**UGE Python Configuration Library: High-Level Design**

**August 31, 2016**

**Prepared by:**

**Siniša Veseli**

**Consultant**

**Skaisoft Corp.**

**[sinisa.veseli@skaisoft.com](mailto:sinisa.veseli@skaisoft.com)**

**Change History**

|  |  |  |
| --- | --- | --- |
| **Date** | **Description** | **Author** |
| 4/18/16 | Initial version. | S. Veseli |
| 8/31/16 | Revised document to reflect changes after implementation. | S. Veseli |

Table of Contents

[1 Introduction 5](#_Toc448532445)

[2 Assumptions and Remarks 5](#_Toc448532446)

[3 High-Level Design 5](#_Toc448532447)

[3.1 Communication with Qmaster 5](#_Toc448532448)

[3.2 UGE Objects and their Python Representation 6](#_Toc448532449)

[3.2.1 Collections of Key/Value Pairs 6](#_Toc448532450)

[3.2.2 List of Strings 7](#_Toc448532451)

[3.2.3 List of Key/Value Collections 7](#_Toc448532452)

[3.3 Metadata versus UGE Data 8](#_Toc448532453)

[3.4 PyCL Object Classes 9](#_Toc448532454)

[3.4.1 QconfObject Class 9](#_Toc448532455)

[3.4.2 Initializing Objects 10](#_Toc448532456)

[3.4.3 Support for API CRUD Operations 11](#_Toc448532457)

[3.4.4 Object Versioning 12](#_Toc448532458)

[3.5 API Functionality 13](#_Toc448532459)

[3.5.1 Initialization 13](#_Toc448532460)

[3.5.2 Object Factory Methods 13](#_Toc448532461)

[3.5.3 CRUD and List Methods 14](#_Toc448532462)

[3.5.4 Support for UGE Upgrades 16](#_Toc448532463)

[3.5.5 Exceptions 17](#_Toc448532464)

[3.5.6 Top-Level Module 17](#_Toc448532465)

[3.5.7 Logging 17](#_Toc448532466)

[3.5.8 Testing 18](#_Toc448532467)

[3.5.9 Documentation 18](#_Toc448532468)

[3.6 Command Line Interfaces 18](#_Toc448532469)

[4 Conclusions 19](#_Toc448532470)

[5 References 19](#_Toc448532471)

[6 Appendix: PyCL Object Details and API Specification 20](#_Toc448532472)

[6.1 ClusterQueue 21](#_Toc448532473)

[6.2 Cluster Configuration 22](#_Toc448532474)

[6.3 Scheduler Configuration 24](#_Toc448532475)

[6.4 Execution Host 26](#_Toc448532476)

[6.5 Host Group 27](#_Toc448532477)

[6.6 Complex Configuration 28](#_Toc448532478)

[6.7 Project 30](#_Toc448532479)

[6.8 User 31](#_Toc448532480)

[6.9 Access List 32](#_Toc448532481)

[6.10 Job Class 33](#_Toc448532482)

[6.11 Resource Quota Set 34](#_Toc448532483)

[6.12 Parallel Environment 35](#_Toc448532484)

[6.13 Share Tree 36](#_Toc448532485)

[6.14 Calendar 37](#_Toc448532486)

[6.15 Checkpointing Environment 38](#_Toc448532487)

[6.16 Managers 39](#_Toc448532488)

[6.17 Operators 40](#_Toc448532489)

[6.18 Submit Hosts 40](#_Toc448532490)

[6.19 Admin Hosts 41](#_Toc448532491)

# Introduction

The purpose of this document is to describe high-level design for the Univa Grid Engine Python Configuration Library (UGE PyCL) that can be used by other applications and tools to configure and manage one or more UGE clusters.

# Assumptions and Remarks

* UGE PyCL will require access to UGE qmaster via the “qconf” command.
* There will be no user authentication/authorization performed by the library. All library calls utilize the same user account under which they are invoked. Hence, any library calls that modify UGE configuration will have to be invoked from an account that has appropriate UGE administrative privileges.
* Ideas and design concepts discussed in this document do not rely on a specific Python version. However, some examples and code snippets may use modules or syntax specific to Python 2.
* UGE software version used for developing this design was 8.1.3p7 [1].
* This document attempts to be as detailed as possible in terms of specifying UGE PyCL high-level design and functionality. Nevertheless, it is expected that the library implementation phase will likely result in additions or slight modifications of some features.

# High-Level Design

In this section we describe considerations driving UGE PyCL design.

## Communication with Qmaster

UGE PyCL will communicate with qmaster via the “qconf” command: retrieving UGE objects will involve parsing output of qconf “display” commands, while adding or modifying objects will require constructing appropriate object definition files and passing those to qconf add or modify commands. Hence, the high-level library API class will need the same set of variables as the qconf command itself:

* SGE\_ROOT
* SGE\_CELL
* SGE\_QMASTER\_PORT
* SGE\_EXECD\_PORT

The above variables can be either passed to the API class constructor, or they can be inherited from the user’s environment. Note that configuring API objects programmatically should allow different API class instances to manage different UGE clusters at the same time. For example, one should be able to copy queue configuration from one cluster to another using the following Python script:

qconf1 = QconfApi(sge\_root=’/opt/uge’, sge\_cell=’default’, sge\_qmaster\_port=11111, sge\_execd\_port=11112)

qconf2 = QconfApi(sge\_root=’/opt/uge2’, sge\_cell=’default’, sge\_qmaster\_port=21111, sge\_execd\_port=21112)

all\_q = qconf1.get\_queue(’all.q’)

qconf2.modify\_queue(all\_q)

Figure 1: Python code snippet that illustrates using multiple API class instances to manage different clusters.

Note that various PyCL objects and API methods will be described in more detail later in this document.

## UGE Objects and their Python Representation

Although their formatting and display are not consistent across different qconf commands, all UGE objects fit into one of the following three categories:

* Collection of key/value pairs (queues, host groups, projects, job classes, etc.)
* List of strings (lists of manager names, queue names, execution host names, etc.)
* List of key/value collections (share tree, list of resource quota sets)

### Collections of Key/Value Pairs

All individual UGE objects with more than one attribute can be described as collections of key/value pairs. A sample definition of project “P1” illustrates this in Figure 2:

$ qconf -sprj P1

name P1

oticket 100

fshare 100

acl ACL1

xacl NONE

Figure 2: Sample UGE project “P1” object, shown using output of the “qconf -sprj P1” command.

In Python, the natural representation for all such objects is dictionary, which is easily converted into JSON, as illustrated in Figure 3.

>>> import json

>>> p1\_data = {'oticket': 100, 'xacl': 'NONE', 'fshare': 100, 'name': 'P1', 'acl': 'ACL1'}

>>> json.dumps(p1\_data)

'{"oticket": 100, "xacl": "NONE", "acl": "ACL1", "fshare": 100, "name": "P1"}'

Figure 3: Python code snippet showing dictionary that represents sample UGE project “P1”, as well as its conversion to the JSON string.

In most cases we can simply use native python types or strings as containers for values assigned to UGE object keys (UGE keywords like ‘NONE’ will be discussed later). There are, however, some cases where using a list of strings or a dictionary works slightly better in terms of parsing or modifying key values. For example, values for several “queue” object keys like “hostlist” and “slots” can be represented using lists of strings, while values for all keys of the “complex” object can be represented using dictionaries (see Figure 4).

>>> complex\_data = {'arch' : {'shortcut' : 'a', 'type' : 'RESTRING', 'relop' : '==', 'requestable' : 'YES', 'consumable' : 'NO', 'default' : 'NONE', 'urgency' : 10, 'aapre' : 'NO'}}

Figure 4: Python code snippet showing a dictionary representation of the “arch” key value in the UGE “complex” configuration.

### List of Strings

Python list of strings is a natural representation qconf commands that return list of names. An example is a list of execution hosts (see Figures 5 and 6).

$ qconf -sel

uge-exec-001

uge-exec-002

uge-exec-003

Figure 5: Sample output of the “qconf -sel” command.

>>> import json

>>> host\_list = ['uge-exec-001', 'uge-exec-002', 'uge-exec-003']

>>> json.dumps(host\_list)

'["uge-exec-001", "uge-exec-002", "uge-exec-003"]'

Figure 6: Python code snippet showing list of execution hosts, as well as its conversion to the JSON string.

### List of Key/Value Collections

For those qconf commands that return list of key/value collections we can use Python list of dictionaries (see Figures 7 and 8 showing a sample share tree and its Python representation).

id=0

name=Root

type=0

shares=1

childnodes=1,2

id=1

name=U1

type=0

shares=10

childnodes=NONE

id=2

name=U2

type=0

shares=25

childnodes=NONE

Figure 7: Sample output of the “qconf -sstree” command.

>>> import json

>>> stree\_data = [{'id' : 0, 'name' : 'Root', 'type' : 0, 'shares' : 1, 'childnodes' : [1,2]}, {'id' : 1, 'name' : 'U1', 'type' : 0, 'shares' : 10, 'childnodes' : 'NONE'}, {'id' : 2, 'name' : 'U2', 'type' : 0, 'shares' : 25, 'childnodes' : 'NONE'}]

>>> json.dumps(stree\_data)

'[{"childnodes": [1, 2], "type": 0, "id": 0, "shares": 1, "name": "Root"}, {"childnodes": "NONE", "type": 0, "id": 1, "shares": 10, "name": "U1"}, {"childnodes": "NONE", "type": 0, "id": 2, "shares": 25, "name": "U2"}]'

Figure 8: Python code snippet showing sample share tree representation, as well as its conversion to the JSON string.

## Metadata versus UGE Data

Effective configuration management of multiple UGE clusters requires information about various UGE objects, in addition to actual object configuration data that was discussed in Section 3.2. For example, UGE objects may evolve between different software versions, and hence it is imperative to know which object version was actually stored in the configuration database. Another example is tracking configuration changes: cluster administrators might like to know the time/date of the most recent configuration change, which administrator made the change, and why.

