Server Scripts

Version 1.4

Introduction

Any system that uses QDL as a scripting environment should use the format here to inject configurations. QDL will take these (JSON) and translate them into viable QDL scripts, setting up any environment, passing arguments (suitably processed) etc.

How scripts are accessed

In OA4MP, scripts are embedded in token handlers. See the token handler documentation for more. What is described here is the basic mechanism that can be utilized by any extension that wants to include QDL as its scripting language.

Configuration Format

This is the format for the configuration entry used by QDL . This is an element in JSON and may be put any place. This is the grammar for scripts:

```
{"qdl" :
   BLOCK | [BLOCK+], // Either a block or array of blocks
   }
```

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

}


CODE:
  ("load" : LINE) | ("code"  : LINE+)

// NOTE: "load" implies zero or more arguments. "code" has no arguments and
//       any will be ignored. It is possible to send enormous blocks of code this
         way, but is discouraged. Put it in a script and call that.


// XMD = eXecution MetaData, how this should be loaded by the scripting engine
//       This sets up the environment. Since 'environment' is overused, we chose
//       xmd instead.

XMD:
```

```
  "xmd":{
    "exec_phase" : PHASES,
    "token_type":"id" | "access" | "refresh"
   }


ARGS:
  "args" : ARG | [ARG+] // Either a single arg or an array of them

ARG:
   STRING | INTEGER | DECIMAL | BOOLEAN | JSON

PHASE:
   ("pre" | "post") _ ("auth" | "token" | "refresh" | "exchange" | "user_info")

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

One leading question is this: Why have a separate load call? The reason is that many people who will want to use this are not proficient in QDL. This lets them call a script (from a library, *e.g.*) and send along a standard JSON object with no knowledge of how it works. This is probably the most common use case for scripting and lets an administrator delegate the scripting work to others without having to require them to learn yet another language. If they know the name of the script and the inputs, they can quiddle.

Handlers

There are 3 main handlers:

- id_token
- access_token
- refresh_token

These actually run the scripts in them. They have certain types of information available. For instance, the id_token handler does not have any information about access tokens, because it is created in the authorization phase before access tokens exist. You only need an access_token handler if you want to have a JWT returned (such as a SciToken). If there is no access_token handler, you will get back a plain string for the token. Similarly, the refresh token handler is used when a JWT for the refresh token is required. None of the handlers require QDL scripts and will return very basic information.

Phases

A phase (denoted as `exec_phase` in the xmd block) itells OA4MP when to load the code or script and attempt to execute it.

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 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. If you set something in, say, the pre_auth phase, it will be there in the post_exchange phase (e.g.) However, 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. This means that for each handler, the script will be run in the given phases. In this case, no phases specified means to run at every phase. This is typically for a driver script, where all the logic is off-loaded to QDL

Examples

Running code directly

Running a single line of code.

```
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.

Running multiple lines of code.

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

(This must be in a tokens configuration as part of, *e.g.*, an identity token.) This takes a claims.uid (like `'http://cilogon.org/serverA/users/12345'`) parses it and asserts a new claim, `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.

Accessing information in the runtime.

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

Variable	U	Component	Description	Comment
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.
claims.	+	id token	claims	The current set of user claims that will be used to create the ID token.
access_token.	+	access token	claims	The current set of claims used to create the access token (if that token requires them).
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).
xas.	-	all	extended attributes	Extra attributes (namespace qualified) that may be sent by a client.
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.
refresh_token.	+	refresh token	claims	Claims used to create the refresh token, if supported.
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.
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_scopes.	-	refresh, access token	TX scopes	requested scopes for TX
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.
`access_control.client_id`		all	client id	The client id is always a specific entry in the access control list.
`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.

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.

Extended Attributes

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.

Tip - 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
```

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.

Email

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).

Errors

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 and error with a specific code. Both work.

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.

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

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, CILogon 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.

Example: Checking argument

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'];
   z. := in_group2(g_reject., g.);
   z.;
[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(@||, z.);
true
```

which is identical to evaluating

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

Since the reduce function just slaps `||` between all elements of the list.

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.

Extended attributes

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}}
```

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 configuration file 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.oa4mp. as a stem :

```
   say(xas.);
{
  oa4mp :  {
    /refresh/lifetime :[1000000],
    role : [researcher,admin]
    }
  }
}
```

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

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.