Federation Service API's

Marshall Brinn, GENI Program Office, USA
Jonathon Duerig, University of Utah, USA
Aaron Helsinger, GENI Program Office, USA
Tom Mitchell, GENI Program Office, USA
Rob Ricci, University of Utah, USA
Tom Rothe, EICT, Germany
Leigh Stoller, University of Utah, USA
Wim Van de Meerssche, iMinds/Ghent University, Belgium
Brecht Vermeulen, iMinds/Ghent University, Belgium
Gary Wong, University of Utah, USA

API Version 2

Revised: November 13, 2013
Converted to AsciiDoc: September, 2016

Introduction

This document proposes a set of standard API's that any GENI-compatible Federation should or may provide. The document describes what is required and what is optional in the calls and responses to these APIs.

The GENI Software Architecture rests on the interaction between different entities:

- **Aggregates** : Collections of resources managed and presented in accordance with the AM API

- **Members** : Experimenters or other human consumers of aggregate resources

- **Authorities** : Services that manage assertions about members and their permissions with respect to aggregate resources.

There are two fundamental authority types in GENI:

- *Member Authority* [MA]: Manages and asserts attributes about particular members

- *Slice Authority* [SA]: Manages slice objects and generates credentials for members with respect to slices.

A ' Federation ' is a collection of Authorities and Aggregates that establish mutual trust and common policies to facilitate the sharing of resources among members. A Federation Registry is a software service representing a given Federation, providing lists of Slice Authorities, Member Authorities and aggregates associated with that federation, and providing a set of PKI certificates that any aggregate belonging to a given federation accepts as trust roots. The relationship of Federations to Federation Registries is 1:1.

The Authorities of any given Federation are free to implement their own Authorization (AuthZ) scheme. The API's allow for passing credentials to the calls, but an Authority may choose to allow or disallow calls using logic and policies that are internal to that Federation. There is no universal

(cross-Federation) requirement for any particular policy regarding Authority AuthZ.

Authorities are fundamentally independent of one another. The objects defined at one Authority are not necessarily entitled to any services provided by another Authority. Each aggregate may choose to trust or not trust any particular Authority. Likewise, any Authority may chose to trust or not trust any other Authority. A Federation Registry may choose to advertise or not advertise any particular aggregate, regardless of whether that aggregate trusts the Authorities advertised by that Federation Registry. Similarly, a given Slice Authority or Member Authority may be advertised by a single Federation Registries or by multiple Federation Registries. Federation Registry API calls are unprotected. There is no notion of trust between Federation Registries or between Federation Registries and Authorities or Aggregates.

This document describes the APIs of the Federation Registry as well as the MA and SA. It is expected that a well-behaved GENI-compatible tool will allow for interacting with any Federation Registry and Authority that implement the standard API's described in this document.

API General Properties

The APIs described here share some common properties, which should be assumed for the rest of this document:

- The wire-protocol is XML/RPC. It is thus language independent on both client and server side of the API calls.

- Most calls are protected, running over SSL and thus requiring the caller to use its certificate and private key. Certain calls are unprotected and can be accessed with no requirement for a validated client-side certificate . Such calls will noted in the API documentation below.

- Each call takes an 'options' argument, a dictionary allowing for passing specific non-standard/optional arguments

- Each protected method takes a 'credentials' argument, a list of type/credential tuples that help the Federation Registry or Authority invoke whatever AuthZ logic it may choose. As noted above, the Federation Registry or Authority may choose to use or disregard these credentials. Unprotected methods do not take a 'credentials' argument.

- Each Federation Registry or Authority provides a get_version method, which describes the version number of the API provided, credential types supported, supplementary object fields and other data for interpreting API call returns.

- A Federation Registry or Authority is free to implement additional methods beyond those specified in this document.

- The URN is the fundamental identifier in all Federation API's. URN's are globally unique at any given time, though not necessarily unique over time. Disambiguation for entities with the same URN over time may be provided by an optional UUID argument for certain API methods. The format of URN's is documented at http://groups.geni.net/geni/wiki/GeniApiIdentifiers.

These API's are provided in pseudo-code (i.e. language independent) format, with inputs and outputs (optional and required) described by comments, e.g.

An example

```python
#!python
# Perform method fed_func
#
# Arguments:
#  argl : ...
#  credentials : list of {type : credential} tuples representing credentials
#      provided by caller to support AuthZ on method call.
#       [NB: This argument will be omitted in descriptions below. ]
#  options : … [ Recognized options: ….]
#
# Return:
#  E.g. a list of dictionaries with these fields mandatory (…)
#                                      and these fields optional (…)
def fed_func (arg1, arg2, credentials, options)
```

API 'get_version' methods

Different Federation Authorities will provide different sets of methods bundled into services. Further, they will manage different kinds of objects and support different details for these objects.

Each Federation Registry or Authority API provides a 'get_version' method, which provides information to the caller (or a tool composing calls for a tool user) about versions and options supported by that API. The call takes no argument and is unguarded (anyone can call it). The return from the get_version call will be a dictionary including the following entries (by key):

- VERSION: A string with the version number of the Federation API (e.g. "2", the version for this document). Note: this is the version of the API not the version of the implementation. This field is mandatory for all services.

- URN : The URN of the service being contacted. This field is mandatory for SA and MA services, optional for Federation Registry service.

- IMPLEMENTATION: A dictionary of information of the implantation of the service: {"code_version" : code_version, "code_url" : code_url, "code_release_date" : code_release_date, "site_update_date" : site_update_date}. Of these, code_version is of type STRING, code_url is of type URL, code_release_date and site_update_date are of format DATETIME. The format of the code_version string is implementation specific. This field is optional for services; in addition, all the sub-fields for the IMPLEMENTATION field are optional.

- SERVICES: The list of names of services the given URL supports. This field is optional (with default being the default service for that authority, i.e. SERVICE for Federation Registry, SLICE for Slice Authority, MEMBER for MemberAuthority).

- CREDENTIAL_TYPES: A list of recognized credential types (e.g. [geni_sfa, geni_abac]) and list of supported credential versions on protected API methods. Format is analogous to that in the AM API: a list of {"type": cred_type, "version" : cred_version} dictionaries of all supported credential types and versions. "[Required for Authorities only]"

- ROLES : A list of recognized roles for slice/project membership (required only for those Slice

Authorities supporting membership). The same set of roles refers to both slice and project membership at a given SA.

