[KBase Authorization]

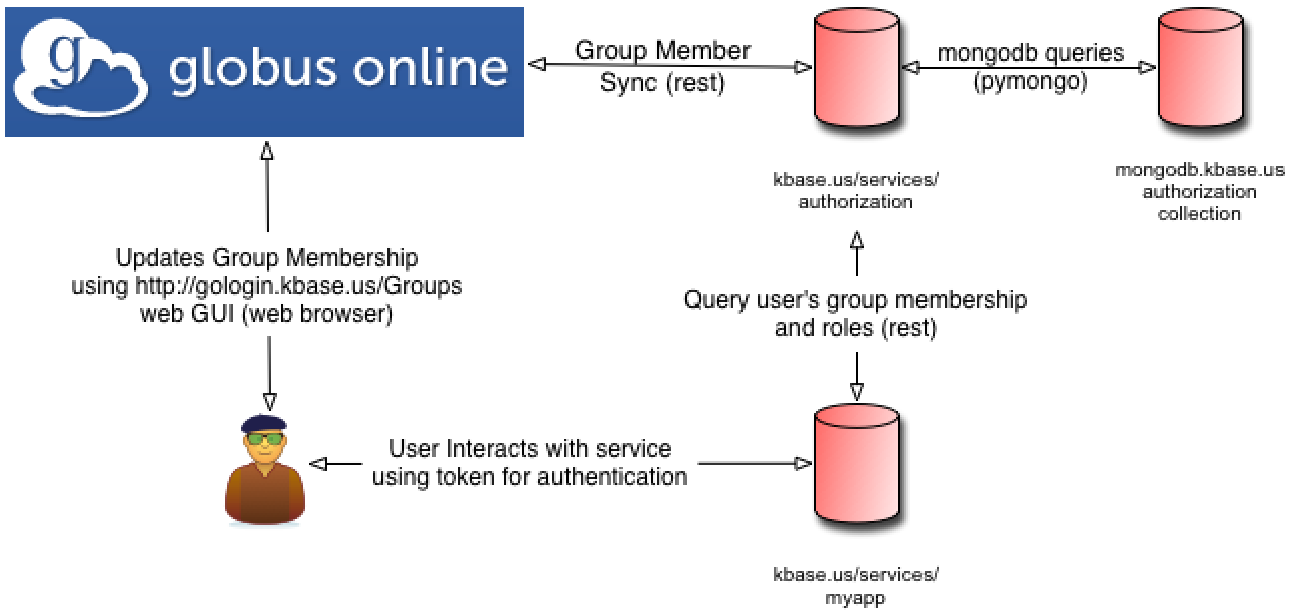
Table of Contents

Introduction

This iPython Notebook is a brief introduction to the overall design and architecture of the KBase Authorization Service as of May 2013. The authz service is considered to be an initial implementation providing core functionality for cross service authorization within the KBase environment. The design goal is to provide a lightweight service that stores assertions about the operations that users can perform on KBase documents - a service requiring authorization information must query the authz service and then enforce the permissions that are returned. When a relying service creates a new object, the relying service must assign the new document to an existing access control list in the authz service, or generate a new access control list for the new object.