For this reason, PyCL objects will contain both UGE configuration data and object metadata. Different configuration tools might rely on different sets of metadata keys, and the list of metadata keys in usage will likely grow with time. At a minimum, however, metadata will have to contain set of static keys that are needed by the PyCL library itself (e.g., for recreating configuration objects from JSON strings):

* object\_class
* object\_version

In addition to those, some of the optional metadata keys that may be added to PyCL objects are listed below:

* object\_name (may be needed for identifying some objects)
* created\_by
* created\_on
* modified\_by
* modified\_on
* uge\_cluster
* uge\_version
* description
* …

Note that some of the above metadata keys may be added automatically by the PyCL library, while others may be added by configuration tools using PyCL object interfaces.

Although PyCL objects may have other data members, their JSON representations will be based on dictionary formed only from metadata dictionary key/value pairs, as well as from UGE object assigned to the reserved key “data”. For example, JSON string representation for a PyCL object that “wraps” sample UGE project “P1” from Figure 2 is shown below:

>>> import json

>>> p1\_data = {'oticket': 100, 'xacl': 'NONE', 'fshare': 100, 'name': 'P1', 'acl': 'ACL1'}

>>> p1\_json\_data = {'object\_class' : 'Project', 'object\_version' : '1.0', 'modified\_by' : 'sveseli', 'modified\_on' : '2016-04-15T19:30:07.969048', 'uge\_cluster' : 'C1', 'data' : p1\_data}

>>> json.dumps(p1\_json\_data)

'{"object\_version": "1.0", "modified\_by": "sveseli", "uge\_cluster": "C1", "data": {"oticket": 100, "xacl": "NONE", "acl": "ACL1", "fshare": 100, "name": "P1"}, "modified\_on": "2016-04-15T19:30:07.969048", "object\_class": "Project"}'

Figure 9: Python code snippet showing full JSON string representation (data and metadata) for a sample UGE “P1” project.

Note that timestamps in metadata will be strings using ISO 8601 format.

## PyCL Object Classes

All qconf definitions and logic related to a given UGE object will be encapsulated within its corresponding PyCL wrapper class. For example, the PyCL “Queue” class will know that retrieving UGE queue objects requires a “qconf -sq <queue\_name>” command, it will have the list of keys needed to create a new queue object, it will provide interfaces needed to modify queue configuration, it will know how to parse and unpack JSON string, and also how to prepare files suitable for a “qconf –Aq <queue\_definition\_file>” or for a “qconf –Mq <queue\_definition\_file>” command.

### QconfObject Class

PyCL object classes will derive from a common “QconfObject” base class (see Figure 10). This class will contain functionality and interfaces common to all PyCL objects (e.g., conversion to JSON strings, logic for parsing output of qconf commands that display UGE objects, etc.).

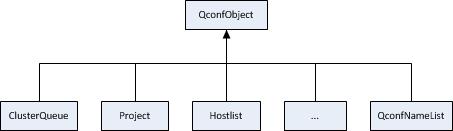


Figure 10: All PyCL object classes will use QconfObject as a base class.

In addition to classes wrapping UGE objects, the above diagram also contains special “QconfNameList” class, which will support all regular Python list features, and will be used as a container for various lists of object names, such as cluster queues, managers, execution hosts, etc.

### Initializing Objects

Constructors for QconfObject-based classes will take several named (and all optional) arguments: “data”, “metadata”, and “json” (see Figure 11).

class QconfObject:

def \_\_init\_\_(self, name=None, data=None, metadata=None, json=None):

...

Figure 11: QconfObject class constructor signature.

The “data” argument should contain UGE object configuration data (e.g., dictionary in case of objects like queues or projects, list of dictionaries for share trees). The “metadata” argument should be a dictionary containing any user-provided key/value pairs, which will be merged with static class metadata. The “json” argument should contain object’s JSON string representation discussed earlier. If specified, JSON string will be unpacked, and resulting key/value pairs will be merged with the provided metadata and/or data. In case identical keys are specified in multiple places (e.g., in both metadata dictionary and as part of the JSON string), values coming from the JSON string will take precedence.

The scheme described above allows for creating empty, partially or fully configured objects, and also objects that can be easily updated and manipulated, as shown in Figure 12.

>>> all\_q = ClusterQueue()

>>> all\_q.data['qname'] = 'all.q'

Figure 12: Python code snippet that shows how an empty cluster queue object can be created and manipulated.

Keep in mind, however, that requirement for supporting multiple UGE releases complicates things in terms of invoking object constructors using classes that are applicable for a given UGE release. For this reason, the PyCL library will provide object factory methods that should be able to instantiate appropriate versions of object wrapper classes, using either provided JSON string representation, or UGE object data (see Figure 13).

qconf = QconfApi(sge\_root='/opt/uge', sge\_cell='default')

all\_q = qconf.generate\_queue(data={'qname' : 'all.q'})

Figure 13: Python code snippet showing sample usage of factory methods for creating PyCL objects.

### Support for API CRUD Operations

In order to support API Create, Read, Update and Delete (CRUD) operations PyCL UGE object wrapper classes will have to know appropriate set of qconf command switches. Read operations will also require ability to parse qconf “display” command output, and to generate object’s JSON representation. On the other hand, create and update (or modify) operations will need functionality for parsing JSON strings and creating UGE object configuration files.

Most UGE objects have a set of keys that must be present in their configuration files in order for those files to be accepted by the qmaster. For this reason, every PyCL wrapper class will have a dictionary of required keys and their respective default values (see example below for UGE project objects).

REQUIRED\_DATA\_DEFAULTS = {

'name' : None,

'oticket' : 0,

'fshare' : 0,

'acl' : None,

'xacl' : None

}

Figure 14: Python code snippet showing default values for required data keys for UGE projects.

Most keys that have default values will not have to be provided by the library user. For certain keys values will have to be set by the library user before UGE object is added or modified via the relevant PyCL API calls (e.g., “qname” attribute of UGE cluster queue). Note that default values for certain keys may be installation specific, and will have to be generated at runtime (for example, those keys that rely on $SGE\_ROOT)

Some objects (e.g., cluster configuration) have a set of keys that are optional (i.e., they can be removed deleted from object’s configuration). In order to accommodate such cases, PyCL wrapper class may utilize dictionaries of optional data keys and their respective default values. For example:

OPTIONAL\_HOST\_DATA\_DEFAULTS = {

'mailer' : '/bin/mail',

'xterm' : '/usr/bin/xterm'

}

OPTIONAL\_GLOBAL\_DATA\_DEFAULTS = {

'execd\_spool\_dir' : 'SGE\_ROOT/SGE\_CELL/spool',

'mailer' : '/bin/mail',

'xterm' : '/usr/bin/xterm',

…

}

Figure 15: Python code snippet showing default values for optional data keys for UGE cluster (global and host-specific) configuration. Keys that rely on variables like SGE\_ROOT and SGE\_CELL will have to be evaluated at runtime.

### Object Versioning

Over time, UGE objects will evolve with new UGE releases, and hence their corresponding PyCL wrapper classes will have to change as well. Any time PyCL object wrapper class gets modified in a non-trivial manner it will get assigned a new version string. For example, this would happen if a new key gets added to the set of required object keys, or if a default key value changes. The library must keep track of old PyCL object versions as long as the corresponding UGE product release is supported. This means two things:

1. For each UGE object there may more than one version of PyCL wrapper class that must be supported. For example, there may be three supported versions of the Queue class, but only one supported version of the Project class.
2. Each supported UGE release may correspond to a distinct set of PyCL wrapper classes. For example, future UGE versions 9.0 and 10.0 might use Queue class versions 1.0 and 2.0, respectively, and use the same 1.0 version of the Project class.

The first statement above suggests that PyCL could adopt Java-like “class per file” convention, meaning that the definition of each wrapper class will be specified in its own file, with a version in its name. The exact wrapper class file naming convention is not that important, as long as it is applied consistently across all objects. For example, ClusterQueue class versions 1.0 and 2.0 could be specified in files “cluster\_queue\_v1\_0.py” and “cluster\_queue\_v2\_0.py”, allowing their usage as in the example below:

>>> from cluster\_queue\_v1\_0 import ClusterQueue as ClusterQueueV1

>>> from cluster\_queue\_v2\_0 import ClusterQueue as ClusterQueueV2

>>> q1 = ClusterQueueV1()

>>> q2 = ClusterQueueV2()

>>> print q1.\_\_class\_\_.\_\_name\_\_,q1.VERSION

ClusterQueue 1.0

>>> print q2.\_\_class\_\_.\_\_name\_\_,q2.VERSION

ClusterQueue 2.0

Figure 16: Python code snippet showing possible solution for a problem of supporting multiple versions for a given object. In the above example, both imported python modules contain definitions for the class “ClusterQueue” with the same constructor signature, but with a different value specified for the “VERSION” constant.

Support for distinct sets of objects for different UGE releases can be implemented using release/object set map, where release versions serve as keys and dictionary of object names/versions as values (see Figure 17). Such scheme, combined with the above solution for supporting different object versions, would require minimal amount of maintenance and development effort, and would provide adequate support for supporting multiple UGE releases, as well as support for product upgrades.

>>> UGE\_RELEASE\_OBJECT\_MAP = {}

>>> UGE\_RELEASE\_OBJECT\_MAP['8.5.0p1'] = { 'ClusterQueue' : '1.0', 'Project' : '1.0', 'ShareTree' : '1.0' }

>>> UGE\_RELEASE\_OBJECT\_MAP['8.5.0p2'] = { 'ClusterQueue' : '2.0', 'Project' : '1.0', 'ShareTree' : '1.0' }

>>> UGE\_RELEASE\_OBJECT\_MAP['8.6.0'] = UGE\_RELEASE\_OBJECT\_MAP['8.5.0p2']

>>> UGE\_RELEASE\_OBJECT\_MAP

{'8.6.0': {'ClusterQueue': '2.0', 'Project': '1.0', 'ShareTree': '1.0'}, '8.5.0p2': {'ClusterQueue': '2.0', 'Project': '1.0', 'ShareTree': '1.0'}, '8.5.0p1': {'ClusterQueue': '1.0', 'Project': '1.0', 'ShareTree': '1.0'}}

Figure 17: Python code snippet showing construction of UGE\_RELEASE\_OBJECT\_MAP, which can be used for tracking versions of PyCL object classes that are applicable for a given UGE release.