- SERVICE_TYPES. A list of service types provided by the Federation Registry "[Required for Federation Registry only]"

- API_VERSIONS A dictionary of different peer implementation of different version of the same service. Modeled on the Aggregate Manager API, the format of this field is {version1 : url1, version2 : url2, ...}. This field is required for all services. Note that the 'self' version (the version of the service being queried) is required to be included in this dictionary and should be consistent with the "VERSION" field above. The URL's in this field

- FIELDS: A dictionary of object field names (i.e. in additional to the required fields) and associated attributes including:

  - "OBJECT" provides the object type to which the field belongs. The field is optional for fields of the default authority object (i.e. SLICE for Slice Authority, MEMBER for Member Authority, SERVICE for Federation Registry) but mandatory for all other fields.

  - "TYPE" may be one of "URN", "UID", "STRING", "DATETIME", "EMAIL", "KEY","BOOLEAN", "CREDENTIAL", "CERTIFICATE". [NB. This set of types subject to change. See Appendix for more information on these data types.] This field is mandatory for any field listed.

  - "CREATE" attributes may be specified as "REQUIRED", "ALLOWED" or "NOT ALLOWED" (default = "NOT ALLOWED"). These indicate whether the given supplementary field is required, allowed or prohibited in create calls. This attribute is optional for listed fields.

  - "MATCH" attributes may be specified as booleans TRUE or FALSE (default = TRUE). These indicate whether a given field may be specified in an match option of a lookup call. This attribute is optional for listed fields.

  - "UPDATE" attributes may be specified as booleans TRUE or FALSE (default = FALSE). These indicate whether the given field may be specified in an update call. This attribute is optional for listed fields.

  - "PROTECT" attributes may be labeled as "PUBLIC", "PRIVATE" or "IDENTIFYING". These are for the Member Authority only to differentiate between public, identifying and private data fields on members. The default, if not provided, is "PUBLIC", and thus this attribute is optional.

The FIELDS element of the get_version should contain all supplementary (non-mandatory) field objects supported by a given service. Additionally, it may contain mandatory field objects for which the default semantics (for "CREATE", "MATCH", "UPDATE", "PROTECT") should be overridden. Specifically, any values specified override the default values and any values unspecified are defined to be the defaults for that object/field in this document. The FIELDS element is thus optional for all services.

The set of ROLES may vary across Slice Authorities based on local policy. However, the following roles should be defined at any Slice Authority:

Role	Contex	Description
LEAD	PROJECT	May change project membership and create slices within a given project
	SLICE	May change slice membership and perform operations on a given slice
MEMBER	PROJECT	May create slices within given project
	SLICE	May perform operations on given slice

Supplementary field names should be placed in a distinct namespace by a prefix unique to that federation, and starting with an underscore (e.g. *GENI*, *OFELIA* , *FED4FIRE* or *PROTOGENI* etc.).

The API_VERSIONS field of the get_version should contain a dictionary specifying different URL's implementing different versions of the same service. The URL's provided should be absolute, containing publicly accessible addresses. This information may be used by the Federation Registry to provide SERVICE_PEERS information described below. An example API_VERSIONS field from a get_version call:

```
"API_VERSIONS": {
    "1" : "https://example.com/xmlrpc/sa/1",
    "2" : "https://example.com/xmlrpc/sa/2"
}
```

The return from the get_version call will be used to construct and validate options to Federation Registry and Authority API calls, as described in subsequent sections.

The get_version method at any service has the following signature:

```
#!python
# Return information about version and options
#   (e.g. filter, query, credential types) accepted by this service
#
# Arguments: None
#
# Return:
#    get_version structure information as described above
def get_version()
```

The following page provides some example returns from different get_version calls.

Example get_version returns:

The following is an example of a return from a get_version for an SA. The responses are all dictionaries via XMLRPC into the native implementation. They are shown here in JSON-like syntax:

```
{
  "VERSION": "2",
  "URN" : urn:publicid:IDN+example.com+authority+sa",
  "SERVICES": ["SLICE", "PROJECT", "SLICE_MEMBER", "PROJECT_MEMBER"],
  "OBJECTS": [ "PROJECT" ],
  "CREDENTIAL_TYPES": [{"type" : "geni_sfa", version" : 2}, {"type" : "geni_sfa",
"version" : "3"}, {"type" : "geni_abac", "version" : "1"}]
  "ROLES" : ["LEAD", "ADMIN", "MEMBER", "AUDITOR", "OPERATOR" ],
  "FIELDS": {
    "_GENI_PROJECT_UID": {"TYPE" : "UID", "UPDATE" : FALSE},
    "_GENI_SLICE_EMAIL": {"TYPE": "EMAIL", "CREATE": "REQUIRED", "UPDATE": TRUE},
    "_GENI_PROJECT_EMAIL": {"TYPE": "EMAIL", "CREATE": "REQUIRED", "UPDATE": TRUE,
"OBJECT": "PROJECT"}
  }
}
```

The following is an example of a return from a get_version for an MA, provided in JSON-like syntax:

```
{
    "VERSION": "2",
    "URN" : urn:publicid:IDN+example.com+authority+ma",
    "CREDENTIAL_TYPES":  [{"type" : "geni_sfa", version" : 2},
                          {"type" : "geni_sfa", "version" : "3"},
                          {"type" : "geni_abac", "version" : "1"}]
    "SERVICES": ["MEMBER", "KEY"],
    "OBJECTS": [ "KEY" ],
    "FIELDS": {
        "MEMBER_DISPLAYNAME": {"TYPE": "STRING",
                              "CREATE": "ALLOWED",
                              "UPDATE", TRUE,
                              "PROTECT": "IDENTIFYING"},
        "MEMBER_AFFILIATION": {"TYPE": "STRING",
                              "CREATE": "ALLOWED",
                              "UPDATE": TRUE,
                              "PROTECT": "IDENTIFYING"},
        "MEMBER_SSL_PUBLIC_KEY": {"TYPE": "SSL_KEY"},
        "MEMBER_SSL_PRIVATE_KEY": {"TYPE": "SSL_KEY",
                                   "PROTECT": "PRIVATE"},
        "MEMBER_SSH_PUBLIC_KEY": {"TYPE": "SSH_KEY"},
        "MEMBER_SSH_PRIVATE_KEY": {"TYPE": "SSH_KEY",
                                   "PROTECT": "PRIVATE"},
        "MEMBER_ENABLED": {"TYPE": "BOOLEAN",
                          "UPDATE": TRUE}
    }
}
```

The following is an example of a return from a get_version from a Federation Registry, provided in JSON-like syntax:

```
{
    "VERSION": "2",
    "URN" : urn:publicid:IDN+example.com+authority+fr",
    "SERVICE_TYPES" : ["SLICE_AUTHORITY", "MEMBER_AUTHORITY", "AGGREGATE_MANAGER"],
    "FIELDS": {
        "SERVICE_PROVIDER": {"TYPE": "STRING"}}
    }
}
```

API Error Handling

All method calls return a tuple [code, value, output]. What is described as 'Return' in the API's described below is the 'value' of this tuple in case of a successful execution. 'Code' is the error code returned and 'output' is the returned text (e.g. descriptive error message).