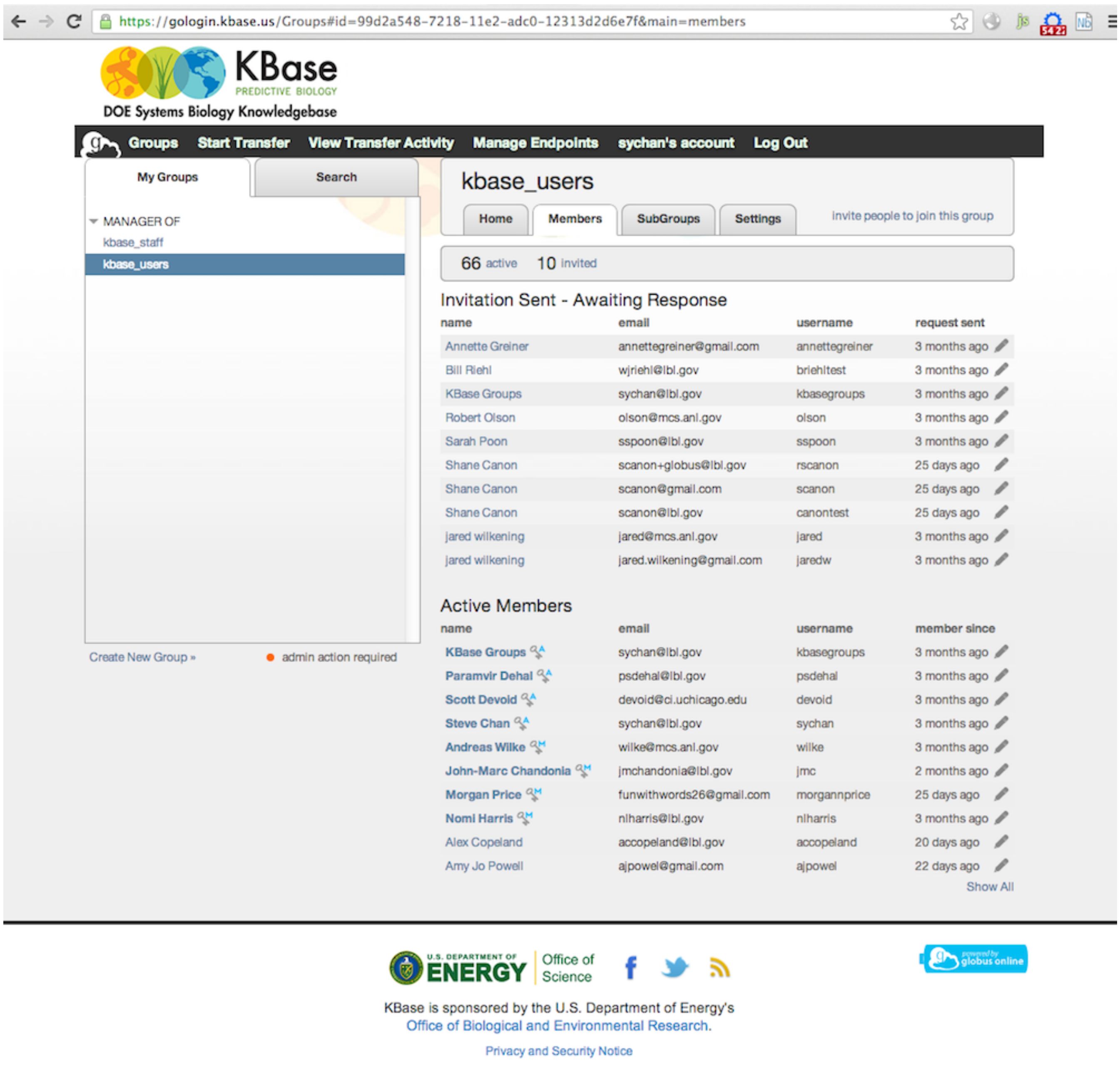
The overall design is in 2 major components:

1. The Globus Online Groups Service provides authentication and self-managed group membership. Globus Online manages a hierarchically structured set of groups, containing lists of users and some generic policies. Not actual authorization information is stored within Globus Online
2. An internal REST oriented service that associates a set of allowed operations on documents with a group of users. The groups of users are pulled from Globus Online and stored locally. This allows authz queries to operate independent of Globus Online, and cuts down on network latency. The permissions are stored as JSON documents in the mongodb.kbase.us database, allowing them to be distributed, scaled and tuned as part of the core infrastructure and opening up the possibility that they can be accessed directly using the appropriate libraries.