In order to simplify object management, PyCL will use QconfObjectFactory class, which will be able to generate PyCL objects appropriate for any version of UGE software.

## API Functionality

### Initialization

As mentioned earlier, QconfApi class will require several variables for communication with qmaster: UGE root directory, cell name, and qmaster and execd ports:

qconf = QconfApi(sge\_root=’/opt/uge’, sge\_cell=’default’, sge\_qmaster\_port=11111, sge\_execd\_port=11112)

Figure 18: Example of QconfApi class initialization.

During initialization QconfApi instance will determine administrator’s username, hostname, as well as UGE software version, which will facilitate using PyCL objects appropriate for the given qmaster version (see discussion in Section 3.4.4).

### Object Factory Methods

QconfApi class will provide factory methods for generating PyCL objects, either from UGE data and metadata, or from JSON string (see Figure 19). The factory methods will take UGE version as argument. If this argument is not provided, API object will use qmaster’s version as default. The factory methods may be invoked directly by library users, or by other API methods.

class QconfApi:

…

def generate\_queue(self, name=None, data=None, metadata=None, json=None,

uge\_version=None, add\_required\_data=True):

…

Figure 19: Signature of a factory method for generating PyCL Queue object.

Note that factory methods will be able to generate objects with added default values for required and/or optional keys that are missing from input data.

### CRUD and List Methods

**Get Methods**

API “get” (read) methods will typically take a name as argument and return the corresponding PyCL object. Steps involved in this process will be the following:

1. Take object’s name as argument for the “get” method.
2. Create new instance of the (empty) PyCL wrapper class (with version appropriate for the given qmaster).
3. Invoke appropriate qconf “show object” command. Raise an “ObjectNotFound” exception in case object is unknown to qmaster.
4. Parse output of the qconf command and form Python structure that represents UGE object’s configuration data.
5. Set metadata and data in the PyCL wrapper object and return it to user.

all\_q = qconf.get\_queue('all.q')

print 'Queue JSON Representation: ', all\_q.to\_json()

Figure 20: Example of using API “get” methods, which always return PyCL objects.

**Add Methods**

The “add” (create) methods will perform appropriate qconf “add” (or “add from file”) command. They will always mimic the original qconf behavior, and will be able to take as input arguments either PyCL objects, or Python structures representing UGE data and metadata, or simply JSON strings. For the dictionary-based UGE objects the “add” process will involve these steps:

1. If needed, create new instance of the PyCL wrapper class from UGE data/metadata, and/or from the given JSON string. PyCL object version will be appropriate for the given qmaster.
2. Verify that all keys are present that must be set by the library user. Raise an “InvalidArgument” PyCL exception if that is not the case.
3. Verify that there are no additional (unknown) keys if optional keys are not allowed. Raise an “InvalidArgument” PyCL exception if that is not the case.
4. Verify that UGE object that is being added does not exist, by invoking qconf “show object” command. Raise an “ObjectAlreadyExists” PyCL exception if that is not the case.
5. Add to the new PyCL object all required keys that are missing, with their default values.
6. Generate UGE object configuration file.
7. Invoke appropriate qconf “add from file” command.
8. Return the new PyCL object to the user.

Note that the above process will allow users to take advantage of UGE default values for most object keys. For example, just like with standard qconf command, one should be able to create new queue by specifying only its name, and taking default values for all other properties.

new\_q = qconf.add\_queue(data={'qname' : 'new.q'})

Figure 21: Example of using API “add” method with a dictionary-based UGE object.

In addition to using UGE data and metadata, one should also be able to create new queue by passing its JSON representation, e.g.:

new\_q = qconf.add\_queue(json='{"object\_version": "1.0", "object\_class": "ClusterQueue", "data": {"qname": "new.q", "slots": ["2", "[uge=exec-001=1]"],…}}')

Figure 22: Example of using API “add” method with JSON string.

For adding names to UGE configuration (e.g., managers or operators), process will be much simpler, as input arguments are simply Python strings, or list of strings to be added. Such methods will have no return values. However, if any of the input names have already been configured, an “ObjectAlreadyExists” exception will be raised, consistent with other API “add” methods.

**Modify Methods**

The “modify” (update) methods will perform appropriate qconf “modify from file” command, and will be able to take as input arguments either PyCL objects, or Python structures representing UGE data and metadata, or simply JSON strings. For the dictionary-based UGE objects the “modify” process will involve these steps:

1. If needed, create new instance of the PyCL wrapper class from UGE data/metadata, and/or from the given JSON string. PyCL object version will be appropriate for the given qmaster.
2. Verify that all keys are present that must be set by the library user. Raise an “InvalidArgument” PyCL exception if that is not the case.
3. Verify that there are no additional (unknown) keys if optional keys are not allowed. Raise an “InvalidArgument” PyCL exception if that is not the case.
4. Verify that UGE object that is being updated actually exists, by invoking qconf “show object” command. Raise an “ObjectNotFound” PyCL exception if that is not the case.
5. Add to the new PyCL object all required keys that are missing, with values taken from the old object (i.e., merge new object into the old one).
6. Generate UGE object configuration file.
7. Invoke appropriate qconf “modify from file” command.
8. Return the new (modified) PyCL object to the user.

The above process will allow users to perform updates with partially configured objects. In other words, one should be able to modify objects using only a subset of keys:

all\_q = qconf.modify\_queue(data={'qname' : 'all.q', 'slots' : ['2', '[uge=exec-001=1]']})

Figure 23: Example of using API “modify” method with a dictionary-based UGE object.

Like with the “add” methods, “modify” methods will also take JSON strings, e.g.:

all\_q = qconf.modify\_queue(json='{"object\_version": "1.0", "object\_class": "ClusterQueue", "data": {"qname": "all.q", "slots": ["2", "[uge=exec-001=1]"],…}}')

Figure 24: Example of using API “modify” method with JSON string.

**Delete Methods**

API “delete” methods will take object’s name as argument and invoke qconf “delete” command. Those methods will not have a return value.

qconf.delete\_queue('new.q')

Figure 25: Example of using API “delete” methods.

**List Methods**

API “list” methods will not take arguments; they will invoke qconf “show list” command appropriate for a given object, and will return a “QconfNameList” object:

queue\_name\_list = qconf.list\_queues()

print 'Queue Name List JSON Representation: ', queue\_name\_list.to\_json()

Figure 26: Example of using API “list” methods, which always return QconfNameList object.

### Support for UGE Upgrades

PyCL API itself will not provide explicit methods for upgrading objects from one version to another, but it will provide support for UGE software upgrades. In other words, library users will be able to write scripts that will perform object and configuration upgrades. In most cases existing objects’ keys will not be removed or renamed, and the following process would work:

1. Take object’s old version JSON string representation, and generate corresponding PyCL object (old version).
2. Generate new (upgraded) PyCL object using the old object’s UGE data and specifying “add\_required\_data=True” for the corresponding factory method. This will add keys required for the new PyCL class using with their default values.

Upgrade scripts for more complex cases will have to incorporate appropriate logic that handles those.

### Exceptions

In case of any errors PyCL API methods will raise exceptions. All library exceptions will derive from the base QconfException class, which in turn will extend standard Python Exception.

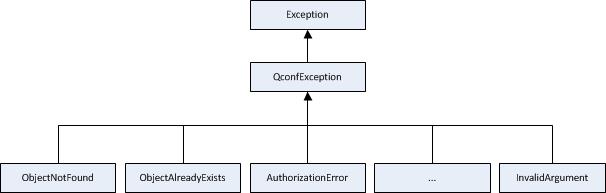


Figure 27: PyCL exceptions will extend QconfException (and standard Python Exception) class.

In those cases where PyCL can determine specific error condition, either by parsing qconf error message or by catching a specific Python error, the library will raise corresponding PyCL exception. For all other cases (either unhandled qconf error, or unexpected Python error) PyCL will raise generic QconfException.

All library exception classes will be initialized with an error message coming either from qconf command error, or from Python exception message. They will also contain unique non-zero error code, which will be used for command line error handling.

### Top-Level Module

PyCL API classes, exceptions and object modules will be accessible under the top-level “uge” module. Hence, the library users will be able to use Python import statements such as “from uge import QconfApi”.

### Logging

PyCL logging framework will be based on the standard Python logging modules and will provide the following functionality:

* The library will be able to log messages into log files, as well as send them to the console.
* There will be several different log levels available (e.g., INFO, WARN, DEBUG, ERROR, CRITICAL).
* Logging configuration (not the code implementation) will determine what messages will be logged and where. In other words, variables like logging level will not be hardcoded.
* Logging configuration will be taken from the following sources: configuration file, environment variables (e.g., PYCL\_LOG\_LEVEL), and command line arguments (for interactive commands).
* Log message format will be configurable, and the system must have ability to prepend timestamp to all messages. For example, log messages might be formatted as follows:

<TIMESTAMP> [<LOG LEVEL>] <USER>@<HOST> <LOGGER NAME> (<PID>): <LOG MESSAGE>

12/15/17 11:35:13 [INFO] root@head01 QconfApi (1234): Adding queue new.q to cluster C1

### Testing

PyCL testing framework will be based on the Python “nose” package [2] and will provide the following functionality:

* Testing will be configurable as far as UGE installation is concerned.
* Testing will be self-reliant (no external setup will be needed other than a valid UGE installation, and running UGE services).
* Testing will be adding, modifying and removing various objects, but will not leave any of its objects in UGE. In other words, the PyCL test suite will restore UGE state after all tests complete.
* Unit tests will include all PyCL objects and API methods.
* Basic successful outcomes for all methods will be tested.
* Testing for various failure scenarios and error handling will be added as problems are found and fixed.