Each Federation Registry and Authority is free to define and return its own specific error codes.

However we suggest the following essential set of error codes to report on generic conditions:

CODE_NAME	CODE_VALUE	DESCRIPTION
NONE	0	No error encountered – the return value is a successful result. An empty list from a query should be interpreted as 'nothing found matching criteria'.
AUTHENTICATION_ERROR	1	The invoking tool or member did not provide appropriate credentials indicating that they are known to the Federation or that they possessed the private key of the entity they claimed to be
AUTHORIZATION_ERROR	2	The invoking tool or member does not have the authority to invoke the given call with the given arguments
ARGUMENT_ERROR	3	The arguments provided to the call were mal-formed or mutually inconsistent.
DATABASE_ERROR	4	An error from the underlying database was returned. (More info should be provided in the 'output' return value]
DUPLICATE_ERROR	5	An error indicating attempt to create an object that already exists
NOT_IMPLEMENTED_ERROR	100	The given method is not implemented on the server.
SERVER_ERROR	101	An error in the client/server connection

Standard API Method

Each Federation Registry and Authority manages the state of or access to objects. There are some standard methods that apply to standard operations on objects of specific types. All services support the following API's for the object types that are required or provided in get_version.

```python
#!python
# Creates a new instance of the given object with a 'fields' option
# specifying particular field values that are to be associated with the object.
# These may only include those fields specified as 'ALLOWED or 'REQUIRED'
# in the 'Creation' column of the object descriptions below
# or in the "CREATE' key in the supplemental fields in the
# get_version specification for that object.
# If successful, the call returns a dictionary of the fields
# associated with the newly created object.
#
#
# Arguments:
#
#    type : type of object to be created
#   options:
#       'fields', a dictionary field/value pairs for object to be created
#
# Return:
#   Dictionary of object-type specific field/value pairs for created object
#
#
def create(type, credentials, options)
```

```python
#!python
# Updates an object instance specified by URN with a 'fields' option
#  specifying the particular fields to update.
# Only a single object can be updated from a single update call.
# The fields may include those specified as 'Yes' in the 'Update' column
# of the object descriptions below, or 'TRUE' in the 'UPDATE' key in the
# supplemental fields provided by the get_version call.
# Note: There may be more than one entity of a given URN at an authority,
# but only one 'live' one (any other is archived and cannot be updated).
#
# Arguments:
#   type: type of object to be updated
#   urn: URN of object to update
#     (Note: this may be a non-URN-formatted unique identifier e.g. in the case of
keys)
#   options: Contains 'fields' key referring dictionary of
#        name/value pairs to update
#
# Return: None
#
def update(type, urn, credentials, options)
```

```python
#!python
# Deletes an object instance specified by URN
# Only a single object can be deleted from a single delete call.
# Note: not all objects can be deleted. In general, it is a matter
#     of authority policy.
#
# Arguments:
#   type: type of object to be deleted
#   urn: URN of object to delete
#     (Note: this may be a non-URN-formatted unique identifier e.g. in the case of
keys)
#
# Return: None
#
def delete(type, urn, credentials, options)
```

```python
#!python
# Lookup requested details for objects matching 'match' options.
# This call takes a set of 'match' criteria provided in the 'options' field,
# and returns a dictionary of dictionaries of object attributes
# keyed by object URN matching these criteria.
# If a 'filter' option is provided, only those attributes listed in the 'filter'
# options are returned.
# The requirements on match criteria supported by a given service
# are service-specific; however it is recommended that policies
# restrict lookup calls to requests that are bounded
# to particular sets of explicitly listed objects (and not open-ended queries).
#
# See additional details on the lookup method in the document section below.
#
#
# Arguments:
#    type: type of objects for which details are being requested
#    options: What details to provide (filter options)
#            for which objects (match options)
#
# Return: List of dictionaries (indexed by object URN) with field/value pairs
#   for each returned object
#
def lookup (type, credentials, options)
```

Some additional details on the lookup call:

The options argument to the lookup call is a dictionary. It contains an entry with key 'match' that contains a dictionary of name/value pairs. The names are of fields listed in the get_version for that object. The values are values for those fields to be matched. The semantics of the match is to be an "AND" (all fields must match).

The value in the dictionary of a 'match' option can be a list of scalars, indicating an "OR". For example, a list of URNs provided to the SLICE_URN key would match any slice with any of the listed URNs.

The options argument may include an additional dictionary keyed "filter" which is a list of fields associated with that object type (again, as specified in the get_version entry for that object). No "filter" provided means all fields are to be returned; a 'filter' provided with an empty list returns an empty set of fields (i.e. a dictionary of URN's pointing to empty dictionaries).

The return of the call will be a dictionary of dictionaries, one per matching object indexed by URN, of fields matching the filter criteria. If the query found no matches, an empty dictionary is returned (i.e. no error is reported, assuming no other error was encountered in processing).

If a lookup method call requests information in the 'match' criteria about objects whose disclosure is prohibited to the requester by policy, the call should result in an authorization error. If the 'filter' criteria requests fields whose disclosure is prohibited to the requestor by policy, the method must not return the specific data fields. Rather, it should return a dictionary with no entry for the prohibited fields. E.g. {"urn_1" : {"PUBLIC_KEY" : public_key_1, "PRIVATE_KEY" : private_key_1}, "urn_2" : {"PUBLIC_KEY" : public_key_2}}

API Method Examples:

A Member Authority (MA) manages information about member objects. The MA method lookup(type="MEMBER") could take an options argument such as

```
{
    "match": {"MEMBER_LASTNAME": "BROWN"},
    "filter": ["MEMBER_EMAIL", "MEMBER_FIRSTNAME"]
}
```

Such a call would find any member with last name Brown and return a dictionary keyed by the member URN containing a dictionary with their email, and first name.

```
{
    "urn:publicid:IDN+mych+user+abrown" :
        {"MEMBER_EMAIL": abrown@williams.edu,
         "MEMBER_FIRSTNAME": "Arlene"},

    "urn:publicid:IDN+mych+user+mbrown" :
        {"MEMBER_EMAIL": mbrown@umass.edu,
         "MEMBER_FIRSTNAME": "Michael"},

    "urn:publicid:IDN+mych+user+sbrown" :
        {"MEMBER_EMAIL": sbrown@stanford.edu,
         "MEMBER_FIRSTNAME": "Sam"}
}
```

