(*var*)** and if it is zero, nothing was requested.

Claims objects are always directly serialized into the token for the JWT. All of these are in the state and you simply use them. When all is done, they are unmarshalled and replace their previous values. NOTE that while your QDL workspace state is preserved, the next time it is invoked, the current values of these will be put into your workspace. i.e., what the system has is authoritative. If you need to preserve some bit of this then stash it in a variable other than one of the reserved ones.

Also, the current set of signing keys are injected into the JWT utilities and available there, so issuing a **create_jwt(arg.)** will just create a correctly signed JWT.

Detailed notes

Access Control (access_control.)

The variable **access_control.** is from QDL's ACL system and contains the current client identifier in the **client_id** entry and a list of admin ids (if there are any) in the **admins.** entry. Generally the admin list will have at most one entry, which is the one that allowed this flow in the first place. This is not alterable.

Authorization headers (auth_headers.)

Many identity providers (IDPs) send information about the user in the header when the user authenticates. Some fo these, such as the acr claim are standard, but many times they are not. A common construct is to prefix claims intended for OIDC or OAuth with a string, such as

OIDC_CLAIM_. In the QDL runtime environment almost all the claims are put into a stem called **`auth_headers.`** A few claims, such as authorization, which may contain the client password, are not returned, but otherwise the claims are normalized so the name is lower case (as per RFC 7230 § 3.2 they are case insensitive) and the values are asserted as either strings or as a list of strings. A typical example of **`auth_headers.`** is (formatted, line breaks added for readability):

```
                    accept : text/html,application/xhtml+xml,\
                             application/xml;q=0.9,image/avif,\
                             image/webp,*/*;q=0.8
           accept-encoding : gzip, deflate, br
           accept-language : en-US,en;q=0.5
                connection : keep-alive
         oidc_claim_email : bob@physics.bgsu.edu
     oidc_claim_first_name : Robert
         oidc_claim_groups : [bgsu-all, bgsu_physics, high-energy]
      oidc-claim_last_name : Smith
            sec-fetch-dest : document
            sec-fetch-mode : navigate
            sec-fetch-site : none
            sec-fetch-user : ?1
upgrade-insecure-requests : 1
               user-agent : Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:125.0)\
                            Gecko/20100101 Firefox/125.0
```

You may either process these directly as any other stem or use the http header filter claim source . If that seem to be a lot of unnecessary extra information, remember that QDL is leaving making sense of it to you, since there is no standard on what a server can pass back. As much as possible is returned. except if there is a bona fide security issue (such as cookies, passwords, &c).

A final note is that this is available only in the **`pre_`** or **`post_auth`** phase. The variable is defined in other phases, but is empty. If you want/need to save some values, stash them in another variable that is not controlled by the system.

Claim sources (claim_sources.)

OA4MP is able to use a list of claim sources (a stem of stems) to automatically get user meta data claims. This was the original system in OA4MP before QDL and is still used by a some clients. Proper usage is to set these in sequence in some init script and then OA4MP manages them after that (including getting data for the user info endpoint). The use of this is discouraged, however, since control passes outside of QDL, hence there is only very clumsy access to the sources once set. The far better way is to just get the information you want directly in a QDL script.

Extended Attributes (xsa.)

These are sent by the client to the service in the initial request and are of the form:

NS:*path* = v0,v1,v2,…

Where NS is the namespace **oa4mp** or **cilogon** and *path* is just a standard path.

E.g. If a client sent this as part of its set of parameters

```
oa4mp:/tokens/access/lifetime=3600000&oa4mp:/roles=admin,users
```

Then in the environment, there would be a stem **`xas.`** with the value

```
{oa4mp:{/tokens/access/lifetime:3600000,/roles:[admin,users]}}
```

Note that OA4MP does nothing with these except pass them along to the scripting environment. They are also read only. One potential use is that the client can send custom information about group roles for a given user directly.

A final note is that there is no canonical way for OA4MP or QDL to determine what the types of variables are, since the request jumps through so many hoops, so they are all string-valued. In the case of the /**`refresh/lifetime`**. this means it needs to have **`to_number()`** invoked as needed, etc.

Example. Configuring the OA4MP client to send XAs

Now, the client ***must*** send the parameters as uris that start with either **`oa4mp:`** or **`cilogon:`** and these may be many valued. A typical use is in the webapp client configuration (which allows for static attributes and is useful for testing), so in the OA4MP client configuration you might have and entry like

```
<client name="myclient">
        <parameters>
          <parameter key="oa4mp:role">researcher</parameter>
          <parameter key="oa4mp:role">admin</parameter>
          <parameter key="oa4mp:/refresh/lifetime">1000000</parameter>
        </parameters>
<!-- bunch of other stuff - - >
</client>
```

If all goes well, then in the runtime environment you would get **`xas.`** as the stem :

```
{
  oa4mp :  {
    /refresh/lifetime :[1000000],
    role : [researcher,admin]
    }
  }
}
```

Flow State (flow_states.)