### Documentation

All PyCL API and object classes (i.e., those classes intended for users directly) will be documented using Sphinx Python Documentation generator [3]. API documentation will include usage examples for all API methods.

## Command Line Interfaces

Although primary usage of the PyCL library will be through its API, we anticipate that there will also be a need for command line interfaces (CLIs). Those may be used for scripting of various administrative tasks that might not be suitable for using UGE qconf command directly. For example, if managing production clusters requires logging of all configuration changes, PyCL logging capabilities might give an advantage over using the qconf command.

In order to ensure consistency and ease of use, all of the PyCL command line interfaces should have the following functionality:

* All command line tools will be scriptable.
* All CLI classes will be built on top of the PyCL API.
* Commands will return exit status of 0 in those cases where everything worked as expected. Non-zero exit status will be returned in case of any errors or incorrect usage, and it will correspond to error code of the underlying PyCL API exception.
* Clear error message pointing to the cause of problem will be displayed in case of errors. In most cases, this message will be taken directly from the underlying PyCL exception.
* All tools will print help message and exit (exit status of 0) for the following options:

-h|--help|-?

* All tools will print PyCL version information and exit (exit status of 0) for the following options:

-v|--version

* Logging on the screen will be controlled either using environment variable PYCL\_LOG\_LEVEL, or by providing the following option:

--log=<log level>

* All command parameters may provide “single-dash” flags/options, but they must always offer the “double-dash” syntax (--option=<option value>).

All CLI classes should derive from the base QconfCli class, which will provide set of features and command line options common to all command line tools. Those features include parsing of the command line arguments, error handling, etc.

# Conclusions

In this document we outlined high-level design for the UGE Python Configuration Library. The design aims to be as detailed as possible in terms of specifying the library functionality and structure of its objects. Nevertheless, we expect that the library implementation phase will likely result in additions or slight modifications of some features.

# References

[1] For investigating UGE qconf command functionality and features we relied on official UGE Administrators’ guide, as well as on qconf man pages.

[2] Nose package home page: https://pypi.python.org/pypi/nose

[3] Sphinx project home page: http://www.sphinx-doc.org

# Appendix: PyCL Object Details and API Specification

In this appendix we show PyCL object details: required keys, default UGE values, and corresponding Python types. We also show method signatures for the relevant API methods. Note the following:

* PyCL object keys will be identical to UGE (qconf) keys.
* PyCL objects will use native Python types whenever possible, and will also attempt to convert UGE object values to appropriate types. In those cases where conversion to native types is not possible UGE object values will be interpreted either as strings, or as string lists (UGE object list entries are separated by commas). Standard UGE string keyword “NONE” (or “none”) will be converted to Python None. Similarly, UGE keywords “YES” and “NO” will be converted to Python Boolean values. The library will handle reverse conversions (None to “NONE”, etc.) when creating or modifying UGE objects.
* Some object keys will have to be set by the library user. All other missing keys will be replaced by their default values when creating or modifying UGE objects.
* Default values for certain keys that rely on variables like SGE\_ROOT and SGE\_CELL may be generated at runtime.
* PyCL will accept any input that results in a valid UGE object configuration.
* When manipulating objects, Python string list is equivalent to a single string where list items are joined by commas. In other words, key value specifications like the ones shown below are equivalent as far as PyCL API is concerned:

>>> all\_q.data['slots'] = ['1','[uge-exec-001=2]']

>>> all\_q.data

{'slots': ['1', '[uge-exec-001=2]']}

>>> all\_q.data['slots'] = '1,[uge-exec-001=2]'

>>> all\_q.data

{'slots': '1,[uge-exec-001=2]'}

The same is true for corresponding JSON representations.

* Full JSON representation for any PyCL object is based on dictionary that will be formed from metadata dictionary key/value pairs, as well as from UGE object assigned to the reserved key “data” (see Section 3).
* PyCL objects will support JSON representation with both native Python types and with UGE keywords. With native UGE syntax generated JSON strings will have values of “NONE”, “YES” or “NO, rather than null, true or false. Library users will be able to specify JSON representation mode when invoking “to\_json()” method on PyCL objects.

QconfApi class constructor signature is shown below:

\_\_init\_\_(self, sge\_root=None, sge\_cell=’default’, sge\_qmaster\_port=6444,

sge\_execd\_port=6445)

## ClusterQueue

UGE data for ClusterQueue objects (“-sq <queue name>”) are represented as Python dictionaries. All of the keys below are required for add/update operations, and are returned by the get operation:

UGE/PYTHON KEY DEFAULT UGE VALUE DEFAULT PYTHON VALUE

qname template None

hostlist NONE None

seq\_no 0 0

load\_thresholds np\_load\_avg=1.75 'np\_load\_avg=1.75'

suspend\_thresholds NONE None

nsuspend 1 1

suspend\_interval 00:05:00 '00:05:00'

priority 0 0

min\_cpu\_interval 00:05:00 '00:05:00'

qtype BATCH INTERACTIVE 'BATCH INTERACTIVE'

ckpt\_list NONE None

pe\_list make 'make'

jc\_list NO\_JC,ANY\_JC ['NO\_JC','ANY\_JC']

rerun FALSE False

slots 1 1

tmpdir /tmp '/tmp'

shell /bin/csh '/bin/csh'

prolog NONE None

epilog NONE None

shell\_start\_mode unix\_behavior 'unix\_behavior'

starter\_method NONE None

suspend\_method NONE None

resume\_method NONE None

terminate\_method NONE None

notify 00:00:60 '00:00:60'

owner\_list NONE None

user\_lists NONE None

xuser\_lists NONE None

subordinate\_list NONE None

complex\_values NONE None

projects NONE None

xprojects NONE None

calendar NONE None

initial\_state default 'default'

s\_rt INFINITY float('inf')

h\_rt INFINITY float('inf')

d\_rt INFINITY float('inf')

s\_cpu INFINITY float('inf')

h\_cpu INFINITY float(‘inf’)

s\_fsize INFINITY float('inf')

h\_fsize INFINITY float('inf')

s\_data INFINITY float('inf')

h\_data INFINITY float('inf')

s\_stack INFINITY float('inf')

h\_stack INFINITY float('inf')

s\_core INFINITY float('inf')

h\_core INFINITY float('inf')

s\_rss INFINITY float('inf')

h\_rss INFINITY float('inf')

s\_vmem INFINITY float('inf')

h\_vmem INFINITY float('inf')

Sample JSON representation for a ClusterQueue object (shortened for simplicity) is shown below:

>>> all\_q.to\_json()

'{"object\_version": "1.0", "modified\_by": "sveseli", "uge\_cluster": "C1", "object\_class": "ClusterQueue", "modified\_on": "2016-04-15T19:30:07.969048", "data": {"qname": "all.q", "hostlist": "@allhosts", "seq\_no": 0, "load\_thresholds": "np\_load\_avg=1.75",…}}'

UGE data for a list of queues (“-sql”) will be returned within the QconfNameList object, which will support all regular Python list features. Sample JSON representation is shown below:

>>> queue\_list.to\_json()

'{"object\_version": "1.0", "uge\_cluster": "C1", "object\_class": "QconfNameList", "description": "list of queues", "data": ["all.q", "long.q",…]}'

QconfApi class method signatures relevant to ClusterQueue objects are as follows:

ClusterQueue generate\_queue(self, name=None, data=None, metadata=None, json=None,

uge\_version=None, add\_required\_data=True)

ClusterQueue add\_queue(self, pycl\_object=None, name=None, data=None,

metadata=None, json=None)

ClusterQueue modify\_queue(self, pycl\_object=None, name=None, data=None,

metadata=None, json=None)

ClusterQueue get\_queue(self, name)

None delete\_queue(self, name)

QconfNameList list\_queues(self)

Note that in order to generate, add, or modify cluster queue, its name can be specified either explicitly, or as part of other input data (queue object, data dictionary or JSON string).

## Cluster Configuration

UGE data for ClusterConfiguration objects (“-sconf [global | <host name>]”) are represented as Python dictionaries. There are no required keys, and arbitrary non-default keys are allowed for add/update operations. However, the default values are different for the “global” versus the host-specific configuration. This will be handled by two sets of defaults for optional keys.

For the “global” configuration default key/value pairs returned by the get operation are shown below:

UGE/PYTHON KEY DEFAULT UGE VALUE DEFAULT PYTHON VALUE

execd\_spool\_dir $SGE\_ROOT/$SGE\_CELL/spool '$SGE\_ROOT/$SGE\_CELL/spool'

mailer /bin/mail '/bin/mail'

xterm /usr/bin/xterm '/usr/bin/xterm'

load\_sensor none None

prolog none None

epilog none None

shell\_start\_mode unix\_behavior 'unix\_behavior'

login\_shells sh,bash,ksh,csh,tcsh ['sh','bash','ksh','csh','tcsh']

min\_uid 0 0

min\_gid 0 0

user\_lists none None

xuser\_lists none None

projects none None

xprojects none None

default\_jc none None

enforce\_jc false False

enforce\_project false False

enforce\_user auto 'auto'

load\_report\_time 00:00:40 '00:00:40'

max\_unheard 00:04:00 '00:04:00'

reschedule\_unknown 00:00:00 '00:00:00'

loglevel log\_warning 'log\_warning'

administrator\_mail none None

set\_token\_cmd none None

pag\_cmd none None

token\_extend\_time none None

shepherd\_cmd none None

qmaster\_params none None

execd\_params KEEP\_ACTIVE=ERROR 'KEEP\_ACTIVE=ERROR'

reporting\_params accounting=true reporting=false flush\_time=00:00:13 joblog=false sharelog=00:00:00