A Slice Authority (SA) manages information about slice objects. The SA method update(type="SLICE") could take the following options argument to change the slice description and extend the slice expiration:

{

```
  "fields" : { "SLICE_DESCRIPTION": "Updated Description",
```

```
  "SLICE_EXPIRATION": "2013-07-29T13:15:30Z" }
```

}

An example of lookup(type="SLICE)" at an SA that wanted to retrieve the slice names for a list of slice URNs could specify options:

{ "match": {

```
  "SLICE_URN": [
```

```
  "urn:publicid+IDN+this_sa:myproject+slice+slice1",
```

```
  "urn:publicid+IDN+this_sa:myproject+slice+slice2",
```

```
  "urn:publicid+IDN+this_sa:myproject+slice+slice3"
```

```
  ]},
```

```
  "filter": ["SLICE_NAME"]
```

}

API Method Examples (cont.):

An example of create(type="SLICE") call would specify required options e.g.:

{

```
  'fields' : {
```

```
  "SLICE_NAME": "TEST_SLICE",

  "SLICE_DESCRIPTION": "My Test Slice",

  "SLICE_EMAIL": myemail@geni.net,

  "SLICE_PROJECT_URN": "urn:publicid+IDN+this_sa+project+myproject"

  }

}
```

and receive a return dictionary looking like:

```
{

  "SLICE_URN": "urn:publicid+IDN+this.sa+slice+TESTSLICE",

  "SLICE_UID": "···",

  "SLICE_NAME": "TESTSLICE",

  "SLICE_CREDENTIAL": ".....",

  "SLICE_DESCRIPTION": "My Test Slice",

  "SLICE_PROJECT_URN": "urn:publicid+IDN+this_sa+project+myproject",

  "SLICE_EXPIRATION": "2013-08-29T13:15:30Z",

  "SLICE_EXPIRED": "FALSE",

  "SLICE_CREATION": "2013-07-29T13:15:30Z",
```

```
    "SLICE_EMAIL": myemail@geni.net
```

}

API Authentication

This document suggests that the Authentication required for the Federation APIs is implicit in the SSL protocol: the invoker of the call must have its cert and private key to have a valid SSL connection. Moreover, the cert must be signed by a member of the trust chain recognized by the Federation.

Support for Speaks-for API Invocations

Best practices dictate that individuals should speak as themselves: that is, the entity on the other side of an SSL connection is the one referred to by the certificate on the connection. Obviously, people typically use tools or software interfaces to create these connections. When a tool is acting directly on a user's desktop using the user's key and cert with the user's explicit permission, it may be acceptable to consider the tool as speaking as the user. But for many tools, the tool is acting on behalf of the user in invoking Federation or AM API calls. In this case, it is important for the tool to not speak as the user but to speak for the user, and to have the service to whom the tool is speaking handle the authorization and accountability of this request accordingly.

Accordingly, a Federation Registry and associated Authorities should support speaks-for API transactions. These API transactions use the same signatures as the calls described in this document, with these enhancements:

- A 'speaking_for' option containing the URN of the user being spoken for

- A speaks-for credential in the list of credentials: a statement signed by the user indicating that the tool has the right to speak for the user, possibly limited to a particular scope (e.g. slice, project, API call, time window).

The service call is then required to determine if the call is being made in a speaks-for context or not (that is, the 'speaking_for' option provided). If so, the call must determine if the tool is allowed to speak for the user by checking for the presence of a valid speaks-for credential and the spoken-for user's cert. If so, the call should validate if the user is authorized to take the proposed API action. If so, the action is taken and accounted to the user, with identity of the speaking_for tool logged. If the call is 'speaks-for' but any of these additional criteria are not met, the call should fail with an authorization error. If the call is not a 'speaks-for', then the normal authorization is performed based on the identity (certificate) provided with the SSL connection.

Aggregates are also encouraged to support speaks-for authentication and authorization, but this is an aggregate-internal policy and implementation decision, and outside the scope of this document.

Federation Registry API

The Federation Registry provides a list of Slice Authorities, Member Authorities and Aggregates associated with a given Federation. The URL for accessing these methods (i.e. the URL of the Federation Registry) is to be provided out-of-band (i.e. there is no global service for gaining access to Federation Registry addressees).

All Federation Registry calls are unprotected; they have no requirement for passing a client-side cert or validating any client-cert cert that is passed.

The Federation Registry implements the SERVICE service and supports the SERVICE object.

Services have a particular type that indicates the kind of service it represents. The full list of supported services should be provided by a TYPES key in the Federation Registry get_version call, for example:

{{{ { ... "SERVICE_TYPES" : ["SLICE_AUTHORITY", "MEMBER_AUTHORITY", "AGGREGATE_MANAGER", ...] ... } }}}

This table contains a set of "example" services types (of which only SLICE_AUTHORITY, MEMBER_AUTHORITY and AGGREGATE_MANAGER are required for any given federation):

Service	Description
SLICE_AUTHORITY	An instance of the Slice Authority Federation service described in this document
MEMBER_AUTHORITY	An instance of the Member Authority Federation service described in this document
AGGREGATE_MANAGER	An instance of an Aggregate Manager satisfying the Aggregate Manager API
STITCHING_COMPUTATION_SERVICE	A topology service for supporting cross-aggregate stitching
CREDENTIAL_STORE	A service holding credentials for the federation, typically for supporting federation authentication services
LOGGING_SERVICE	A service to support federation-level event logging

The following table describes the standard fields for services (aggregates and authorities) provided by Federation Registry API calls. (The 'Required' column indicates whether the field must be present for a valid service, 'match' indicates whether the field can be used in a lookup match criterion):

Name	Type	Description	Required	Match
SERVICE_URN	URN	URN of given service	Yes	Yes
SERVICE_URL	URL	URL by which to contact the service	Yes	Yes
SERVICE_TYPE	STRING	Name of service type (from Federation Registry get_version.TYPES)	Yes	Yes
SERVICE_CERT	Certificate	Public certificate of service	No	No
SERVICE_NAME	String	Short name of service	Yes	No
SERVICE_DESCRIPTION	String	Descriptive name of service	No	No
SERVICE_PEERS	List of Dictionaries	URLs and version info for other running version of same service (see below)	No	No

The SERVICE_PEERS field is similar to that in the AM API: a list of {version", 'url'} dictionaries for other supported peer services of different versions. It is provided to allow a user/tool to determine which URL to contact without needing to poll the get_version call across a set of services. The current service URL (provided in the SERVICE_URL field) should always be included in the

SERVICE_PEERS. The information provided by SERVICE peers should be consistent with that provided by the API_VERSIONS field from the get_version call to these specific services. An example would be as follows: {{{ [{'version' : '1', 'url' : 'https://example.com/xmlrpc/v1'}, {'version' : '2', 'url' : 'https://example.com/xmlrpc/v2'},]

}}}

The Federation Registry API supports these standard API methods for type="SERVICE":

|| '''Method ''' || ''' Description ''' || || lookup || lookup services matching given match criteria. ||

Note that even though the Federation Registry API does not require authentication and thus no client certificates, the API uses the common API signatures for all 'lookup' methods and thus takes a list of credentials. This list, however, should be empty and ignored by the implementation.