globus online
Group Member Sync (rest)
mongodb queries (pymongo)
kbase.us/services/ authorization
mongodb.kbase.us authorization collection
Updates Group Membership using http://gologin.kbase.us/Groups web GUI (web browser)
Query user's group membership and roles (rest)
User Interacts with service using token for authentication
kbase.us/services/ myapp

Globus Online Groups Management Interface

Globus Online provides a relatively mature, skinnable identity and group management interface at gologin.kbase.us In addition to the basic identity/authentication services we have used so far, they also provide a *self-service* groups management interface accessible at https://gologin.kbase.us/Groups



Invitation Sent - Awaiting Response

name	email	username	request sent
Annette Greiner	annettegreiner@gmail.com	annettegreiner	3 months ago
Bill Riehl	wjriehl@lbl.gov	briehltest	3 months ago
KBase Groups	sychan@lbl.gov	kbasegroups	3 months ago
Robert Olson	olson@mcs.anl.gov	olson	3 months ago
Sarah Poon	sspoon@lbl.gov	sspoon	3 months ago
Shane Canon	scanon+globus@lbl.gov	rscanon	25 days ago
Shane Canon	scanon@gmail.com	scanon	25 days ago
Shane Canon	scanon@lbl.gov	canontest	25 days ago
jared wilkening	jared@mcs.anl.gov	jared	3 months ago
jared wilkening	jared.wilkening@gmail.com	jaredw	3 months ago

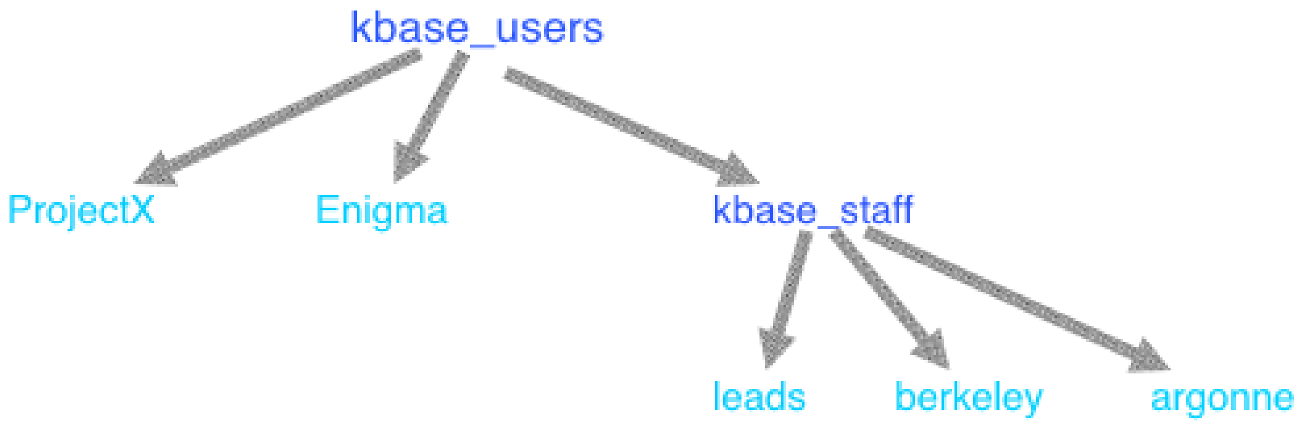
Active Members

name	email	username	member since
KBase Groups	sychan@lbl.gov	kbasegroups	3 months ago
Paramvir Dehal	psdehal@lbl.gov	psdehal	3 months ago
Scott Devoid	devoid@ci.uchicago.edu	devoid	3 months ago
Steve Chan	sychan@lbl.gov	sychan	3 months ago
Andreas Wilke	wilke@mcs.anl.gov	wilke	3 months ago
John-Marc Chandonia	jmchandonia@lbl.gov	jmc	2 months ago
Morgan Price	funwithwords26@gmail.com	morgannprice	25 days ago
Nomi Harris	nlharris@lbl.gov	nlharris	3 months ago
Alex Copeland	accopeland@lbl.gov	accopeland	20 days ago
Amy Jo Powell	ajpowel@gmail.com	ajpowel	22 days ago

KBase is sponsored by the U.S. Department of Energy's Office of Biological and Environmental Research.

