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.

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.

Tip

In the workspace, if you want to look at your module functions, you can reduce the clutter with the *-compact* switch, *e.g.*

```
   )funcs -compact

[poloa,polob]#init()
[poloa,polob]#read
…
```

This lists all the aliases from a given module as a list.

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 build your jars or use the distro.

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 included in the store once it has been imported

```
  admin#store_types.
[
 client,
 client_approval,
 admin_client,
 permission,
 transaction,
 tx_record
]
```

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 is not imported in load because you want to make probably several different modules with descriptive aliases. You must then import them as needed. This loader gives ***two*** stores, a general store and a specific permission store.

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

```
module_import('oa2:/qdl/store', 'poloa');
module_import('oa2:/qdl/store', 'polob');
```

This now gives two modules to two probably different stores.

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. You must load the correct module though, which as namespace **`oa2:/qdl/p_store`**.

Importing and initializing the module:

```
module_import('oa2:/qdl/p_store', 'p');
p#init(cfg.);
```

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

```
  p#size();
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.

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.

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

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"
```

Initialization

Each store requires initialization before use. This means it needs a file, config name and store type.

Example

If I import a client module named clienta, the correct

```
clienta#init('/home/ncsa/dev/csd/config/server-oa2.xml',
'localhost:oa4mp.oa2.mariadb', 'client')
```

fromXML

Description

Convert a storage record from XML format to a stem.

Usage

```
fromXML(string)
```

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

Arguments

A string that is

Output

A stem.

Examples

```
x := file_read('ligo_client.xml');
ligo. := fromXML(x);
```

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. The result is not guaranteed to be a functional object (this lets you use an XML template and fill in the rest of it).

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(file, name, type | stem.)
```

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).

Output

A boolean value of *true* if this suceeded.

Examples

The following are equivalent

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

```
true
```

As well as

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

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:

```
   module_import('oa2:/qdl/store', 'admin');
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';
```

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
```

read

Description

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

Usage

read(id)

Arguments

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

Output

A stem consisting of the attributes for this object. Note that the indices of the stem are found with the *keys()* call.

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 | stem.)
```

Arguments

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

stem. = a stem of the object. Only the identifier will be accessed.

Output

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.

Examples

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

The object represented in *a.* is not in the store now.

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 objects

list. = a list (integer indices) of 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

```
    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, iit 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

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

toXML(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#toXML(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.

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.

Examples

Blurb:
The store module(s) allow you to access OA4MP stores in QDL. Supported stores are
client for clients.
client_approval for approvals of clients (including admin clients)
permission for managing the permissions between admins and their clients. Use the p_store module
admin_client for administrative clients
tx_record for tokens created by the exchange endpoint (RFC 8693)
transaction for pending transactions.
There is a list of these in the variable store_types.

Every store allows for the following commands (online help is available for these in teh workspace using e.g.

)help p_store#init

fromXML Take and object's XML serialization (as a string, e.g. in a file) and convert to a stem.
init initialize the store. You cannot use a store until you call this.
keys Lists the keys possible in the stem for this store.
read Read an object from the store by identifier. Note the result is stem
remove Remove an object from the store by identifier.
save Save an object to the store. If the object exists, it is updated, otherwise a new object is created in the store.
search Search the store by key and value for all objects. This returns a list of stems (which may be huge -- plan your queries!)
size The size of the store, i.e., number of objects in the store
toXML Serialize a stem object into XML (as a string).
update Updates an existing object. This will fail if the object does not exist.
*get_admins Lists the admins for a given client
*get_clients Lists the clients for a given admin
*client_count Return the number of clients this associated with the admin.
* = only available in permission stores.

Note that you should create a module for each type of store you want and each store using the module_import command

The alias should be descriptive, so something like 'client' or 'trans' and yes, you may have as many imports as you like talking to different stores -- this is an easy way to move objects from one store to another