Additionally, the Federation Registry API supports the following methods:

{{{ #!python # Return list of trust roots (certificates) associated with this Federation. # # Often this is a concatenatation of the trust roots of the included authorities. # Note: Some of this information can be retrieved by # lookup(fields={"SERVICE_CERT"}) # However certificates of federation-level certs, certificate authorities or other # non-service certificate signers can only be retrieved through this call. # # Arguments: # None # # Return: # List of certificates representing trust roots of this Federation. def get_trust_roots() }}}

{{{ #!python # Lookup the authorities for a given URNs # # There should be at most one (potentially none) per URN. # # This requires extracting the authority from the URN and then looking up the authority in the Federation Registry's set of services. # # Arguments: # urns: URNs of entities for which the authority is requested # # Return: # List of dictionaries {urn : url} mapping URLs of Authorities to given URN's def lookup_authorities_for_urns(urns) }}}

The ''lookup_authorities_for_urns'' method maps object URN's to authority URN's. Note that the transformation from the URN's of objects (e.g. slice, project, member) to the URN's of their authority is a simple one, for example:

|| '''Type''' || '''Object URN''' || '''Authority URN''' || || Slice || urn:publicid:IDN+sa_name+slice+slice_name || urn:publicid:IDN+sa_name+authority+sa || || Member || urn:publicid:IDN+ma_name+user+user_name || urn:publicid:IDN+ma_name+authority+ma ||

Slice Authority API

The Slice Authority API provides services to manage slices and their associated permissions. To support its AuthZ policies, a particular SA may choose to manage objects and relationships such as projects and slice/project membership. The SA API is thus divided into a set of services, each of which consists of a set of methods. Of these, only the SLICE service is required, the others are optional. If an SA implements a given service, it should implement the entire service as specified. All available SA service methods are available from the same SA URL. The get_version method should indicate, in the 'SERVICES' tag, which services the given SA supports.

All SA calls are protected; passing and validating a client-side cert is required.

The following is a list of potential SA services.

|| ''' Service ''' || ''' Description ''' || ''' Required ''' || ''' Object ''' || || SLICE || Managing generation, renewal of slice credentials and slice lookup services || YES || SLICE || || SLICE_MEMBER ||Defining and managing roles of members with respect to slices || NO || || || SLIVER_INFO || Providing information about what Aggregates have reported having slivers for a given slice. Non-authoritative/advisory || NO || SLIVER_INFO || || PROJECT || Defining projects (groupings of slices) and project lookup services || NO || PROJECT || || PROJECT_MEMBER || Defining and managing roles of members with respect to projects || NO || ||

Slice Service Methods

The Slice Authority manages the creation of slices, which are containers for allocating resources. It provides credentials (called slice credentials) which aggregates may use to make authorization decisions about allocating resources to a particular user to a particular slice. These slice credentials are one of the fields that may be provided from the create_slice call or requested in the lookup_slices call.

The credentials passed to SA Slice Service methods are SA-specific. But a common case is for a tool to want to pass additional information about a user, obtained from the MA, to the SA to allow the SA to make informed authorization decisions. These credentials may be in the form of an SFA-style User Credential or ABAC credential. Common useful information from the MA to the SA about users would be slice-independent (the SA should know all slice-specific information about users) information about roles and attributes of that user. Two conventional roles are: * PI: The user has a PI lead and is typically considered appropriate for creating projects or slices (if there are no projects) * ADMIN: The user has special 'admin' privileges and can perform operations not otherwise authorized.

Note that renewal of slice expiration is handled in the update_slice call (with "SLICE_EXPIRATION" specified as the options key. The semantics of slice expiration is that slice expiration may only be extended, never reduced. Further restrictions (relative to project expiration or relative to slice creation, e.g.) are SA-specific.

The following table contains required fields for slice objects and whether they are allowed in lookup 'match' criteria, required at creation or allowed at update:
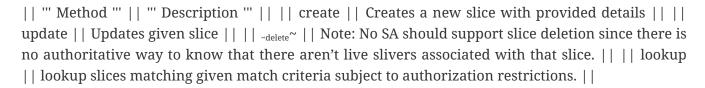
|| ''' Name ''' || ''' Type ''' || ''' Description ''' || ''' Match ''' || ''' Creation ''' || ''' Update ''' || || SLICE_URN || URN || URN of given slice || Yes || No || No || || SLICE_UID ||UID || UID (unique within authority) of slice || Yes || No || No || || SLICE_CREATION || DATETIME || Creation time of slice || No || No || No || || SLICE_EXPIRATION || DATETIME || Expiration time of slice || No || Allowed || Yes || || SLICE_EXPIRED || BOOLEAN || Whether slice has expired || Yes || No || No || || SLICE_NAME || STRING || Short name of Slice|| No || Required || No || || SLICE_DESCRIPTION || STRING || Description of Slice || No || Allowed || Yes || || SLICE_PROJECT_URN || URN || URN of project to which slice is associated (if SA supports project) || Yes || Required (if SA supports project) || No ||

To clarify the semantics of the SLICE_PROJECT_URN field: it is a required field for those SAs that

support the PROJECT service (and in this context may be matched and is required at creation time, but not updatable). In SAs that do not support projects, the field is not meaningful and should not be supported.

NB: SLICE_NAME must adhere to the restrictions for slice names in the Aggregate Manager (AM) API, namely that it must be ⇐ 19 characters, only alphanumeric plus hyphen, no leading hyphen.