{'accounting' : True, 'reporting' : False, 'flush\_time' : '00:00:13', 'joblog' : False, 'sharelog' : '00:00:00'}

finished\_jobs 0 0

gid\_range 20000-20100 '20000-20100'

qlogin\_command builtin 'builtin'

qlogin\_daemon builtin 'builtin'

rlogin\_command builtin 'builtin'

rlogin\_daemon builtin 'builtin'

rsh\_command builtin 'builtin'

rsh\_daemon builtin 'builtin'

max\_aj\_instances 2000 2000

max\_aj\_tasks 75000 75000

max\_u\_jobs 0 0

max\_jobs 0 0

max\_advance\_reservations 0 0

auto\_user\_oticket 0 0

auto\_user\_fshare 0 0

auto\_user\_default\_project none None

auto\_user\_delete\_time 86400 86400

delegated\_file\_staging false False

reprioritize 0 0

jsv\_url none None

jsv\_allowed\_mod ac,h,i,e,o,j,M,N,p,w ['ac','h','i','e','o','j','M','N','p', 'w']

cgroups\_params cgroup\_path=none cpuset=false mount=false freezer=false freeze\_pe\_tasks=false killing=false forced\_numa=false h\_vmem\_limit=false m\_mem\_free\_hard=false m\_mem\_free\_soft=false min\_memory\_limit=0

{'cgroup\_path' : None, 'cpuset' : False, 'mount' : False, 'freezer' : False, 'freeze\_pe\_tasks' : False, 'killing' : False, 'forced\_numa' : False, 'h\_vmem\_limit' : False, 'm\_mem\_free\_hard' : False, 'm\_mem\_free\_soft' : False, 'min\_memory\_limit' : 0}

On the other hand, for the host-specific configuration default key/value pairs returned by the get operation are the following:

UGE/PYTHON KEY DEFAULT UGE VALUE DEFAULT PYTHON VALUE

mailer /bin/mail '/bin/mail'

xterm /usr/bin/xterm '/usr/bin/xterm'

By default ClusterConfiguration keys do not include object’s name (“global” or host name). This will be handled by introducing “object\_name” metadata key. Sample JSON representation for a ClusterConfiguration object (shortened for simplicity) is shown below:

>>> global\_conf.to\_json()

'{"object\_version": "1.0", "modified\_by": "sveseli", "uge\_cluster": "C1", "object\_class": "ClusterConfiguration", "object\_name": "global", "modified\_on": "2016-04-15T19:30:07.969048", "data": {"execd\_spool\_dir": "/opt/tools/uge/default/spool",…}}'

UGE data for a list of configurations (“-sconfl”) will be returned within the QconfNameList object (see sample JSON representation below):

>>> conf\_list.to\_json()

'{"object\_version": "1.0", "uge\_cluster": "C1", "object\_class": "QconfNameList", "description": "list of configurations", "data": ["global", "uge-exec-001",…]}'

Relevant QconfApi class method signatures are as follows:

ClusterConfiguration generate\_conf(self, name=None,

data=None, metadata=None, json=None,

uge\_version=None, add\_optional\_data=False)

ClusterConfiguration add\_conf(self, pycl\_object=None,

name=None, data=None, metadata=None, json=None)

ClusterConfiguration modify\_conf(self, pycl\_object=None,

name=None, data=None, metadata=None, json=None)

ClusterConfiguration get\_conf(self, name=None)

None delete\_conf(self, name=None)

QconfNameList list\_confs(self)

## Scheduler Configuration

UGE data for SchedulerConfiguration objects (“-ssconf”) are represented as Python dictionaries. All of the keys below are required for add/update operations, and are returned by the get operation:

UGE/PYTHON KEY DEFAULT UGE VALUE DEFAULT PYTHON VALUE

algorithm default 'default'

schedule\_interval 0:0:15 '0:0:15'

maxujobs 0 0

queue\_sort\_method load 'load'

job\_load\_adjustments np\_load\_avg=0.50 'np\_load\_avg=0.50'

load\_adjustment\_decay\_time 0:7:30 '0:7:30'

load\_formula np\_load\_avg 'np\_load\_avg'

schedd\_job\_info false False

flush\_submit\_sec 1 1

flush\_finish\_sec 1 1

params none None

reprioritize\_interval 0:0:0 '0:0:0'

halftime 168 168

usage\_weight\_list wallclock=0.000000,cpu=1.000000,mem=0.000000,io=0.000000

['wallclock=0.000000', 'cpu=1.000000', 'mem=0.000000' , 'io=0.000000']

compensation\_factor 5.000000 5.0

weight\_user 0.250000 0.25

weight\_project 0.250000 0.25

weight\_department 0.250000 0.25

weight\_job 0.250000 0.25

weight\_tickets\_functional 0 0

weight\_tickets\_share 0 0

share\_override\_tickets TRUE True

share\_functional\_shares TRUE True

max\_functional\_jobs\_to\_schedule 200 200

report\_pjob\_tickets TRUE True

max\_pending\_tasks\_per\_job 50 50

halflife\_decay\_list none None

policy\_hierarchy OFS 'OFS'

weight\_ticket 0.010000 0.01

weight\_waiting\_time 0.000000 0.0

weight\_deadline 3600000.000000 3600000.0

weight\_urgency 0.100000 0.1

weight\_priority 1.000000 1.0

fair\_urgency\_list NONE None

max\_reservation 0 0

default\_duration INFINITY float('inf')

backfilling ON True

prioritize\_preemptees FALSE False

preemptees\_keep\_resources FALSE False

max\_preemptees 0 0

preemption\_distance 00:15:00 '00:15:00'

preemption\_priority\_adjustments none None

Sample JSON representation for a SchedulerConfiguration object (shortened for simplicity) is shown below:

>>> scheduler\_conf.to\_json()

'{"object\_version": "1.0", "modified\_by": "sveseli", "uge\_cluster": "C1", "object\_class": "SchedulerConfiguration", "modified\_on": "2016-04-15T19:30:07.969048", "data": {"algorithm": "default", "schedule\_interval": "0:0:15", "maxujobs": 0, "queue\_sort\_method": "load",…}}'

Relevant QconfApi class method signatures are given as follows (no add/delete/list methods):

SchedulerConfiguration generate\_sconf(self, data=None,

metadata=None, json=None, uge\_version=None, add\_required\_data=True)

SchedulerConfiguration modify\_sconf(self, pycl\_object=None,

data=None, metadata=None, json=None)

SchedulerConfiguration get\_sconf(self)

## Execution Host

UGE data for ExecutionHost objects (“-se <host name>”) are represented as Python dictionaries. All of the keys below are required for add/update operations:

UGE/PYTHON KEY DEFAULT UGE VALUE DEFAULT PYTHON VALUE

hostname template None

load\_scaling NONE None

complex\_values NONE None

user\_lists NONE None

xuser\_lists NONE None

projects NONE None

xprojects NONE None

usage\_scaling NONE None

report\_variables NONE None

license\_constraints NONE None

license\_oversubscription NONE None

Two additional keys (“load\_values” and “processors”) are returned by the get operations. Sample JSON representation for an ExecutionHost object (shortened for simplicity) is given below:

>>> execution\_host.to\_json()

'{"object\_version": "1.0", "modified\_by": "sveseli", "uge\_cluster": "C1", "object\_class": "ExecutionHost", "modified\_on": "2016-04-15T19:30:07.969048", "data": {"hostname": "uge-exec-001", "load\_scaling": null, "complex\_values": "m\_mem\_free=2007.000000M",…}}'

UGE data for a list of execution hosts (“-sel”) will be returned within the QconfNameList object:

>>> execution\_host\_list.to\_json()

'{"object\_version": "1.0", "uge\_cluster": "C1", "object\_class": "QconfNameList", "description": "list of execution hosts", "data": ["uge-exec-001", "uge-exec-002",…]}'

QconfApi class method signatures relevant to ExecutionHost objects are as follows:

ExecutionHost generate\_ehost(self, name=None, data=None,

metadata=None, json=None, uge\_version=None, add\_required\_data=True)

ExecutionHost add\_ehost(self, pycl\_object=None, name=None, data=None,

metadata=None, json=None)

ExecutionHost modify\_ehost(self, pycl\_object=None, name=None, data=None,

metadata=None, json=None)

ExecutionHost get\_ehost(self, name)

None delete\_ehost(self, name)

QconfNameList list\_ehosts(self)

In order to generate, add, or modify execution host, its name can be specified either explicitly, or as part of other input data (data dictionary or JSON string).

## Host Group

UGE data for HostGroup objects (“-shgrp <host group name>”) are represented as Python dictionaries. There are two keys required for add/update operations, and are also returned by the get operations:

UGE/PYTHON KEY DEFAULT UGE VALUE DEFAULT PYTHON VALUE

group\_name @template None

hostlist NONE None

Sample JSON representation for a HostGroup object is shown below:

>>> host\_group.to\_json()

'{"object\_version": "1.0", "modified\_by": "sveseli", "uge\_cluster": "C1", "object\_class": "HostGroup", "modified\_on": "2016-04-15T19:30:07.969048", "data": {"group\_name": "@allhosts", "hostlist": ["uge-exec-001", "uge-exec-002", "uge-exec-003",…]}}'

UGE data for a list of host groups (“-shgrpl”) will be returned within the QconfNameList object:

>>> host\_group\_list.to\_json()

'{"object\_version": "1.0", "uge\_cluster": "C1", "object\_class": "QconfNameList", "description": "list of host groups", "data": ["@allhosts",…]}'

QconfApi class method signatures relevant to HostGroup objects are as follows:

HostGroup generate\_hgrp(self, name=None, data=None, metadata=None,

json=None, uge\_version=None, add\_required\_data=True)

HostGroup add\_hgrp(self, pycl\_object=None, name=None, data=None,

metadata=None, json=None)

HostGroup modify\_hgrp(self, pycl\_object=None, name=None, data=None,

metadata=None, json=None)

HostGroup get\_hgrp(self, name)

None delete\_hgrp(self, name)

QconfNameList list\_hgrps(self)

In order to generate, add, or modify host group, its name can be specified either explicitly, or as part of other input data (data dictionary or JSON string).

