Using the QDL with OA4MP Stores

Introduction

Access to the stores that OA4MP uses are available in a QDL session via a module. These allow you to search, get, save, etc. objects that are stored there. One key point is that reading an object converts it to QDL (a stem of its properties) so you can work with it as you like, then simply save the result which will be translated back to the correct database entry. Using just the standard QDL databaase module means you get the raw output and must take care when saving or updating. This module does all the translation and checking for that and includes convenience methods for working with versions.

Use Overview

Supported stores are accessed by creating a module. Each module gives access to a specific store type (*e.g.*, all the clients under an SQL database). You may have as many of these as you want. You must have access to the database to do this. It is therefore properly an administrative tool, not a client side tool. For updating clients as a standard user, you should use the Client Management endpoint and associated protocols in OA4MP.

Importing the module

Since this is a Java module,you need to make sure that the compiled classes are in the class path. The easiest way to do this is to use the OA4MP (or CILogon) distribution.

Supported stores

There are several supported stores. These are supplied as the argument to the initialization call.

- client
- client_approval
- admin_client
- permission (note this is used with module oa2:/qdl/p_store)
- transaction
- tx_record

Example

The store types are stashed in an extrinsic (*aka* global) variable, **`$$STORE_TYPE.`**, and is available as soon as the store is loaded (so no need to import one first). Here we print it:

```
print($$STORE_TYPE.)
```

```
        admin : admin_client
     approval : client_approval
       client : client
   permission : permission
  transaction : transaction
           tx : tx_record
```

Tip: Since it is extrinsic, you need to supply the -extrinsic flag when listing variables:

```
   )vars -extrinsic
$$STORE_TYPE.
(and maybe other variables too.)
```

Using the module

Before any store can be accessed, the corresponding module must be loaded, imported and initialized. Loading the module can be done directory or it can be done in the configuration file.

Example

Here is an example snippet from a configuration file

```
<modules>
   <!-- other stuff -->
       <module type="java"
           import_on_start="false">
            <class_name>edu.uiuc.ncsa.oa2.qdl.storage.StoreAccessLoader</class_name>
        </module>
</modules>
```

Note that this should not be imported in load because you want to make several different modules with descriptive names. You must then import them as needed.

Note that when importing a module, you should give it a name that makes sense. You may of course import the same module to *e.g.*, access two different stores so you can compare them, copy between them, etc.

Example

```
poloa := j_load('oa4mp.client.store');
polob := import('oa4mp.client.store');
```

This now gives two modules to two (possibly different) stores.

XML Serialization

You can convert a store entry to XML and this makes them portable between stores and external applications. You may also read them back in. This is the purpose of the **`from_xml`** and **`to_xml`** functions.

Versions

The store versions entries and the store module can create, remove, read and list versions. Note that the argument for these operations is to the function, it is *not* set in the object. So if you had an object like

```
   my_client.
{client_id : oa4mp:/123/567, …
```

The proper way to version it with this API is

```
  c#version('oa4mp:/123/567');
3
```

Indicating that version 3 of this object has been created.

If you are working with API calls (such as **`read`**) that accept version numbers, you may either send it directly, e.g. these are the same

```
  c#read('oa4mp:/123/567', 3);
  c#read(['oa4mp:/123/567', 3]);
```

A *versioned id* is a list of the form **`[id, version],`** where **`version`** is an integer that is the number of the version you want. Versions follow QDL indexing rule for sign, so an index of -1 is the most recent, 0 is the oldest, etc., You may send these as part of a call. So for example if you wanted to get a mixed list of various objects, some of which are versioned you could issue

```
  ids. := ['oa4mp:/client/234234',
           ['oa4mp:/client_id/1aaf', -1],
           ['oa4mp:/client_id/3dfb',4],
           'oa4mp:/client_id/4e3c',
           'oa4mp:/client_id/5667',
           'oa4mp:/client_id/6de5'];
  c#read(ids.)
(output)
```

which would get the clients and in particular return the latest version of **`oa4mp:/client_id/1aaf`** and the 4th version of **`oa4mp:/client_id/3dfb`**.

A final caveat is that the system manages the versions by overloading the id (primary key in the store) and changing that will not change the entry but overwrite what has that id, which may simply create a new entry of replace an existing one. Generally you should not change the id for an entry. The exception might be creating one, where you can set it as you like since it does not exist in the system yet.