The Slice Service supports these standard API methods for type="SLICE":

|| ''' Method ''' || ''' Description ''' || || create || Creates a new slice with provided details || || update || Updates given slice || || ~delete~ || Note: No SA should support slice deletion since there is no authoritative way to know that there aren't live slivers associated with that slice. || || lookup || lookup slices matching given match criteria subject to authorization restrictions. ||

Additionally, the Slice service provides the following methods:

{{{ #!python # Provide list of credentials for the caller relative to the given slice. # If the invocation is in a speaks-for context, the credentials will be for the # 'spoken-for' member, not the invoking tool. # # For example, this call may return a standard SFA Slice Credential and some # ABAC credentials indicating the role of the member with respect to the slice. # # Note: When creating an SFA-style Slice Credential, the following roles # typically allow users to operate at known GENI-compatible # aggregates: "*" (asterisk) or the list of "refresh", "embed", # "bind", "control" "info". # # Arguments: # slice_urn: URN of slice for which to get member's credentials # options: Potentially contains 'speaking_for' key indicating a speaks-for # invocation (with certificate of the accountable member # in the credentials argument) # # Return: # List of credential in "CREDENTIALS" format, i.e. a list of credentials with # type information suitable for passing to aggregates speaking AM API V3. def get_credentials(slice_urn, credentials, options) }}}

Slice Member Service Methods

Slices may have a set of members associated with them in particular roles. Certain SA may have policies that require certain types of membership requirements (exactly one lead, never empty, no more than a certain number of members, etc.). To that end, we provide a single omnibus method for updating slice membership in a single transaction, allowing any authorization or assurance logic to be supported at a single point in SA implementations.

The set of recognized role types (e.g. LEAD, ADMIN, MEMBER, OPERATOR, AUDITOR) are to be listed in the get_version for a given Slice Authority.

The following methods are written generically (with type arguments) to support the Slice Member Service as well as the [#ProjectMemberServiceMethods Project Member Service (below)].

{{{ #!python # Modify object membership, adding, removing and changing roles of members # with respect to given object # # Arguments: # type: type of object for whom to lookup membership (# in the case of Slice Member Service, "SLICE", # in the case of Project Member Service, "PROJECT") # urn: URN of slice/project for which to modify membership # Options: # members_to_add: List of member_urn/role tuples for members to add to # slice/project of form # {'SLICE_MEMBER' : member_urn, 'SLICE_ROLE' : role} # (or 'PROJECT_MEMBER/PROJECT_ROLE # for Project Member Service) # members_to_remove: List of member_urn of members to # remove from slice/project #

members_to_change: List of member_urn/role tuples for # members whose role # should change as specified for given slice/project of form # {'SLICE_MEMBER' : member_urn, 'SLICE_ROLE' : role} # (or 'PROJECT_MEMBER/PROJECT_ROLE for Project Member Service) # # Return: # None def modify_membership(type, urn, credentials, options) }}}

{{{ #!python # Lookup members of given object and their roles within that object # # Arguments: # type: type of object for whom to lookup membership # (in the case of Slice Member Service, "SLICE", # in the case of Project Member Service, "PROJECT") # urn: URN of object for which to provide current members and roles # # Return: # List of dictionaries of member_urn/role pairs # [{'SLICE_MEMBER': member_urn, # 'SLICE_ROLE': role }...] # (or PROJECT_MEMBER/PROJECT_ROLE # for Project Member Service) # where 'role' is a string of the role name. def lookup_members(type, urn, credentials, options) }}}

{{{ #!python # Lookup objects of given type for which the given member belongs # # Arguments: # type: type of object for whom to lookup membership # (in the case of Slice Member Service, "SLICE", # in the case of Project Member Service, "PROJECT") # member_urn: The member for whom to find slices to which it belongs # # Return: # List of dictionary of urn/role pairs # [('SLICE_URN' : slice_urn, 'SLICE_ROLE' : role} ...] # (or PROJECT_MEMBER/PROJECT_ROLE # for Project Member Service) # for each object to which a member belongs, # where role is a string of the role name def lookup_for_member(type, member_urn, credentials, options) }}}

Sliver Info Service Methods

Sliver information is authoritatively held in aggregates: aggregates know which slivers are in which slices at that aggregate. As a convenience to tools, aggregates are encouraged to register with the SA which slices they have information about. In this way, tools can reference only certain aggregates and not all known aggregates to get a useful (if not authoritative) set of sliver details for a slice.

It is expected that the sliver_info create, update and delete calls will be restricted to aggregates (in which case no speaks-for credential is required). That said, SAs may implement authorization policies of their choosing on these calls.

The following table contains the required fields for sliver info objects and whether they are allowed in lookup 'match' criteria, required at creation or allowed at update:

'''Name'''	'''Type'''	'''Description'''	'''Match'''	'''Creation'''	'''Update'''
SLIVER_INFO_SLICE_URN	URN	URN of slice for registered sliver	Yes	Required	No
SLIVER_INFO_URN	URN	URN of registered sliver	Yes	Required	No
SLIVER_INFO_AGGREGATE_URN	URN	URN of aggregate of registered sliver	Yes	Required	No
SLIVER_INFO_CREATOR_URN	URN	URN of member/tool that created the registered sliver	Yes	Required	No
SLIVER_INFO_EXPIRATION	DATETIME	Time of sliver expiration	No	Required	Yes
SLIVER_INFO_CREATION	DATETIME	Time of sliver creation	No	Allowed	No

Note that the SLIVER_INFO_URN is the unique key for this data table (there may be multiple slices per aggregate or multiple aggregates per slice, but the sliver is absolutely unique over all slices and aggregates.

The Sliver Info Service supports these standard API methods for type="SLIVER_INFO":

|| ''' Method ''' || ''' Description ''' || || create || Registers new sliver info with provided details || || update || Updates given sliver info || || delete || Deletes given sliver info || || lookup || lookup sliver info matching given match criteria subject to authorization restrictions. ||

Project Service Methods

Projects are groupings of slices and members for a particular administrative purpose. Some SA's will chose to create and manage projects and apply policies about the invocation of SA methods (e.g. the creation of slice credentials based on roles or memberships in projects). A slice can belong to no more than one project; a project may have many slice members.