These are the states that the flow may be in. They are boolean values and setting them has an immediate impact on how processing is done.

Name	Description
`access_token`	Allow creating an access token
`id_token`	Allow creating the ID token
`refresh_token`	Allow creating refresh tokens

user_info	Allow creating user info
get_cert	Allow user to get a cert
get_claims	Allow the user to get claims
accept_requests	Deny *all* requests if false. This is the nuclear option to shutdown access.
at_do_templates	Allow execution of templates for access tokens.

A typical use might be the following. In the post_auth phase (so after the system has gotten claims) check membership and deny all access if not in a group:

```
if[
  exec_phase == 'post_auth'
 ][
  flow_states.accept_requests := has_value('prj_sprout', claims.isMemberOf.);
 ];
```

The effect is that if the isMemberOf claim (these are the groups that a user is in) does not include 'prj_sprout' then all access to the system is refused after that point. Note that system policies do have the right of way, so if the system would not normally let a user get a certificate, setting **get_cert** to **true** would be ignored.

Email (mail., message.)

As of OA4MP 5.4, QDL scripts now have access to QDL Mail,, a facility for sending email messages. If the server has email configured already, then the workspace inherits the configuration and the message (remember that the first line of the message is they subject.) These are in the stem variable **mail.**

```
   jload('mail'); // load the module if you are going to use it.
   mail#cfg(mail.'cfg');  // set the inherited server configuration
   mail#send(mail.'message'); // send the inherited message
```

You could also replace templated values with, e.g., the user metadata claims:

```
   mail#send(mail.'message', claims.);
```

You do not need to use the inherited configuration or messages and can send whatever you like. See the QDL Mail documentation for more information.

(Note: The documentation for QDL Mail talks about getting the correct mail jar. This is done already in OA4MP as part of a standard deployment, so you don't have to worry about it all. If OA4MP is working right, you should just be able to use email.)

```
user. := null;
   try[user. := script_run('utils/get_user', id);]
 catch[
       jload('mail')
       mail#cfg(mail.'cfg'); // use the server default mail
       mail#send(['Error getting user', // first line is the subject
                  'The user with id ' + id + 'was not found',
                  'message reads:' + error_message]);
       // Now, in this case, raise an error and exit
       raise_error('could not get user info for ' + id);
```

```
    ];// end catch
```

You could get more information from the **error_message**, **error_code** and **error_state.** variables inside the catch block, so this is really just to show how it works. If you did not raise an error, processing would fall through to the next line of code after the catch block (which is another option, depending on what you need to do).

Returned Tokens (access_token., refresh_token., claims.)

These variables are the actual content (payload of a JWT) returned by OA4MP. The **claims.** are turned into the id token. Changes to these are done by setting the value. You set the values to what you want, so e.g. **access_token.scope** should be set to a blank delimited string of scopes.

Initial Scope requests (audience., scopes.)

These are the audience and scopes in the **original** request at the time the flow is started. Note that "original" means the first place in the flow where they may be specified. They may not be altered and are lists.

Refresh/Exchange scope requests (tx_audience., tx_resource., tx_scopes.)

In either the refresh or exchange endpoint OA4MP allows for scope, audience and resource parameters as part of the request. These are processed into lists and supplied in their current state. This means that they are what is in the that request. A useful idiom to get the current set of these is something like

```
requested_audience. := (size(tx_audience.) == 0)?audience.:tx_audience.;
requested_scopes. := (0 < size(tx_scopes.))?tx_scopes.:scopes.;
```

which sets the **requested_audience.** and **requested_scopes.** to whatever is available in the workspace.

A Note on variable visibility

A common construct is to have a driver script like this:

```
  if[exec_phase=='post_auth']
then[script_load('my/access.qdl');];
  if[exec_phase=='post_token']
then[script_load('my/token.qdl');];
// .. may be others
```

Remember that in QDL, **[. . .]** (square brackets) tell QDL to create a specific scope for what is inside. The net effect is that every variable defined in the script resides in the then block and won't be saved as part of the state. How to get variables to persists between script invocations? Set the variable in the driver script but make sure not to overwrite a previous value:

```
my_var := (∃my_var)?my_var:null; // initialize to null, unless already set
  if[exec_phase=='post_auth']
then[script_load('my/access.qdl');];
```

```
  if[exec_phase=='post_token']
then[script_load('my/token.qdl');];
// .. may be others
```

So now you can simply set the value of my_var any place in your scripts and have it available.