Handling permission stores

The permissions granted by an admin client to a its regular clients are stored in a permission store. This is an extension of a regular store (so the general API applies), plus a few calls that are very specific. All permissions relate to clients, admin clients and ersatz clients, so you don't need multiple premission

stores generally, just one per database. You must load the correct module though, which as namespace **`oa2:/qdl/p_store.`**

Importing and initializing the module:

```
p := j_load('oa4mp.client.p_store');
p#init(cfg.);
```

As a simple check, here is the size of the store

```
  p#count();
1046
```

Generally the number of permissions will be quite high, since multiple permissions may be needed.

```
  p#get_clients('myproxy:oa4mp,2012:/adminClient/58d9bd82dc3a7628098c2424454474b/
1588334962389');
[
 testScheme:oa4md,2018:/client_id/45fdf5cd13db221337cb32d90e01b834
]
```

This means that the given admin manages (in this case) exactly one client. Compare this with

```
  p#get_admins('testScheme:oa4md,2018:/client_id/45fdf5cd13db221337cb32d90e01b834')
[
 myproxy:oa4mp,2012:/adminClient/58d9bd82dc3a7628098c2424454474b/1588334962389
]
```

which lists the administrative clients (if any) for a given client.

To find out about the keys,

```
  p#keys()
[
 permission_id,
 admin_id,
 can_approve,
 can_create,
 can_remove,
 client_id,
 can_read,
 can_write
]
```

Something to notice is that there is a specific permission id.

The general API for store access.

count

Description

Return the number of objects in the store

Usage

```
count();
```

Arguments

none

Output

A non-negative integer that is the number of items in the store.

Example

```
   clients#count();
1417
```

Tells us that there are 1417 in the store which is referenced by **`clients`**.

create

Description

Create a new object, returning a stem with default values in place.

Note: this is *not* in the store until it is saved. Attempts to update it will therefore fail. This allows separation of creation and save semantics (*aka* CRUD operations).

Usage

```
create([id])
```

Arguments

no args = create the object with a randomly generated identifier

id = use this id to create a new object. Note that it must be a valid uri.

Output

A stem representing the object. This has whatever defaults the system uses, so it's a good bet you want to update it.

Examples

```
  admin#create()
{
 allow_qdl:false,
 last_modified_ts:1610031137666,
 admin_id:oa4mp:/adminClient/1f594ca1aa30d3a06d4d618b5cc3f23b/1610031137666,
```

```
 creation_ts:1610031137666,
 max_clients:50
}
```

This creates the new in this case admin client object with a random identifier. Since it succeeded, there was not an existing object with this identifier in the store. To emphasize the point

```
  admin#read('oa4mp:/adminClient/1f594ca1aa30d3a06d4d618b5cc3f23b/1610031137666')
Error: Could not find the client with id
"oa4mp:/adminClient/1f594ca1aa30d3a06d4d618b5cc3f23b/1610031137666"
```

from_xml

Description

Convert a storage record from XML format to a stem.

Usage

```
from_xml(string)
```

convert the given string (of XML) into the correct type of stem for this store.

Arguments

A string that is the serialization

Output

A stem.

Examples

```
   ligo. := client#from_xml(file_read('ligo_client.xml'));
true
```

This will read in a file as a string and then convert it to XML. Remember that each store has a specific format that must be used, so the conversion is for objects of the correct type. This does not store it, so you must save this if you want it in the store. The result is not guaranteed to be a functional object (this lets you use an XML template and fill in the rest of it) and no checking on the validity is done.

init

Description

Initialize the store. Part of initializing the store is *e.g.*, making any database connection, alllocating local resources *etc*. This must be done before any other calls are made.

Usage

```
init(stem. | file, name{, type} )
```

Arguments

Either all three arguments are passed are a stem containing them is.

file = the full path to the configuration file.

name = the name of the configuration within the *file*.

type = the type of store (see the section above for a listing of supported types).

stem. = a stem of these values.

Output

A boolean value of *true* if this succeeded.

Examples

The following are equivalent, assuming you have imported the module as **`clients`**:

```
  clients#init('/home/me/qdl/config/server-oa2.xml',
               'localhost:command.line',
               $$STORE_TYPE.client)
true
```

As well as

```
  cfg.file := '/home/me/qdl/config/server-oa2.xml';
  cfg.name := 'localhost:command.line';
  cfg.type := $$STORE_TYPE.client;
  clients#init(cfg.);
true
```

and the final argument for type may be supplied, which will override the type in the stem:

```
  approvals#init(cfg., $$STORE_TYPE.approval);
true
```

keys

Description

List the keys of attributes for objects in this store. These are the indices of stems for instance.