The following table contains required fields for project objects and whether they are allowed in lookup 'match' criteria, required at creation or allowed at update:

|| ''' Name ''' || ''' Type ''' || ''' Description ''' || ''' Match ''' || ''' Creation ''' || ''' Update ''' || || PROJECT_URN || URN || URN of given project || Yes || No || No || || PROJECT_UID ||UID || UID (unique within authority) of project || Yes || No || No || || PROJECT_CREATION || DATETIME || Creation time of project || No || No || No || || PROJECT_EXPIRATION || DATETIME || Expiration time of project || No || Required || Yes || || PROJECT_EXPIRED || BOOLEAN || Whether project has expired || Yes || No || No || || PROJECT_NAME || STRING || Short name of Project || Yes || Required || No || || PROJECT_DESCRIPTION || STRING || Description of Project || No || Allowed || Yes ||

The Project Service supports these standard API methods for type="PROJECT":

|| ''' Method ''' || ''' Description ''' || || create || Creates a new project with provided details || || update || Updates given project || || delete || Deletes given project. Note: should fail if there are any active slices associated with project. || || lookup || lookup projects matching given match criteria subject to authorization restrictions. ||

Project Member Service Methods

Projects may have members associated with them in particular roles and thus supports the same methods for member management as described above for the [#SliceMemberServiceMethods Slice Member Service]. The differences are that the type provided is "PROJECT", the urn provided is a project URN and the membership information returned is tagged with "PROJECT_URN" and 'PROJECT_ROLE' as appropriate.

For method signatures, see the listing under the [#SliceMemberServiceMethods Slice Member Service].

|| ''' Method ''' || ''' Description ''' || || modify_membership || Adds/removes/changes roles of members with respect to given project || || lookup_members || Returns list of {PROJECT_MEMBER, PROJECT_ROLE} dictionaries| for members projects matching given criteria || || lookup_for_member || Returns list of {PROJECT_URN, PROJECT_ROLE} dictionaries for projects to which a given member belongs ||

Member Authority API

The Member Authority API provides services to manage information about federation members including public and potentially private or identifying information.

As noted above, this document does not specify required policies for Federations. A given MA is free to implement its own policies. That said, the management of member private information is a subject for particular attention and care.

All MA calls are protected; passing and validating a client-side cert is required.

> While each MA is free to implement its own authorization policy, reasonable security policy should allow calls to succeed only if the following criteria are met:

- The user/tool cert is signed by someone in the Federation's trust chain

- If the cert is held by a tool, then the call must contain a user cert and a 'speaks-for' credential and the tool is trusted by the Federation to perform speaks-for.

- The requestor is asking for their own identifying info or has privileges with respect to the people about whom they are asking for that identifying info.

- Access to private info (SSL or SSH keys) should be restricted only to the user's own keys for ordinary users.

Like the Slice Authority, the Member Authority provides a set of services each consisting of a set of methods. Some services are required for any MA implementation, others are optional, as indicated by this table:

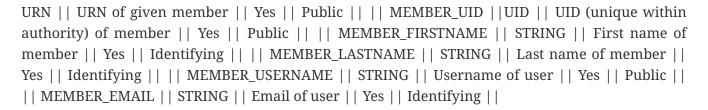|| '''Service''' || '''Description''' || '''Required''' || '''Object''' || MEMBER || Services to lookup and update information about members || YES || MEMBER || || KEY || Services to support storing, deleting and retrieving keys (e.g. SSH) for members || NO || KEY ||

Member Service Methods

The information managed by the MA API is divided into three categories, for purposes of applying different AuthZ policies at these different levels:

- Public: Public information about a member (e.g. public SSH or SSH keys, speaks-for credentials, certificates)

- Private: Private information (e.g. private SSL or SSH keys) that should be given only to the member or a tool speaking for the member with a valid speaks-for credential

- Identifying: Information that could identify the given member (e.g. name, email, affiliation)

The following table contains required fields for member objects and whether they are allowed in lookup 'match' criteria and their protection (public, private, identifying):

|| ''' Name ''' || ''' Type ''' || ''' Description ''' || ''' Match ''' || ''' Protection ''' || || MEMBER_URN ||

URN || URN of given member || Yes || Public || || MEMBER_UID ||UID || UID (unique within authority) of member || Yes || Public || || MEMBER_FIRSTNAME || STRING || First name of member || Yes || Identifying || || MEMBER_LASTNAME || STRING || Last name of member || Yes || Identifying || || MEMBER_USERNAME || STRING || Username of user || Yes || Public || || MEMBER_EMAIL || STRING || Email of user || Yes || Identifying ||

The MEMBER Service supports these standard API methods for type="MEMBER":

|| ''' Method ''' || ''' Description ''' || || update || update info associated with given member by URN || || lookup || lookup info associated with members matching match criteria. ||

Note: the "lookup" call provides public information for all members matching the 'match' criteria. It will also provide identifying (e.g. email or name) or private (e.g. SSL private key) information for members for whom the caller is authorized. When a field requested is unauthorized, the key will not be provide in the returned dictionary for that member. When the field requested has a key but a blank/null value, the access is authorized but the value for that field is, in fact, blank. A blank (null, not empty list) fields option indicates that the caller wants to see all fields to which the caller is authorized. If a list of fields is specified in the fields option, only those authorized fields from among the specified set is provided for each matched member.

The following are additional methods provided by the MEMBER service:

{{{ #!python # Provide list of credentials (signed statements) for given member # This is member-specific information suitable for passing as credentials in # an AM API call for aggregate authorization. # Arguments: # member_urn: URN of member for which to retrieve credentials # options: Potentially contains 'speaking_for' key indicating a speaks-for # invocation (with certificate of the accountable member in the credentials argument) # # Return: # List of credential in "CREDENTIALS" format, i.e. a list of credentials with # type information suitable for passing to aggregates speaking AM API V3. def get_credentials(member_urn, credentials, options) }}}

Key Service Methods

The Key Service provides methods to allow for storing, deleting and retrieving SSH or similar keys for members. It is not intended for retrieving SSL public/private keys or certs.

The following table contains the required fields for key objects and whether they are allowed in lookup 'match' criteria, required at creation or allowed at update:

|| '''Name''' || '''Type''' || '''Description''' || '''Match''' || '''Creation''' || '''Update''' || || KEY_MEMBER || URN || URN of member associated with key pair || Yes || Required || No || || KEY_ID || STRING || Unique identifier for member/key pair: typically a fingerprint or hash of public key joined with member information || Yes || No || No || || KEY_TYPE || STRING || Type of key (e.g. PEM, openssh, rsa-ssh) || Yes || Required || No || || KEY_PUBLIC || KEY || Public key value || Yes || Required || No || || KEY_PRIVATE || KEY || Private key value || Yes || Allowed || No || || KEY_DESCRIPTION || STRING || Human readable description of key pair || Yes || Allowed || Yes ||

The Key Service supports these standard API methods for type="KEY":

|| ''' Method ''' || ''' Description ''' || || create || Creates a new record for a key associated with a member. The 'KEY_ID' returned from this call is the unique identifier for this key for this member and can be used as the 'urn' variable in the other key management API calls below. || || update || urn is the key_id || || delete || urn is the key_id || || lookup || lookup keys matching given match criteria subject to authorization restrictions. ||

Note that access to key information is subject to authorization policy. The public keys are likely to be readily available but access to the private keys will be tightly restricted (often only to the user or authorized proxy). Requests to lookup key information for prohibited filter criteria results in omitting these fields. For example, if one asks for KEY_PUBLIC and KEY_PRIVATE for a list of member_urn's, the result may return both KEY_PUBLIC and KEY_PRIVATE for certain (permitted) users, and only KEY_PUBLIC for other (restricted) users.

