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 there are no runtime guarantee as to execution order., hence the 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.

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.

Errors

If there is an error inside a script, how can this get propagated to the runtime engine? The answer is that inside QDL, you set the

`sys_err.`

stem. rather than the standard **`raise_error()`**, since that tells the runtime engine to handle it differently and allows for different propagation outside of QDL (to e.g. an OAuth server), rather than dispatching it internally. If absent, then no error has occurred. Thanks to Evolution, this allows you to construct three distinct types of exception, standard QDL exceptions, OAuth 2 exceptions and CILogon DB service exceptions.

Standard QDL Exception

These are used for throwing exceptions inside of QDL scripts that are propagated outside. They simply require a message

QDL Key	OAuth	Description
`ok`	**`--`**	Must be **`false`** to trigger error handling
`message`	**`description`**	Human readable description

E.g.

```
if[
 script_args() != 2
 ]then[
 sys_err.ok := false;
 sys_err.message := 'Sorry, but you must supply both a username (principal) and
password.';
 return();
 ];
```

Note if you use **`raise_error`** this won't be picked up by the runtime engine, but will be treated like any other QDL exception. Eventually the script will bomb, but you won't have much control over that.

In this case, a standard QDL exception internally will be raised with the give message which will be propagated up the stack.

There are extensions available which are handled by the runtime engine differently.

OAuth 2 extension

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
`custom_error_uri`	**`custom_error_uri`**	(Optional) error uri not in OAuth 2 spec.

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

E.g. Construct a 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 := 'unauthorized_client'; // authorizing the client failed
sys_err.message := 'unknown client'; // it failed because of an unregistered client
```

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

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

Optionally, the **`error_uri`** could have been set and that would have been returned as well, as per the specification.

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.)
 ][
          sys_err.ok := false;
   sys_err.errorType := 'invalid_request';
     sys_err.message := 'the org.cilogon.userinfo scope is required.';
     return([]);
];
```

CILogon extension

QDL Key	CILogon	Description
ok	--	Must be **false** to trigger error handling
message	**error_description**	Human readable description of the error
error_type	**error**	Type of the error
code	**status**	(Optional) integer code for the error type
error_uri	**error_uri**	(Optional) OAuth 2 error_uri
custom_error_uri	**custom_error_uri**	(Optional) error uri not in OAuth 2 spec.

E.g. 1 Construct a n explicit CILogon error

```
              sys_err.ok := false;
            sys_err.code := 65541; // hex 0x10005, create transaction failed
           sys_err.error := 'access_denied';
         sys_err.message := 'could not create transaction, user not found';
sys_err.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
}
```

Note especially that sys_err is a variable that only exists inside the runtime engine and is not generally available outside of that. It allows us to easily control which errors in the runtime should be made available to the user (vs. having a try … catch block that allows the errors to be handled inside the scripts).

Example: Propagating errors

The example here is Checking the number of arguments for a script to see if it should run and sending a useful message back. The main script, `do_at.qdl`, calls `init.qdl`. The calling script should always check if there is an error to propagate back:

In `do_at.qdl`:

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

In `init.qdl`:

```
if[
  script_args() != 2
 ]then[
        sys_err.ok := false;
   sys_err.message :=
          ‘You need to supply both a username and password. Request denied.';
sys_err.error_type := ‘access_denied‘;
   return();
 ];
// Otherwise, do stuff.
```

The effect would be to throw an exception in the runtime engine which Java then will process as if the exception had happened there. Note that for debugging purposes, (such as running the script inside your workspace) `sys_err` is benign – it just sets a variable which then goes away on the return().

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.` The result is left conformable, so if you wanted to check a list of names, you might do something like

```
   g. := claims.isMemberOf.; // Keep it readable here
   g_reject. := ['all_disabled_usr','grp_banned','mg_deny_all','mg_deny_web'];
   z. := in_group2(g_reject., g.);
```

```
   z.;
[false,false,false,true]
```

So the user is in the `mg_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[
  sys_err.ok := false;
  sys_err.message := 'User not in group. Cannot determine scopes.';
  sys_err.error_uri:='https://phys.bsu.edu/users/register";
  sys_err.error_type := 'access_denied';
  sys_err.status:=404
  return();
];
```

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.