Usage

```
keys([b])
```

Arguments

No arguments = list all of the keys

b = boolean, if true, list the key that is the unique identifier for this type of object. If false (the default), all keys are printed.

Output

Either a single key or a list of all the keys. Note this is the only call that can be made without initializing the store.

Examples

Get all of the keys:

```
   admin := import('oa4mp.client.store');
true
   admin#keys();
[
 admin_id,
 name,
 email,
 creation_ts,
 secret,
 last_modified_ts,
 config,
 issuer,
 max_clients,
 vo
]
```

This means that for this client, these are the stem indices that will be recognized by the system. Note the unknown keys are ignored so if you set

```
   my_admin.id := 'my:id';
```

(so you are referencing a new key named **id**, which does nto correspond to a column in the backing database)

then try to *save* this object, this will not succeed, since the required key as per above is *admin_id*.

Another example. Get the unique id key for this object

```
   admin#keys(true);
admin_id
```

list_versions

Description

Given an id or stem of them, list the current set of versions in the store,

Usage

```
list_versions(id), list_versions(arg.)
```

Arguments

id = a string that is the id of the entry

arg. = a stem of ids.

Output

A conformable list of version numbers. Note that the version numbers are always a list.

Examples

```
  ids. := {
 'old_admin' : 'oa4mp:/admin_id/e18234',
  'previous' : 'oa4mp:/admin_id/e793c',
  'old_ligo' : 'oa4mp:/admin_id/f1a54'
};
   admin_store#list_versions(ids.)
{
'old_admin' : [],
 'previous' : [0,1,3,4,6],
 'old_ligo' : [0,1,2]
}
```

The meaning is that **`old_admin`** has no versions, **`previous`** has several, but a few have been deleted, and **`old_ligo`** also has a few versions.

read

Description

Get an object from the store by (unique) identifier.

Usage

```
read(id), read(id, version), read(id.)
```

Arguments

id = the unique identifier (a uri) for this in the store.

version = the version number

id. = a stem of simple ids or versioned ids.

Output

A conformable stem consisting of the attributes for this object. If you supply a stem, the objects with corresponding keys are returned. Note that if there is no such object, a **`null`** is returned. then
See also: *keys()*.

Examples

```
    a. :=   admin#read('myproxy:oa4mp,2012:/adminClient/95bff80b6a23d2612c56/16051275');
    a.
{
 allow_qdl:false,
 last_modified_ts:1605128630000,
 admin_id:myproxy:oa4mp,2012:/adminClient/95bff80b6a23d2612c56/16051275,
 name:Test admin client #42,
 creation_ts:1605128630000,
 vo:aqTvMdAUdiTcbko6kItlZaF7SFFbI6Rr_xCArhTa6LfIbwmHQ,
 secret:L7InEVi8pRfKuW1u4SzXL-sLRsoWj19IxpQ9yIbuQ-EXDiUHWn3Q,
 config:{},
 issuer:https:/physics.bsu.edu,
 max_clients:50,
 email:bob@phys.bsu.edu
}
```

Getting ids can be done by hook, crook or with the *search* call.

remove

Description

Remove an object from the store. Note that there is ***no*** confirmation done since this is not an interactive API. The object is deleted straight up.

Usage

```
remove(id), remove(id, version), remove(ids.)
```

Arguments

id = the string that is a unique identifier for this object.

version = the version of the object to remove

ids. = a stem of the ids or versioned ids.

Output

A conformable result with *true* if the object is no longer in the store. Note that if there were no such object to start with, the result would still be true, because this is properly an assertion about the state of the store after this call. If the argument cannot be decoded, it is skipped

Examples

```
   admin#remove(a.client_id)
true
```

The object represented in *a.* is not in the store now. Note this assumes that the primary id of the object is **`client_id`**.

Another example, deleting versions

```
  ids. := {
 'old_admin' : 'oa4mp:/admin_id/e18234',
  'previous' : ['oa4mp:/admin_id/e793c', 3],
  'old_ligo' : ['oa4mp:/admin_id/f1a54',2],
      'date' : date_ms()
};
  admin_store#remove(ids.)
{
  'old_admin' : true,
  'previous' : false,
  'old_ligo' : true
}
```

So this tells us that 2 of the entries could be removed and one could not. Normally this means that the operation failed at the database level, so check the database logs. The **`date`** entry could not be understood as an id by the system and we not processed, hence no result is returned for it.

restore

Description

restore an archived version, replacing the active version.