Appendix: Federation Object Models

As described, each Federation service method takes a set of options that provide further details on the request. Many of these options reflect the fields of the underlying object models. For example, the Slice Authority manages slice objects and allows for options for querying for and by slice object fields.

Different Federation Authorities will implement different subsets of the possible set of Federation services. Those that do implement a given service should implement the API's described above. The fields of the objects maintained through these API's are flexible: some fields are required but different Authorities may have their own additional data, to be returned by the get_version method.

The following diagram reflects the different objects maintained within the full range of Authority services, their interactions and mandatory fields.

[[Image(FedObjectModel.png, 50%,nolink)]]

Appendix B: API Data Types

The following table describes the data types referenced in the document above, in terms of format and meaning.

|| ''' Type ''' || ''' Description ''' || ''' Format ''' || || URN || Standard GENI identifier, guaranteed to be unique across all GENI services and authorities at a given time, but may be reused by obsolete/expired objects (e.g. slices) || '''Example:''' urn:publicid:IDN+mych+user+abrown ''' Details: ''' urn:publicid:IDN+AUTHORITY+TYPE+NAME where AUTHORITY is the unique fully qualified identifier of the authority creating the URN (e.g. ch.geni.net), TYPE is the type of entity (e.g. slice, user, tool, project) and name is the unique name of the entity (e.g. slice_name, user_name, tool_name, project_name). See http://groups.geni.net/geni/wiki/GeniApiIdentifiers for data type definitions. || || UID || Unique identifier within the scope of a single authority, not guaranteed to be unique across authorities || '''Example:''' 8e405a75-3ff7-4288-bfa5-111552fa53ce '''Details:''' Varies by implementation but the python UUID4 standard is a good example. See RFC4122 standard http://www.ietf.org/rfc/rfc4122.txt || || STRING || Generic UTF-8 string || || || INTEGER || Generic integer argument || || || DATETIME || String representing a date/time in RFC3339 format (http://tools.ietf.org/html/rfc3339). || '''Examples:''' 2013-06-15T02:39:08+03:00, 2013-06-15T02:39:08-

05:00, 2014-02-23T11:00:05Z '''Details''': DATETIME values in the Federation API will be strings in RFC3339-compliant format. We "recommend" that implementers use parsers that fully comply with this standard. However, due to the flexibility in the spec and different interpretations chosen by different common parsers, we "require" that such DATETIME values: 1) contain an uppercase T between the time and date portions, 2) contain a timezone suffix, either an uppercase Z (for UTC) or +/-HH:MM, and 3) do not contain fractional seconds.|| || EMAIL || Well-formed email address compliant with RFC2822 http://tools.ietf.org/html/rfc2822#section-3.4.1 || '''Example:''' jbrown@geni.net || || KEY || SSH or SSL public or private key (contents, not filename) || Key-specific format || || BOOLEAN || XMLRPC encoded boolean || '''Example:''' True || || CREDENTIALS || List of dictionaries, one per credential, tagged with credential type and version (as indicated in the GENI AM API specification) || '''Details:''' Credentials = [{ geni_type: <string, case insensitive>, geni_version: <string containing an integer>, geni_value : <credential as string>, <others> }]. See http://groups.geni.net/geni/wiki/GAPI_AM_API_V3/CommonConcepts#credentials or http://groups.geni.net/geni/wiki/GeniApiCertificates for credential format and semantic specification. || || CERTIFICATE || X509 v3 certificate (contents, not filename) || Standard X509 v3 PEM certificate format. A chain of such certificates may be concatenated. See http://en.wikipedia.org/wiki/X.509 and http://groups.geni.net/geni/wiki/GeniApiCertificates for more details ||

As noted above, this list is subject to change as the API develops over time.

Appendix C: API V1 and V2 Mappings

Federation API V2 makes significant changes to the previous (V1) Federation API. Specifically, it generalizes many of the API calls thy introducing a 'type' argument. This table summarizes the changes to V1 calls and their equivalent in V2.

|| ''' Authority ''' || ''' V1 method ''' || ''' V2 alternative '' || || Federation Registry || || || || || lookup_aggregates || lookup(type="SERVICE") match: {"SERVICE_TYPE":"SLICE_AUTHORITY"}|| || || lookup_slice_authorities || lookup(type="SERVCE") match:{"SERVICE_TYPE":"SLICE_AUTHORITY"} || || || lookup_member_authorities || lookup(type="SERVICE") match:"{"SERVICE_TYPE":"MEMBER_AUTHORITY"} || || Slice Authority || || || || || create_slice || create(type="SLICE") || || || lookup_slices || lookup(type="SLICE") || || || update_slice || update(type="SLICE") || || || modify_slice_membership || modify_membership(type="SLICE") || || || lookup_slice_members || lookup_members(type="SLICE") || || || lookup_slices_for_member || lookup_for_member(type="SLICE") || || || create_sliver_info || create(type="SLIVER_INFO") || || || delete_sliver_info || delete(type="SLIVER_INFO") || || || update_sliver_info || update(type="SLIVER_INFO") || || || lookup_sliver_info || lookup(type="SLIVER_INFO") || || || create_project || create(type="PROJECT") || || || lookup_projects || lookup(type="PROJECT") || || || update_project || update(type="PROJECT") || || || modify_project_membership || modify_membership(type="PROJECT") || || || lookup_project_members || lookup_members(type="PROJECT") || || || lookup_projects_for_member || lookup_for_member(type="PROJECT") || || Member Authority || || || || || lookup_public_member_info || lookup(type="MEMBER") with fields option containing list of public fields only || || || lookup_identifying_member_info || lookup(type="MEMBER") with fields option containing list of identifying fields only || || || lookup_private_member_info ||

lookup(type="MEMBER") with fields option containing list of private fields only || || || update_member_info || update(type="MEMBER") || || || create_key || create(type="KEY") || || || delete_key || delete(type="KEY") || || || update_key || update(type="KEY") || || || lookup_keys || lookup(type="KEY") ||