Because Globus Online provides the authentication and group management interface as an independent service, KBase does not have **"root"** access to arbitrary groups managed by Globus Online - users manage their own group membership by browsing for groups, creating their own groups, requesting membership in existing groups, or inviting others to groups they control.

In this model, the KBase platform operates a tree of groups rooted at the "kbase_users" group. Once a user joins the kbase_users group, it is possible to administratively move/add/delete them from any child groups of kbase_users. Due to the decentralized model of Globus Online, it is not possible to draft/coerce an arbitrary Globus Online identity into kbase_users - membership must be consensual (at least, initially). In the diagram below, once a user has accepted membership in kbase_users, it is possible for the administrator/owner of this tree to add/remove the user to any child group - but not before they accept membership.



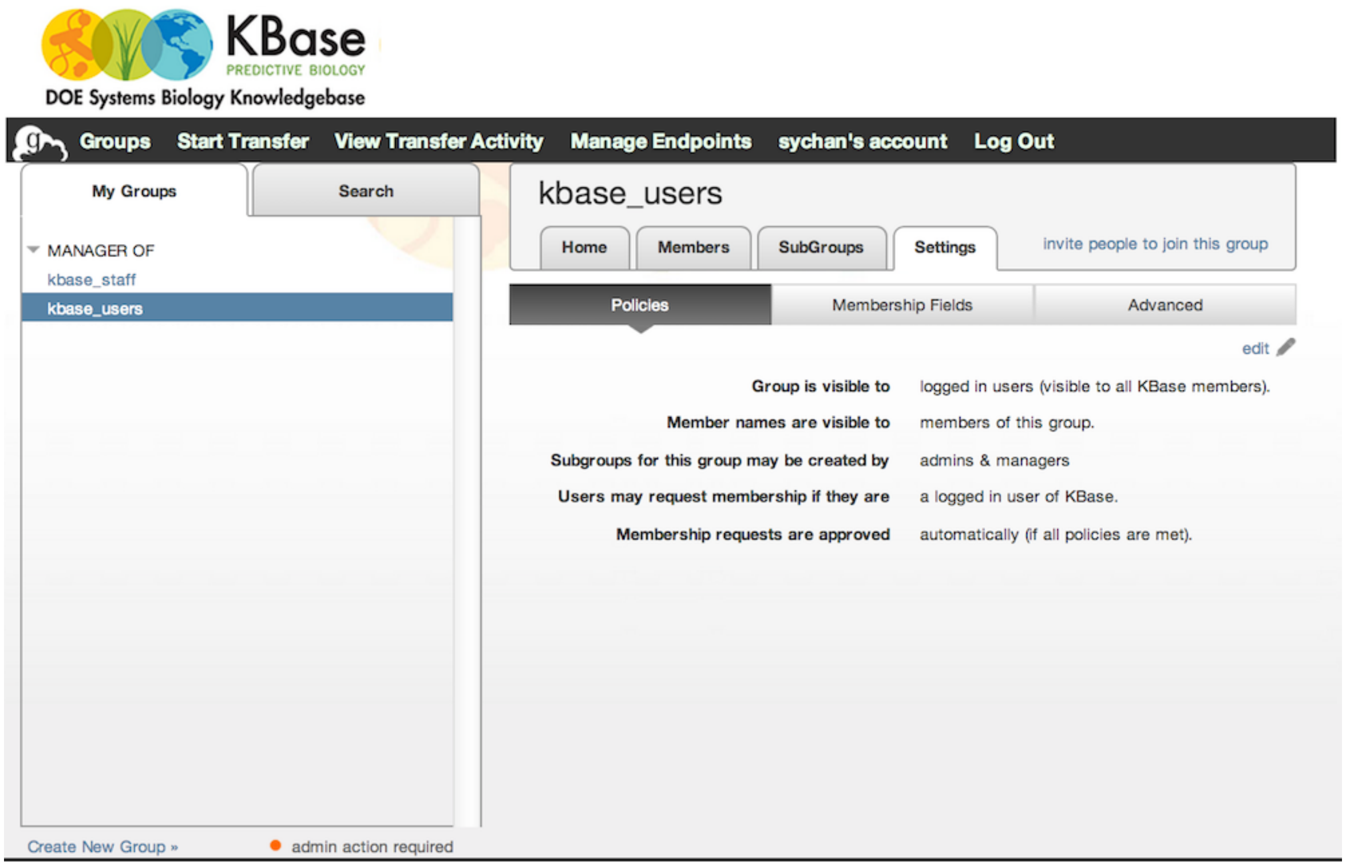
Note that in the diagram above, only the dark blue groups currently exist. The light blue groups are hypothetical groups to illustrate the hierarchy.