Usage

```
restore(id, version), restore(ids.)
```

Arguments

id = a string that is the identifier

version = the version number to restore.

ids. = a stem of versioned ids.

Output

A conformable stem of booleans with a true or false.

Example

```
   clients#restore('oa4mp:/client_id/1ec3a766', -1)
true
```

The most recent archived version of this was restored, overwriting what was active. Note that the version still stays in the system unless you explicitly remove it.

save

Description

Save an object to the store. Note that if the object does not exist in the store, it is created and if it does exist, it is updated.

Usage

```
save(stem. | list.)
```

Arguments

stem. = single stem holding the attributes for an object

list. = a list of such objects.

Output

A conformable list of booleans. So if there is a single object, you will get a scalar and if there is a list of them, you will get a result for each element. Note that this may give partial results. So if there is a failure in saving an object a log entry will be made and the corresponding result will be *false*.

Examples

```
    clients#save(my_client.);
true
```

search

Description

Search the store by key using a regex.

Usage

```
search(key, regex)
```

Arguments

key = the key of the object

regex = a regular expression to be applied to the contents of each *key*'s value

Output

A **stem** of objects. Note that this may be very, very large if your regex is not carefully scoped, as in, it is possible to pull the entire store into the workspace. This may or may not work given memory limitations and such so do be careful in your choice of search parameters.

Examples

```
   admin_search. := admins#search('admin_id', '.*256.*');
   size(admin_search.);
1
    my_admin. := admin_search.0;
```

In this case, we search the admin_id attributes in the entire store for one that has the string *256* embedded in it. (*E.g.*, the trick to get a specific object by id using a snippet of the whole identifier). This tells us there is exactly one element found. To access it you would need to grab the entry you want.

toXML

Description

Create the XML representation of an object. Objects may be serialized to XML (*e.g.* using the CLI or command line interface in OA4MP).

Usage

```
to_xml(stem.)
```

Arguments

stem. = the object to be converted

Output

A string consisting of XML for this object. It may be archived, sent to another person or whatever.

Examples

Taking the output of the **`read`** example, we convert it to an XML document:

```
  admin#to_xml(a.)
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>OA4MP stream store</comment>
<entry key="allow_qdl">false</entry>
<entry key="last_modified_ts">2020-11-11T21:03:50.000Z</entry>
<entry key="admin_id">myproxy:oa4mp,2012:/adminClient/95bff80b6a23d2612c56/16051275</entry>
<entry key="name">Test admin client #42</entry>
<entry key="vo">aqTvMdAUdiTcbko6kItlZaF7SFFbI6Rr_xCArhTa6LfIbwmHQ</entry>
<entry key="creation_ts">2020-11-11T21:03:50.000Z</entry>
<entry key="secret">L7InEVi8pRfKuW1u4SzXL-sLRsoWj19IxpQ9yIbuQ-EXDiUHWn3Q</entry>
```

```
<entry key="issuer">https:/physics.bsu.edu</entry>
<entry key="max_clients">50</entry>
<entry key="email">bob@phys.bsu.edu</entry>
</properties>
```

Again this is a string and can be saved to a file.

To get and serialize an id in one call

```
   my_client =c#to_xml(c#read('oa4mp:/client_id/f2e1823415'));
   size(my_client);
1684
```

update

Description

Update an object. This will fail if the object does not exist and is more or less equivalent to a database UPDATE command.

Usage

```
update(stem. | list.)
```

Arguments

stem. = single object to be updated.

list. = list of objects (as stems) to be updated

Output

A conformable result to the argument consisting of booleans as to whether the update worked or not. Note that any error messages will be written to the log. Note that this must be a list only if you are sending multiples since there is no way to tell it apart from the stem of an entry.

version

Description

Create a version from an existing entry. The entry must already exist in the store to do this.

Usage

```
version(id | id.)
```

Arguments

id = a string that is the id of the entry

id. = a stem of ids to be versioned.

Output

A conformable result with the number of the version created

Example

```
    admin_store#version('myproxy:oa4mp,2012:/adminClient/b6a23612c')
3
```

indicating that a version named 3 was created and that is the highest version number, meaning the most recent.