## Complex Configuration

UGE data for ComplexConfiguration objects (“-sc”) are represented as Python dictionaries, where keys are complex attribute names and values are dictionaries of attribute data. Built-in complex attributes (shown below) are required, but optional attributes are allowed.

UGE/PYTHON KEY DEFAULT UGE COMPLEX DATA

#name shortcut type relop requestable consumable default urgency aapre

#-----------------------------------------------------------------------------------------------

arch a RESTRING == YES NO NONE 0 NO

calendar c RESTRING == YES NO NONE 0 NO

cpu cpu DOUBLE >= YES NO 0 0 NO

d\_rt d\_rt TIME <= YES NO 0:0:0 0 NO

display\_win\_gui dwg BOOL == YES NO 0 0 NO

h\_core h\_core MEMORY <= YES NO 0 0 NO

h\_cpu h\_cpu TIME <= YES NO 0:0:0 0 NO

h\_data h\_data MEMORY <= YES NO 0 0 NO

h\_fsize h\_fsize MEMORY <= YES NO 0 0 NO

h\_rss h\_rss MEMORY <= YES NO 0 0 NO

h\_rt h\_rt TIME <= YES NO 0:0:0 0 NO

h\_stack h\_stack MEMORY <= YES NO 0 0 NO

h\_vmem h\_vmem MEMORY <= YES NO 0 0 NO

hostname h HOST == YES NO NONE 0 NO

load\_avg la DOUBLE >= NO NO 0 0 NO

load\_long ll DOUBLE >= NO NO 0 0 NO

load\_medium lm DOUBLE >= NO NO 0 0 NO

load\_short ls DOUBLE >= NO NO 0 0 NO

m\_cache\_l1 mcache1 MEMORY <= YES NO 0 0 NO

m\_cache\_l2 mcache2 MEMORY <= YES NO 0 0 NO

m\_cache\_l3 mcache3 MEMORY <= YES NO 0 0 NO

m\_core core INT <= YES NO 0 0 NO

m\_mem\_free mfree MEMORY <= YES YES 0 0 YES

m\_mem\_free\_n0 mfree0 MEMORY <= YES YES 0 0 YES

m\_mem\_free\_n1 mfree1 MEMORY <= YES YES 0 0 YES

m\_mem\_free\_n2 mfree2 MEMORY <= YES YES 0 0 YES

m\_mem\_free\_n3 mfree3 MEMORY <= YES YES 0 0 YES

m\_mem\_total mtotal MEMORY <= YES YES 0 0 YES

m\_mem\_total\_n0 mmem0 MEMORY <= YES YES 0 0 YES

m\_mem\_total\_n1 mmem1 MEMORY <= YES YES 0 0 YES

m\_mem\_total\_n2 mmem2 MEMORY <= YES YES 0 0 YES

m\_mem\_total\_n3 mmem3 MEMORY <= YES YES 0 0 YES

m\_mem\_used mused MEMORY >= YES NO 0 0 NO

m\_mem\_used\_n0 mused0 MEMORY >= YES NO 0 0 NO

m\_mem\_used\_n1 mused1 MEMORY >= YES NO 0 0 NO

m\_mem\_used\_n2 mused2 MEMORY >= YES NO 0 0 NO

m\_mem\_used\_n3 mused3 MEMORY >= YES NO 0 0 NO

m\_numa\_nodes nodes INT <= YES NO 0 0 NO

m\_socket socket INT <= YES NO 0 0 NO

m\_thread thread INT <= YES NO 0 0 NO

m\_topology topo RESTRING == YES NO NONE 0 NO

m\_topology\_inuse utopo RESTRING == YES NO NONE 0 NO

m\_topology\_numa unuma RESTRING == YES NO NONE 0 NO

mem\_free mf MEMORY <= YES NO 0 0 NO

mem\_total mt MEMORY <= YES NO 0 0 NO

mem\_used mu MEMORY >= YES NO 0 0 NO

min\_cpu\_interval mci TIME <= NO NO 0:0:0 0 NO

np\_load\_avg nla DOUBLE >= NO NO 0 0 NO

np\_load\_long nll DOUBLE >= NO NO 0 0 NO

np\_load\_medium nlm DOUBLE >= NO NO 0 0 NO

np\_load\_short nls DOUBLE >= NO NO 0 0 NO

num\_proc p INT == YES NO 0 0 NO

qname q RESTRING == YES NO NONE 0 NO

rerun re BOOL == NO NO 0 0 NO

s\_core s\_core MEMORY <= YES NO 0 0 NO

s\_cpu s\_cpu TIME <= YES NO 0:0:0 0 NO

s\_data s\_data MEMORY <= YES NO 0 0 NO

s\_fsize s\_fsize MEMORY <= YES NO 0 0 NO

s\_rss s\_rss MEMORY <= YES NO 0 0 NO

s\_rt s\_rt TIME <= YES NO 0:0:0 0 NO

s\_stack s\_stack MEMORY <= YES NO 0 0 NO

s\_vmem s\_vmem MEMORY <= YES NO 0 0 NO

seq\_no seq INT == NO NO 0 0 NO

slots s INT <= YES YES 1 1000 YES

swap\_free sf MEMORY <= YES NO 0 0 NO

swap\_rate sr MEMORY >= YES NO 0 0 NO

swap\_rsvd srsv MEMORY >= YES NO 0 0 NO

swap\_total st MEMORY <= YES NO 0 0 NO

swap\_used su MEMORY >= YES NO 0 0 NO

tmpdir tmp RESTRING == NO NO NONE 0 NO

virtual\_free vf MEMORY <= YES NO 0 0 NO

virtual\_total vt MEMORY <= YES NO 0 0 NO

virtual\_used vu MEMORY >= YES NO 0 0 NO

# >#< starts a comment but comments are not saved across edits --------

An example of a Python dictionary representing complex attribute data “slots” from the table above is as follows:

>>> slots\_attribute = {'shortcut' : 's', 'type' : 'INT', 'relop' : '<=', 'requestable' : True, 'consumable' : True, 'default' : 1, 'urgency' : 1000, 'aapre' : True}

Sample JSON representation for a ComplexConfiguration object (shortened for simplicity) is shown below:

>>> complex\_configuration.to\_json()

'{"object\_version": "1.0", "modified\_by": "sveseli", "uge\_cluster": "C1", "object\_class": "ComplexConfiguration", "modified\_on": "2016-04-15T19:30:07.969048", "data": {"slots": {"shortcut" : "s", "type" : "INT", "relop" : "<=", "requestable" : true, "consumable" : true, "default" : 1, "urgency" : 1000, "aapre" : true},…}}'

QconfApi class method signatures relevant to ComplexConfiguration objects are as follows:

ComplexConfiguration generate\_cconf(self, data=None,

metadata=None, json=None, uge\_version=None, add\_required\_data=True)

ComplexConfiguration modify\_cconf(self, pycl\_object=None,

data=None, metadata=None, json=None)

ComplexConfiguration get\_cconf(self)

ComplexConfiguration add\_cattr(self, name, data)

ComplexConfiguration modify\_cattr(self, name, data)

ComplexConfiguration delete\_cattr(self, name)

## Project

UGE data for Project objects (“-sprj <project name>”) are represented as Python dictionaries. Optional keys are not allowed. The keys required for add/update operations, and also returned by the get operations are the following:

UGE/PYTHON KEY DEFAULT UGE VALUE DEFAULT PYTHON VALUE

name template None

oticket 0 0

fshare 0 0

acl NONE None

xacl NONE None

Sample JSON representation for a Project object is shown below:

>>> project.to\_json()

'{"object\_version": "1.0", "modified\_by": "sveseli", "uge\_cluster": "C1", "object\_class": "Project", "modified\_on": "2016-04-15T19:30:07.969048", "data": {"name": "P1", "oticket": 0, "fshare": 0, "acl": null, "xacl": null}}'

UGE data for a list of projects (“-sprjl”) will be returned within the QconfNameList object:

>>> project\_list.to\_json()

'{"object\_version": "1.0", "uge\_cluster": "C1", "object\_class": "QconfNameList", "description": "list of projects", "data": ["P1",…]}'

QconfApi class method signatures relevant to Project objects are as follows:

Project generate\_prj(self, name=None, data=None,

metadata=None, json=None, uge\_version=None, add\_required\_data=True)

Project add\_prj(self, pycl\_object=None, name=None, data=None,

metadata=None, json=None)

Project modify\_prj(self, pycl\_object=None, name=None, data=None,

metadata=None, json=None)

Project get\_prj(self, name)

None delete\_prj(self, name)

QconfNameList list\_prjs(self)

In order to generate, add, or modify project, its name can be specified either explicitly, or as part of other input data (data dictionary or JSON string).

## User

UGE data for User objects (“-suser <user name>”) are represented as Python dictionaries. Optional keys are not allowed. The keys required for add/update operations, and also returned by the get operations are the following:

UGE/PYTHON KEY DEFAULT UGE VALUE DEFAULT PYTHON VALUE

name template None

oticket 0 0

fshare 0 0

delete\_time 0 0

default\_project NONE None

Sample JSON representation for an User object is shown below:

>>> user.to\_json()

'{"object\_version": "1.0", "modified\_by": "sveseli", "uge\_cluster": "C1", "object\_class": "User", "modified\_on": "2016-04-15T19:30:07.969048", "data": {"name": "U1", "oticket": 0, "fshare": 0, "delete\_time": 0, "default\_project": null}}'

UGE data for a list of users (“-suserl”) will be returned within the QconfNameList object:

>>> user\_list.to\_json()

'{"object\_version": "1.0", "uge\_cluster": "C1", "object\_class": "QconfNameList", "description": "list of users", "data": ["U1",…]}'

QconfApi class method signatures relevant to User objects are as follows:

User generate\_user(self, name=None, data=None,

metadata=None, json=None, uge\_version=None, add\_required\_data=True)

User add\_user(self, pycl\_object=None, name=None, data=None, metadata=None,

json=None)

User modify\_user(self, pycl\_object=None, name=None, data=None, metadata=None,

json=None)

User get\_user(self, name)

None delete\_user(self, name)

QconfNameList list\_users(self)

In order to generate, add, or modify user, its name can be specified either explicitly, or as part of other input data (data dictionary or JSON string).