The workaround we have for this is to automatically enroll any user that registers and account at gologin.kbase.us in the kbase_users group. In addition, if a user logs in at gologin.kbase.us, they will be prompted to join kbase_users as well. This lowers the administrative overhead of bringing users into our groups tree.

The Globus groups system can be accessed via the browser interface shown above, as well as a REST API. Globus provides a python library that supports operations on the groups, in addition to direct access to the rest API with any http client library.

Membership in a group causes membership to propagate upwards in the tree. For example, if a user accepts an invitation to the argonne group show above, they would automatically be propagated into the kbase_staff and kbase_users group. Group membership does *not* propagate downward. So becoming a member of kbase_users does not result in membership to either ProjectX, Enigma or kbase_staff.

In addition, it is possible to set administrative properties such as member visibility, and the ability to create subgroups for individual groups.



Group is visible to	logged in users (visible to all KBase members).
Member names are visible to	members of this group.
Subgroups for this group may be created by	admins & managers
Users may request membership if they are	a logged in user of KBase.
Membership requests are approved	automatically (if all policies are met).

The exact layout of the tree is open to discussion based on appropriate use cases. However, the tree structure would probably create usability issues if the tree depth became too deep, so a flatter structure is recommended.

Internal KBase Authz Service

The internal KBase authz service stores assertions about what operations a set of users is allowed to perform on documents. The REST service is a python/django application at the endpoint https://kbase.us/services/authorization/Roles Access is authenticated by the usual Authorization: OAuth *kbase token* headers, and access is restricted to members of the kbase_users group. A browser can access it using a REST plugin (for Chrome I recommend Advanced Rest Client)

The service is documented in a Google Doc that is fairly complete. Note that the Roles defined within the KBase authz service are "flat", the hierarchy of the Globus Online Groups system is only reflected in membership inheritance, and in the names of Globus Groups used for automatically updating user rosters.

Structure of Roles

For a complete description please see the main API documentation. The brief introduction is that the Roles are JSON documents that map a set of users (possibly pulled from Globus Online) to operations on sets of documents. Here is an example:

```
{
    "role_updater": [
        "sychan",
        "kbauthorz",
        "thomasoniii",
        "devoid",
        "wilke",
        "teharrison"
    ],
    "description": "List of user ids who are considered KBase users",
    "read": [],
    "create": [],
    "modify": [],
    "role_owner": "sychan",
    "role_id": "kbase_users",
    "impersonate": [],
    "members": [
        "kbasetest",
        "sspoon",
        "kycl4rk",
        "seaver",
        "devoid",
        "ranantha",
        "kbauthorz",
        "landml",
        "psdehal",
        "kbasegroups",
        "wjriehl",
        "sychan",
        "annettegreiner",
        "thomasoniii",
        "nlharris",
        "wilke"
    ],
    "_id": "5069f456f43dc373bb677d94",
    "globus_group": "/kbase_users",
    "delete": []
}
```

This role is the placeholder at the root of the kbase groups tree, and merely contains a list of users, with no document associated with the *verbs* of:

- read
- create
- modify
- impersonate
- delete

The globus_group attribute is used to refer to a group in Globus Online that is polled to retrieve current group members. It is structured like a directory path rooted at /kbase_users.