Alternate approach would be to simply set an extrinsic (*i.e.* global) variable any place you want it by prefixing it with $$:

```
   $$my_var := 'bar'; // persists through all phases.
```

This requires no special handling but some people do not like global variables.

Error Handling

Errors in OA4MP scripts have to be handled as per the OAuth 2 spec. If there is an error inside a script, how can this get propagated to the runtime engine so that it may be handled by another component? For instance, if running a QDL script inside an OAuth 2 server, OAuth 2 has its own error handling specification that needs to be followed. What follows is how to configure error handling for use outside of QDL. There are two ways. The old way was used before QDL had its current error handling. The new way raises an error with a specific code. Both work. The newer way is much more flexible and hands off – errors just propagate. The old way required checking an error code after each script invocation and manually propagating the errors.

New Way – using raise_error()

This has a reserved code of in the system variable **`oa4mp_error`** (= 1000) and the (optional) stem should have certain entries for processing. The state stem that is passed in the error call contains the information for OA4MP. Most basic example:

```
if[
 script_args() != 2
 ]then[
 raise_error(
   'Sorry, but you must supply both a username (principal) and password.',
   oa4mp_error);
 ];
```

Supported stem entries. Note that the message (first argument) to raise_error is returned as the error_description.

QDL Key	OAuth	Description
`error_type`	**`error`**	OAuth 2 specific error type.
`status`	**`(HTTP status)`**	(Optional) The HTTP status set in the response.
`error_uri`	**`error_uri`**	(Optional) OAuth 2 error_uri

Full Example.

In this example, we need to raise an error for OA4MP during a flow with a specific HTTP status:

```
raise_error('raise_error test',
            oa4mp_error,
          {'error_type' :'error_type_message',
               'status' : 401
            'error_uri':'https://localhost/oops'}
);
```

This results in an OAuth error (to the client's callback uri) with HTTP status 401 and body

```
{
            "error" : "error_type_message",
 "error_description" : "raise_error test",
         "error_uri" : "https://localhost/oops",
             "state" : "b5gF_Sup2WNlLsB5ZPcjjFpEnPPSmowqeTwP-7GCAAs"
}
```

(In this case, the state value returned is part of the OAuth spec and added as needed.)

Tip: If you set the custom_error_uri below, you might want to set the status to 302.

Old Way – Set an error variable and return (deprecated!).

In a QDL script, you set the *error variable*

`sys_err.`

stem. If absent or **`ok`** is **`true`**, then no error has occurred. You can construct two types of exception, OAuth 2 exceptions and CILogon DB service exceptions (only available if you are running the CILogon extension to OA4MP). Note that this you simply return from the script at that point and must, in effect, do your own stack handling, so you must check each call after it returns if **`sys_err.ok`** is false and return. Raising an error does not require this and is hence more attractive.

Example: Propagating errors manually

If you set the error variable, you must check each script to see if an error should be propagated back. This example shows that. In this example, the script `'init.qdl'` is called, the ok flag on the error variable is checked and if true, then return immediately.

```
script_run('init.qdl');
if[!sys_err.ok][return();]; // If there was an error, return
```

Key value pairs for the error variable are:

QDL Key	OAuth	Description
`ok`	**`--`**	Must be **`false`** to trigger error handling
`message`	**`description`**	Human readable description
`error_type`	**`error`**	OAuth 2 specific error type.

status	(HTTP status)	(Optional) The HTTP status set in the response.
error_uri	error_uri	(Optional) OAuth 2 error_uri

If the HTTP status is not set, the specification says to default to 401.

E.g. Construct the error variable error in QDL in the case that there is an unauthorized client.

```
        sys_err.ok := false; // not ok, there is an issue
    sys_err.status := 401;
sys_err.error_type := 'unauthorized_client'; // authorizing the client failed
   sys_err.message := 'unknown client'; // unregister clients
 sys_err.error_uri := 'https://bgsu.edu/error';
```

This results in an OAuth error (to the client's callback uri) with HTTP status 401 and body

```
{
             "error" : "unauthorized_client",
  "error_description" : "unknown client"
}
```

Setting the **error_uri** returns it as well, as per the specification. (Note, the QDL runtime may add other information, such as **state** to this, the point is that creating **sys_err.** in QDL allows the OAuth error handler to take the appropriate action.)

E.g. Checking for scopes.

In this case, a scope is missing that is critical for operation. This throws a standard OAuth 2 error.

```
if[
 'org.cilogon.userinfo' ∉ scopes. // or !has_value('org.cilogon.userinfo',scopes.)
 ][
    raise_error('the org.cilogon.userinfo scope is required.',
                oa4mp_error,
               {'error_type' : 'invalid_request'});
];
```

CILogon extension

In addition to the OA4MP values above, the CILogon extension to OA4MP supports the following ***during authorization only***

QDL Key	CILogon	Description
code	status	(Optional) integer code for the error type
custom_error_uri	custom_error_uri	(Optional) error uri not in OAuth 2 spec.