## Access List

UGE data for AccessList objects (“-su <list name>”) are represented as Python dictionaries. Optional keys are not allowed. The keys required for add/update operations, and also returned by the get operations are the following:

UGE/PYTHON KEY DEFAULT UGE VALUE DEFAULT PYTHON VALUE

name None None

type ACL 'ACL'

fshare 0 0

oticket 0 0

entries NONE None

Sample JSON representation for an AccessList object is shown below:

>>> acl.to\_json()

'{"object\_version": "1.0", "modified\_by": "sveseli", "uge\_cluster": "C1", "object\_class": "AccessList", "modified\_on": "2016-04-15T19:30:07.969048", "data": {"name": "arusers", "oticket": 0, "fshare": 0, "type": "ACL", "entries": null}}'

UGE data for a list of access lists (“-sul”) will be returned within the QconfNameList object:

>>> acl\_list.to\_json()

'{"object\_version": "1.0", "uge\_cluster": "C1", "object\_class": "QconfNameList", "description": "list of access lists", "data": ["arusers",…]}'

QconfApi class method signatures relevant to AccessList objects are the following:

AccessList generate\_acl(self, name=None, data=None,

metadata=None, json=None, uge\_version=None, add\_required\_data=True)

AccessList add\_acl(self, pycl\_object=None, name=None, data=None,

metadata=None, json=None)

AccessList modify\_acl(self, pycl\_object=None, name=None, data=None,

metadata=None, json=None)

AccessList get\_acl(self, name)

[AccessList] add\_users\_to\_acls(self, user\_names, access\_list\_names)

[AccessList] delete\_users\_from\_acls(self, user\_names, access\_list\_names)

None delete\_acl(self, name)

QconfNameList list\_acls(self)

In order to generate, add, or modify access list, its name can be specified either explicitly, or as part of other input data (data dictionary or JSON string).

## Job Class

UGE data for JobClass objects (“-sjc <job class name>”) are represented as Python dictionaries. Optional keys are not allowed. The keys required for add/update operations, and also returned by the get operations are the following:

UGE/PYTHON KEY DEFAULT UGE VALUE DEFAULT PYTHON VALUE

jcname template None

variant\_list NONE None

owner NONE None

user\_lists NONE None

xuser\_lists NONE None

A {+}UNSPECIFIED '{+}UNSPECIFIED'

a {+}UNSPECIFIED '{+}UNSPECIFIED'

ar {+}UNSPECIFIED '{+}UNSPECIFIED'

b {+}UNSPECIFIED '{+}UNSPECIFIED'

binding {+}UNSPECIFIED '{+}UNSPECIFIED'

c\_interval {+}UNSPECIFIED '{+}UNSPECIFIED'

c\_occasion {+}UNSPECIFIED '{+}UNSPECIFIED'

CMDNAME {+}UNSPECIFIED '{+}UNSPECIFIED'

CMDARG {+}UNSPECIFIED '{+}UNSPECIFIED'

ckpt {+}UNSPECIFIED '{+}UNSPECIFIED'

ac {+}UNSPECIFIED '{+}UNSPECIFIED'

cwd {+}UNSPECIFIED '{+}UNSPECIFIED'

dl {+}UNSPECIFIED '{+}UNSPECIFIED'

e {+}UNSPECIFIED '{+}UNSPECIFIED'

h {+}UNSPECIFIED '{+}UNSPECIFIED'

hold\_jid {+}UNSPECIFIED '{+}UNSPECIFIED'

hold\_jid\_ad {+}UNSPECIFIED '{+}UNSPECIFIED'

i {+}UNSPECIFIED '{+}UNSPECIFIED'

j {+}UNSPECIFIED '{+}UNSPECIFIED'

js {+}UNSPECIFIED '{+}UNSPECIFIED'

l\_hard {+}UNSPECIFIED '{+}UNSPECIFIED'

l\_soft {+}UNSPECIFIED '{+}UNSPECIFIED'

masterl {+}UNSPECIFIED '{+}UNSPECIFIED'

m {+}UNSPECIFIED '{+}UNSPECIFIED'

mbind {+}UNSPECIFIED '{+}UNSPECIFIED'

M {+}UNSPECIFIED '{+}UNSPECIFIED'

masterq {+}UNSPECIFIED '{+}UNSPECIFIED'

N {+}UNSPECIFIED '{+}UNSPECIFIED'

notify {+}UNSPECIFIED '{+}UNSPECIFIED'

now {+}UNSPECIFIED '{+}UNSPECIFIED'

o {+}UNSPECIFIED '{+}UNSPECIFIED'

P {+}UNSPECIFIED '{+}UNSPECIFIED'

p {+}UNSPECIFIED '{+}UNSPECIFIED'

pe\_name {+}UNSPECIFIED '{+}UNSPECIFIED'

pe\_range {+}UNSPECIFIED '{+}UNSPECIFIED'

q\_hard {+}UNSPECIFIED '{+}UNSPECIFIED'

q\_soft {+}UNSPECIFIED '{+}UNSPECIFIED'

R {+}UNSPECIFIED '{+}UNSPECIFIED'

r {+}UNSPECIFIED '{+}UNSPECIFIED'

rou {+}UNSPECIFIED '{+}UNSPECIFIED'

S {+}UNSPECIFIED '{+}UNSPECIFIED'

shell {+}UNSPECIFIED '{+}UNSPECIFIED'

t {+}UNSPECIFIED '{+}UNSPECIFIED'

tc {+}UNSPECIFIED '{+}UNSPECIFIED'

V {+}UNSPECIFIED '{+}UNSPECIFIED'

v {+}UNSPECIFIED '{+}UNSPECIFIED'

Sample JSON representation for a JobClass object (shortened for simplicity) is shown below:

>>> job\_class.to\_json()

'{"object\_version": "1.0", "modified\_by": "sveseli", "uge\_cluster": "C1", "object\_class": "JobClass", "modified\_on": "2016-04-15T19:30:07.969048", "data": {"jcname": "JC1", "variant\_list": null,…, "v": "{+}UNSPECIFIED"}}'

UGE data for a list of job classes (“-sjcl”) will be returned within the QconfNameList object:

>>> job\_class\_list.to\_json()

'{"object\_version": "1.0", "uge\_cluster": "C1", "object\_class": "QconfNameList", "description": "list of job classes", "data": ["JC1",…]}'

QconfApi class method signatures relevant to JobClass objects are the following:

JobClass generate\_jc(self, name=None, data=None,

metadata=None, json=None, uge\_version=None, add\_required\_data=True)

JobClass add\_jc(self, pycl\_object=None, name=None, data=None,

metadata=None, json=None)

JobClass modify\_jc(self, pycl\_object=None, name=None, data=None,

metadata=None, json=None)

JobClass get\_jc(self, name)

None delete\_jc(self, name)

QconfNameList list\_jcs(self)

In order to generate, add, or modify job class, its name can be specified either explicitly, or as part of other input data (data dictionary or JSON string).

## Resource Quota Set

UGE data for ResourceQuotaSet objects (“-srqs <set name>”) are represented as Python dictionaries. Optional keys are not allowed. The keys required for add/update operations, and also returned by the get operations are the following:

UGE/PYTHON KEY DEFAULT UGE VALUE DEFAULT PYTHON VALUE

name template None

description NONE None

enabled FALSE False

limit to slots=0 'to slots=0'

Note that the “limit” keyword designates rule definition. Each RQS may have multiple rule definitions, which will be indicated by a Python list.

Sample JSON representation for a ResourceQuotaSet object is shown below:

>>> rqs.to\_json()

'{"object\_version": "1.0", "modified\_by": "sveseli", "uge\_cluster": "C1", "object\_class": "ResourceQuotaSet", "modified\_on": "2016-04-15T19:30:07.969048", "data": {"name": "RQS1", "description": null, "enabled": false, "limit": ["users {user1,user2} hosts {@lx\_host} to virtual\_free=6g", "users {\*} hosts {@lx\_host} to virtual\_free=4g"]}}'

UGE data for a list of resource quota sets (“-srqsl”) will be returned within the QconfNameList object:

>>> rqs\_list.to\_json()

'{"object\_version": "1.0", "uge\_cluster": "C1", "object\_class": "QconfNameList", "description": "list of resource quota sets", "data": ["RQS1",…]}'

QconfApi class method signatures relevant to ResourceQuotaSet objects are the following:

ResourceQuotaSet generate\_rqs(self, name=None, data=None,

metadata=None, json=None, uge\_version=None, add\_required\_data=True)

ResourceQuotaSet add\_rqs(self, pycl\_object=None, name=None,

data=None, metadata=None, json=None)

ResourceQuotaSet modify\_rqs(self, pycl\_object=None, name=None,

data=None, metadata=None, json=None)

ResourceQuotaSet get\_rqs(self, name)

None delete\_rqs(self, name)

QconfNameList list\_rqss(self)

In order to generate, add, or modify resource quota set, its name can be specified either explicitly, or as part of other input data (data dictionary or JSON string).