Example Uses

In the first example, we show python code to acquire a Globus token using the python client libraries (using implicit ssh-agent authentication) and then we get a listing of the currently defined roles in the authz service.

Acquire a Token (Python version)

In [80]:
```
import requests
import biokbase.Auth
import os
import pprint
from IPython.core.display import Image

token = biokbase.Auth.Token( user_id = 'sychan')
s = requests.Session( headers = { 'Authorization' : 'OAuth ' + token.token,
                                  'content-type': 'application/json'})
```

List Currently Defined Roles

We use the session object **s** defined above to query the root of the REST service.

In [68]:
```
response = s.get( 'https://kbase.us/services/authorization/Roles')
print response.content
```

```
[
    "kbase_test",
    "kbase_test_users",
    "test_123",
    "sychan_test",
    "kbase_users2",
    "kbase_users",
    "kbase_staff"
]
```

Get Basic Self Documentation

The **about** URL parameter provides a short description of the service:

In [81]:
```
response = s.get( 'https://kbase.us/services/authorization/Roles?about')
print response.content
```

```
[
    {
        "documentation": "https://docs.google.com/document/d/1CTkthDUPwNzMF22maLyNIktI1sHdWPwtd3l
        "documentation2": "https://docs.google.com/document/d/1-43UvESzSYtLInqOouBE1s97a6-cRuPghb
        "contact": {
            "email": "sychan@lbl.gov"
        },
        "usage": "This is a standard REST service. Note that read handler takes MongoDB filtering
        "id": "KBase Authorization",
        "resources": {
            "role_updater": "User_ids that can update this role",
            "description": "Description of the role (required)",
            "grant": "List of kbase authz role_ids (strings) that this role allows grant privs",
            "read": "List of kbase object ids (strings) that this role allows read privs",
            "create": "Boolean value - does this role provide the create privilege",
            "modify": "List of kbase object ids (strings) that this role allows modify privs",
            "role_owner": "Owner(creator) of this role",
            "role_id": "Unique human readable identifer for role (required)",
            "impersonate": "List of kbase user_ids (strings) that this role allows impersonate pr
            "members": "A list of the user_ids who are members of this group",
            "owns": "List of document_ids that are owned by the role_owner and role_updaters",
            "globus_group": "Optional group path in Globus Online that is used to synchronize mem
            "delete": "List of kbase object ids (strings) that this role allows delete privs"
        }
    }
]
```

Examine a Role in Python, Perl

In the following section, we query the service for the contents of the kbase_users role using Python. We then use the %%perl magic of the iPython Notebook to run comparable sample code in Perl as an example for Perl developers.

In [63]:
```
response = s.get( 'https://kbase.us/services/authorization/Roles/kbase_users')
print response.content
```

```
[
    {
        "role_updater": [
            "sychan",
            "kbauthorz",
            "thomasoniii",
            "devoid",
            "wilke",
            "teharrison"
        ],
        "description": "List of user ids who are considered KBase users",
        "read": [],
        "create": [],
        "modify": [],
        "role_owner": "sychan",
        "role_id": "kbase_users",
        "impersonate": [],
        "members": [
            "kbasetest",
            "sspoon",
            "kycl4rk",
            "seaver",
            "devoid",
            "ranantha",
            "kbauthorz",
            "landml",
            "psdehal",
            "kbasegroups",
            "wjriehl",
            "sychan",
            "annettegreiner",
            "thomasoniii",
            "nlharris",
            "wilke"
        ],
        "_id": "5069f456f43dc373bb677d94",
        "globus_group": "/kbase_users",
        "delete": []
    }
]
```

