Server Scripts

Version 1.2.2

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.

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":{
   "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”)

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

Examples

Loading a simple script, in the script path, with no arguments

```
{"qdl":{"load":"x.qdl"}}
```

Note that if there were arguments, they would be included in the arg_list.

Running a single line of code.

```
{"qdl":{"code":"init();"}}
```

Loading a script and passing it a list of arguments.

```
{"qdl":
   {
     "load":"y.qdl",
     "xmd":{"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’,
   to_list(4,true,from_json(‘{"server":"localhost","port":443"}’))
   );
```

Loading a script with a single argument

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


{"qdl":
     [
      {"load":"init.qdl", "xmd":{"phase":"pre_auth"}},
      {"load":"lsst.qdl", "xmd":{"phase":"post_token"}}
     ]
}
```

Another example

Here is the JSON supplied:

```
{"qdl": {
  "load": "y.qdl",
  "xmd":   {
    "phase": "pre_auth",
    "token_type": "access"
  },
  "args":   [
    4,
    true,
    -47.5,
        {
      "server": "localhost",
      "port": 443
    },
        [
      3,
      4
    ]
  ]
}}
```

Here is the resulting QDLScript object created (line breaks added) internally, in case you were wondering:

```
QDLScript{code=
script_load(
   'y.qdl',
   to_list(4,true,-47.5,
   from_json('{"server":"localhost","port":443}'),
   from_json('[3,4]'))
   );
properties={
   phase:pre_auth,
   lang_version:1.1,
   token_type:access,
   language:qdl
  }
```

```
}]}
```

A few things to note. The reason that the script is invoked with an argument is that there has to be a hand-off between the scripting engine and the runtime environment. The state literally does not exist until execution (and may be (re)created based on phases) so there is no way to pass an argument list per se.

A HOCON example

This is interesting because there are multiple phases and defaults for every acccess token.

```
tokens{
    access{
       type=wlcg
       issuer="https:cilogon.org"
       audience="https://wlcg.cern.ch/jwt/v1/any"
       lifetime=3600000
       qdl{
          load="fnal/fnal-at2.qdl"
          xmd={exec_phase=["post_token","post_refresh", “post_exchange”]}
          args=["USER_ID", "PASSWORD"]
       } //end QDL
    } // end access token
  } //end tokens
```

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	Requested audiences in the initial request.

			audience	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.

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. If absent, then no error has occurred. If present then it has two attributes

ok = a boolean that if true, means no error happened, false means the runtime engine propagates the error.

message = A message (probably human readable) that will be returned.

Note especially that this 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

Checking the number of arguments for a script to see if it should run and sending a useful message back.

```
if[
  script_args() != 2
 ]then[
   sys_err.ok := false;
   sys_err.message := 'You need to supply both a username and password. Request
denied.'
   return();
 ];
```

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, this is benign – if it's there outside the script runtime engine, it sets a variable which then goes away on the return().

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 and

```
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	Accept deny *all* requests
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.