These are either used in the error variable or as part of the state for the error.

In the authorization phase, these are returned by the DBService. The aim is to allow for a better user experience, so if, e.g. a user is not registered with the system, a custom uri can be supplied by the client

to redirect said user to some institutional sign-up page, rather than getting dumped into a generic CILogon failure page.

E.g.

Construct an explicit CILogon error variable:

```
             sys_err.ok := false;
          sys_err.code := 65541; // hex 0x10005, create transaction failed
     sys_err.error_type := 'access_denied';
        sys_err.message := 'could not create transaction, user not found';
sys_err.custom_error_uri := 'https://physics.bgsu.edu/user/register';
       sys_error.status := 302; // make sure it redirects!
```

(followed by a **`return()`**) or doing the exact some thing by raising an error:

```
raise_error('could not create transaction, user not found',
            oa4mp_error,
          {'code' : 65541, // hex 0x10005, create transaction failed
     'error_type' : 'access_denied';
'custom_error_uri' : 'https://physics.bgsu.edu/user/register'}
);
```

This would result in the following response from the DBService:

```
{
   status=65541,
   error_description= could not create transaction, user not found,
   error = access_denied,
   custom_error_uri= https://physics.bgsu.edu/user/register
}
```

CILogon note about setting the status: If you throw an OAuth 2 error that is processed by the DBService layer, it will be converted to a CILogon error. In that case, the status returned in the response (not to be confused with the HTTP status, since all DBService responses have that set to 200) will be set to 0x100007 (generic QDL error) since there is no easy association of random OAuth errors with CILogon specific errors. If you need to have a specific status returned (e.g., 0x10001, transaction not found), you should set it.

General Examples

Example: Checking arguments

The example here is Checking the number of arguments for a script to see if it should run and sending a useful message back:

```
if[
  size(args()) != 2
 ]then[
```

```
   raise_error('You need to supply both a username and password. Request denied.',
               oa4mp_error,
               {'error_type' : 'access_denied'});
 ];
// Otherwise, do stuff.
```

Example: Checking groups for memberships

QDL allows this quite simply though it might not be apparent. First off, there is a function that is available called **`in_group2`** (There is a deprecated version called **`in_group`** – don't use that, which is clunky and old) which has syntax

```
in_group2(group_names, groups.)
```

where **`group_names`** is a name (a string) or stem of them. `groups.` is the groups from a claim source. Now the rub is that the structure of `groups.` depends on the source. Some sources return just a flat list of names and some return a JSON structure that has to be parsed. **`in_groups2`** handles these cases. The result is conformable with the left argument. So a typical invocation is

```
   in_group2('all_ncsa', claims.isMemberOf.)
true
```

which shows that the name **`all_ncsa`** is in the **`claims.isMemberOf`**. Since the result is left conformable, if you wanted to check a list of names, you might do something like

```
   g. := claims.isMemberOf.; // Keep it readable here
   g_reject. := ['disabled','banned','deny_all','deny_web'];
   deny. := in_group2(g_reject., g.);
   deny.;
[false,false,false,true]
```

So the user is in the **`deny_web`** group. How to check if at least one of those is true? Use the **`reduce`** function with `||` (logical or):

```
   reduce(@||, deny.);
true
```

which is identical to evaluating

```
   false || false || false || true
true
```

Since the reduce function just slaps `||` between all elements of the list. A use might be

```
   if[reduce(@||, deny.)]
 then[flow_state.'accept_requests' := false;
      return();
     ];
```

In which the user is found to be in the **`deny_web`** group so all further access is denied and the flow is stopped dead in its tracks.

A full example

In this example, group memberships are checked and if a person is not in them, an error is raised.

```
chk_group(rejects., groups.) -> reduce(@||, in_group2(rejects., groups.));

if[
  chk_group(g_reject., z.)
]then[
  raise_error('User not in group. Cannot determine scopes.',
  oa4mp_error,
  {'error_uri' :'https://phys.bsu.edu/users/register";
  'error_type' : 'access_denied';
      'status' : 404}
  );
];
```

This defines a function, chk_group and uses that to craft an error. A message is returned along with the status and in this case, a uri is passed back so whatever handles this can redirect the user appropriately.

Setting Extended attributes in the CLI

These are parameters sent to the server by the client in the initial request. First off, the client *must* have the ability to process them turned on. This is in the extended attributes for the client, so in the CLI, set the client id, then issue

```
update -key extended_attributes
```

and when prompted enter

```
{"oa4mp_attributes": {"extendedAttributesEnabled": true}}
```