In [66]:

```
%%perl
use Bio::KBase::AuthToken;
use REST::Client;
use Data::Dumper;

$t = Bio::KBase::AuthToken->new( user_id => 'sychan');
$client = REST::Client->new();
$client->setHost( "https://kbase.us/");
$client->addHeader( 'Authorization', 'OAuth ' . $t->token);
$client->GET('services/authorization/Roles/kbase_users');
print Dumper( $client->responseContent());
```

```
$VAR1 = &apos;[
    {
        "role_updater": [
            "sychan",
            "kbauthorz",
            "thomasoniii",
            "devoid",
            "wilke",
            "teharrison"
        ],
        "description": "List of user ids who are considered KBase users",
        "read": [],
        "create": [],
        "modify": [],
        "role_owner": "sychan",
        "role_id": "kbase_users",
        "impersonate": [],
        "members": [
            "kbasetest",
            "sspoon",
            "kycl4rk",
            "seaver",
            "devoid",
            "ranantha",
            "kbauthorz",
            "landml",
            "psdehal",
            "kbasegroups",
            "wjriehl",
            "sychan",
            "annettegreiner",
            "thomasoniii",
            "nlharris",
            "wilke"
        ],
        "_id": "5069f456f43dc373bb677d94",
        "globus_group": "/kbase_users",
        "delete": []
    }
]&apos;;
```

Creating a new role for an new KBase object

It is often appropriate to create a new role to handle access control for a newly created object. We use the POST method to create a new role called "kb_ws_species8472" for a new workspace object called "kb|ws.species8472". Note that any correspondence between a role name in the authz service and object ID's in other services are purely a matter of convention at this point - it can be possible to enforce rules once such rules are determined.

In [98]:
```
newrole = { "role_updater": ["sychan"],
            "description": "Role for workspace object Species8472",
            "read": [ 'kb|ws.species8472'],
            "create": [ 'kb|ws.species8472'],
            "modify": [ 'kb|ws.species8472'],
            "grant" : ['kb|ws.species8472' ],
            "role_owner": "sychan",
            "role_id": "kb_ws_species8472",
            "impersonate": [],
            "members": ["sychan","kbasetest","psdehal"],
            "delete": [ 'kb|ws.species8472' ],
            "owns": ['kb|ws.species8472'],
            "globus_group": ""
           }
import json
response = s.post( url='https://kbase.us/services/authorization/Roles', data=json.dumps(newrole))
print response.content
```

```
Created
```

Querying for all roles that contain a user

A common use is to query the service for all the roles that include a user. The API includes some shortcuts using URL variables to support this and other common operations.