## Parallel Environment

UGE data for ParallelEnvironment objects (“-sp <name>”) are represented as Python dictionaries. Optional keys are not allowed. The keys required for add/update operations, and also returned by the get operations are the following:

UGE/PYTHON KEY DEFAULT UGE VALUE DEFAULT PYTHON VALUE

pe\_name None None

slots 0 0

used\_slots 0 0

bound\_slots 0 0

user\_lists NONE None

xuser\_lists NONE None

start\_proc\_args NONE None

stop\_proc\_args NONE None

allocation\_rule $pe\_slots '$pe\_slots'

control\_slaves FALSE None

job\_is\_first\_task TRUE True

urgency\_slots min 'min'

accounting\_summary FALSE False

daemon\_forks\_slaves FALSE False

master\_forks\_slaves FALSE False

Sample JSON representation for a ParallelEnvironment object, shortened for simplicity, is shown below:

>>> pe.to\_json()

'{"object\_version": "1.0", "modified\_by": "sveseli", "uge\_cluster": "C1", "object\_class": "ParallelEnvironment", "modified\_on": "2016-04-15T19:30:07.969048", "data": {"pe\_name": "PE1", "slots": 100,…, "master\_forks\_slaves": false}}'

UGE data for a list of parallel environments (“-spl”) will be returned within the QconfNameList object:

>>> pe\_list.to\_json()

'{"object\_version": "1.0", "uge\_cluster": "C1", "object\_class": "QconfNameList", "description": "list of parallel environments", "data": ["PE1",…]}'

QconfApi class method signatures relevant to ParallelEnvironment objects are the following:

ParallelEnvironment generate\_pe(self, name=None, data=None,

metadata=None, json=None, uge\_version=None, add\_required\_data=True)

ParallelEnvironment add\_pe(self, pycl\_object=None, name=None,

data=None, metadata=None, json=None)

ParallelEnvironment modify\_pe(self, pycl\_object=None, name=None,

data=None, metadata=None, json=None)

ParallelEnvironment get\_pe(self, name)

None delete\_pe(self, name)

QconfNameList list\_pes(self)

In order to generate, add, or modify parallel environment, its name can be specified either explicitly, or as part of other input data (data dictionary or JSON string).

## Share Tree

UGE data for ShareTree object (“-sstree”) is represented as list of Python dictionaries, where dictionaries represent tree nodes. The keys that define tree node completely are “id”, “name”, “type”, “shares”, and “childnodes”. However, tree nodes can be added by specifying only path and number of shares.

Sample JSON representation for a ShareTree object is shown below:

>>> share\_tree.to\_json()

'{"object\_version": "1.0", "modified\_by": "sveseli", "uge\_cluster": "C1", "object\_class": "ShareTree", "modified\_on": "2016-04-15T19:30:07.969048", "data": [{"childnodes": [1,2], "type": 0, "id": 0, "shares": 1, "name": "Root"}, {"childnodes": "NONE", "type": 0, "id": 1, "shares": 10, "name": "U1"}, {"childnodes": "NONE", "type": 0, "id": 2, "shares": 25, "name": "U2"}]}'

QconfApi class method signatures relevant to ShareTree objects are the following:

ShareTree generate\_stree(self, data=None, metadata=None, json=None,

uge\_version=None)

ShareTree add\_stree(self, pycl\_object=None, data=None, metadata=None,

json=None)

ShareTree modify\_stree(self, pycl\_object=None, data=None, metadata=None,

json=None)

ShareTree modify\_or\_add\_stree(self, pycl\_object=None, data=None, metadata=None,

json=None)

ShareTree get\_stree(self)

ShareTree get\_stree\_if\_exists(self)

None delete\_stree(self)

None delete\_stree\_if\_exists(self)

ShareTree add\_stnode(self, path, shares)

ShareTree delete\_stnode(self, path)

In the above list methods modify\_or\_add\_stree(), get\_stree\_if\_exists(), and delete\_stree\_if\_exists() do not throw exception if share tree does not exist.

## Calendar

UGE data for Calendar objects (“-scal <calendar name>”) are represented as Python dictionaries. Optional keys are not allowed. The keys required for add/update operations, and also returned by the get operations are the following:

UGE/PYTHON KEY DEFAULT UGE VALUE DEFAULT PYTHON VALUE

calendar\_name None None

year NONE None

week NONE None

Sample JSON representation for a Calendar object is shown below:

>>> calendar.to\_json()

'{"object\_version": "1.0", "modified\_by": "sveseli", "uge\_cluster": "C1", "object\_class": "Calendar", "modified\_on": "2016-04-15T19:30:07.969048", "data": {"calendar\_name": "CAL1", "year": null, "week": "1-2"}}'

UGE data for a list of calendars (“-scall”) will be returned within the QconfNameList object:

>>> calendar\_list.to\_json()

'{"object\_version": "1.0", "uge\_cluster": "C1", "object\_class": "QconfNameList", "description": "list of calendars", "data": ["CAL1",…]}'

QconfApi class method signatures relevant to Calendar objects are the following:

Calendar generate\_cal(self, name=None, data=None, metadata=None, json=None,

uge\_version=None, add\_required\_data=True)

Calendar add\_cal(self, pycl\_object=None, name=None, data=None,

metadata=None, json=None)

Calendar modify\_cal(self, pycl\_object=None, name=None, data=None,

metadata=None, json=None)

Calendar get\_cal(self, name)

None delete\_cal(self, name)

QconfNameList list\_cals(self)

In order to generate, add, or modify calendar, its name can be specified either explicitly, or as part of other input data (data dictionary or JSON string).

## Checkpointing Environment

UGE data for CheckpointingEnvironment objects (“-sckpt <name>”) are represented as Python dictionaries. Optional keys are not allowed. The keys required for add/update operations, and also returned by the get operations are the following:

UGE/PYTHON KEY DEFAULT UGE VALUE DEFAULT PYTHON VALUE

ckpt\_name None None

interface userdefined 'userdefined'

ckpt\_command none None

migr\_command none None

restart\_command none None

clean\_command none None

ckpt\_dir /tmp '/tmp'

signal none None

when sx 'sx'

Sample JSON representation for a CheckpointingEnvironment object is shown below:

>>> ckpt.to\_json()

'{"object\_version": "1.0", "modified\_by": "sveseli", "uge\_cluster": "C1", "object\_class": "CheckpointingEnvironment", "modified\_on": "2016-04-15T19:30:07.969048", "data": {"ckpt\_name": "CKPT1", "interface": "userdefined",…, "when": "sx"}}'

UGE data for a list of checkpointing environments (“-sckptl”) will be returned within the QconfNameList object:

>>> ckpt\_list.to\_json()

'{"object\_version": "1.0", "uge\_cluster": "C1", "object\_class": "QconfNameList", "description": "list of checkpointing environments", "data": ["CKPT1",…]}'

QconfApi class method signatures relevant to CheckpointingEnvironment objects are the following:

CheckpointingEnvironment generate\_ckpt(self, name=None,

data=None, metadata=None, json=None, uge\_version=None, add\_required\_data=True)

CheckpointingEnvironment add\_ckpt(self, pycl\_object=None,

name=None, data=None, metadata=None, json=None)

CheckpointingEnvironment modify\_ckpt(self, pycl\_object=None,

name=None, data=None, metadata=None, json=None)

CheckpointingEnvironment get\_ckpt(self, name)

None delete\_ckpt(self, name)

QconfNameList list\_ckpts(self)

In order to generate, add, or modify checkpointing environment, its name can be specified either explicitly, or as part of other input data (data dictionary or JSON string).

## Managers

List of UGE managers (“-sm”) will be represented using QconfNameList object. Sample JSON representation is shown below:

>>> manager\_list.to\_json()

'{"object\_version": "1.0", "uge\_cluster": "C1", "object\_class": "QconfNameList", "description": "list of managers", "data": ["ugeadmin", "sveseli"]}'

Relevant QconfApi class method signatures are the following:

QconfNameList list\_managers(self)

QconfNameList add\_managers(self, manager\_names)

QconfNameList delete\_managers(self, manager\_names)

Methods for adding and deleting managers will return list of names after requested operation has been completed. In other words, the “add” method will return list that contains old names together with names that have been added, while the “delete” method will return list of old names without names that have been deleted.

## Operators

List of UGE operators (“-so”) will be represented using QconfNameList object. Sample JSON representation is shown below:

>>> operator\_list.to\_json()

'{"object\_version": "1.0", "uge\_cluster": "C1", "object\_class": "QconfNameList", "description": "list of operators", "data": ["ugeadmin", "sveseli"]}'

Relevant QconfApi class method signatures are the following:

QconfNameList list\_operators(self)

QconfNameList add\_operators(self, operator\_names)

QconfNameList delete\_operators(self, operator\_names)

Methods for adding and deleting operators will return list of names after requested operation has been completed. In other words, the “add” method will return list that contains old names together with names that have been added, while the “delete” method will return list of old names without names that have been deleted.

## Submit Hosts

List of UGE submit hosts (“-ss”) will be represented using QconfNameList object. Sample JSON representation is shown below:

>>> submit\_host\_list.to\_json()

'{"object\_version": "1.0", "uge\_cluster": "C1", "object\_class": "QconfNameList", "description": "list of submit hosts", "data": ["shost-001", "shost-002"]}'

Relevant QconfApi class method signatures are the following:

QconfNameList list\_shosts(self)

QconfNameList add\_shosts(self, host\_names)

QconfNameList delete\_shosts(self, host\_names)

Methods for adding and deleting submit hosts will return list of names after requested operation has been completed. In other words, the “add” method will return list that contains old names together with names that have been added, while the “delete” method will return list of old names without names that have been deleted.

## Admin Hosts

List of UGE admin hosts (“-sh”) will be represented using QconfNameList object. Sample JSON representation is shown below:

>>> admin\_host\_list.to\_json()

'{"object\_version": "1.0", "uge\_cluster": "C1", "object\_class": "QconfNameList", "description": "list of admin hosts", "data": ["ahost-001", "ahost-002"]}'

Relevant QconfApi class method signatures are the following:

QconfNameList list\_ahosts(self)

QconfNameList add\_ahosts(self, host\_names)

QconfNameList delete\_ahosts(self, host\_names)

Methods for adding and deleting admin hosts will return list of names after requested operation has been completed. In other words, the “add” method will return list that contains old names together with names that have been added, while the “delete” method will return list of old names without names that have been deleted.