In [102]:
```
# Query for all roles that contain user 'sychan'
response = s.get( 'https://kbase.us/services/authorization/Roles?user_id=sychan')
print response.content
```

```
[
    {
        "role_updater": [
            "sychan",
            "kbauthorz"
        ],
        "description": "Steve&apos;s test role",
        "read": [],
        "create": [],
        "modify": [],
        "role_owner": "sychan",
        "role_id": "sychan_test",
        "impersonate": [],
        "members": [
            "wilke",
            "psdehal",
            "nlharris",
            "sychan",
            "kbasegroups"
        ],
        "_id": "5069f4f1f43dc373bb677d95",
        "delete": []
    },
    {
        "role_updater": [
            "sychan",
            "kbauthorz",
            "thomasoniii",
            "devoid",
            "wilke",
            "teharrison"
        ],
        "description": "List of user ids who are considered KBase users",
        "read": [],
        "create": [],
        "modify": [],
        "role_owner": "sychan",
        "role_id": "kbase_users2",
        "impersonate": [],
        "members": [
            "kbasetest",
            "landml",
            "kbauthorz",
            "sspoon",
            "psdehal",
            "kbasegroups",
            "kycl4rk",
            "seaver",
            "devoid",
            "annettegreiner",
            "sychan",
            "nlharris",
            "wilke",
            "ranantha",
            "thomasoniii"
        ],
        "_id": "512418de90f4719aaa333c0a",
        "delete": []
    },
    {
        "role_updater": [
            "sychan",
            "kbauthorz",
            "thomasoniii",
            "devoid",
            "wilke",
            "teharrison"
        ],
        "description": "List of user ids who are considered KBase users",
        "read": [],
        "create": [],
        "modify": [],
        "role_owner": "sychan",
        "role_id": "kbase_users",
        "impersonate": [],
        "members": [
            "kbasetest",
            "sspoon",
            "kycl4rk",
            "seaver",
            "devoid",
            "ranantha",
            "kbauthorz",
            "landml",
            "psdehal",
            "kbasegroups",
            "wjriehl",
            "sychan",
            "annettegreiner",
            "thomasoniii",
            "nlharris",
            "wilke"
        ],
        "_id": "5069f456f43dc373bb677d94",
        "globus_group": "/kbase_users",
        "delete": []
    },
    {
        "role_updater": [
            "sychan"
        ],
        "description": "Role for workspace object Species8472",
        "grant": [
            "kb|ws.species8472"
        ],
        "read": [
            "kb|ws.species8472"
        ],
        "create": [
            "kb|ws.species8472"
        ],
        "modify": [
            "kb|ws.species8472"
        ],
        "role_owner": "sychan",
        "role_id": "kb_ws_species8472",
        "impersonate": [],
        "members": [
            "kbasetest",
            "psdehal",
            "sychan"
        ],
        "owns": [
            "kb|ws.species8472"
        ],
        "_id": "518a98cf7bc3ca5433df7570",
        "globus_group": "",
        "delete": [
            "kb|ws.species8472"
        ]
    }
]
```

Query for user and document ID

Another common use case is to find what privs a user has on a particular object. Let's add an additional filtering clause to the previous query and say we only want to see the roles for sychan and object names 'kb_ws_species8472'. We show the version with all the individual roles, but it is possible to request a union of all the roles into a super-role that is a single document with all rights merged in.

In [103]:
```
# Query for all roles that contain user 'sychan' and referencing "kb|ws.specieas8472"
params = { 'user_id' : 'sychan',
           'doc_id' : 'kb|ws.species8472' }
response = s.get( 'https://kbase.us/services/authorization/Roles', params = params)
print response.content
```

```
[
    {
        "role_updater": [
            "sychan"
        ],
        "description": "Role for workspace object Species8472",
        "grant": [
            "kb|ws.species8472"
        ],
        "read": [
            "kb|ws.species8472"
        ],
        "create": [
            "kb|ws.species8472"
        ],
        "modify": [
            "kb|ws.species8472"
        ],
        "role_owner": "sychan",
        "role_id": "kb_ws_species8472",
        "impersonate": [],
        "members": [
            "kbasetest",
            "psdehal",
            "sychan"
        ],
        "owns": [
            "kb|ws.species8472"
        ],
        "globus_group": "",
        "delete": [
            "kb|ws.species8472"
        ]
    }
]
```

Delete a role for a defunct object

So, species8472 disappears from the workspace. Lets delete the role.

In [104]:
```
response = s.delete( url='https://kbase.us/services/authorization/Roles/kb_ws_species8472')
print response.status_code
```

```
204
```

Conclusion

There are a lot of additional features available in the API docs, but the usage is basically the same as the examples given. The remaining issues that need to be resolved are:

- a common namespace for KBase objects so that we don't have collisions and/or ambiguity
- a convention for how we want to manage the tree of user groups
- additional features to support actual use cases
- tweaking the REST API so that it is more featureful in terms of REST support. It is currently REST-ish and not fully rest compliant