Query Python API

tractor.api.query is a Python API which provides the same query and control functionality as provided by the command line tool, tq. It is installed with the Python interpreter that ships with Tractor, rmanpy.

You may be able to use this module with your own interpreter, but that will require site-specific configuration to locate or install the Tractor Python modules appropriately.

The following examples assume that the query module has been imported as follows:

```
import tractor.api.query as tq
```

List Functions

The query module provides functions for retrieving info for jobs, tasks, commands, invocations, and blades. They are named the same as the list commands found in the command line tool, namely jobs(), tasks(), commands(), invocations(), and blades().

The first argument to these functions is a search clause that is used to narrow the search.

In this example, this list function fetches all of the jobs owned by a group of users that were spooled over 8 days ago. The return value of the list functions is a list of dictionaries, each dictionary representing one of the matched items.

```
>>> jobs = tq.jobs("owner in [freddie brian john roger] and spooltime < -8d")
>>> print jobs[0]
{u'comment': u'', u'maxslots': 0, u'numblocked': 0, u'spoolcwd': u'/home/freddie/unitard, u'numerror': 0,
u'aftertime': None, u'elapsedsecs': 0.0, u'owner': u'paul', u'spoolhost': u'zanzibar, u'spooladdr':
u'138.72.196.92', u'jid': 2680, u'service': u'', u'title': u'Somebody To Render', u'numactive': 0,
u'editpolicy': u'', u'spooltime': u'2014-02-17 08:16:53-08', u'projects': [u'flash_gordon', u'hot_space'],
u'crews': [u'queen'], u'maxcid': 2, u'metadata': u'', u'numready': 0, u'tags': [], u'afterjids': [],
u'stoptime': u'2014-02-17 08:16:57.971-08', u'envkey': [], u'tier': u'', u'etalevel': 1, u'numtasks': 2,
u'numdone': 2, u'maxtid': 2, u'assignments': u'', u'spoolfile': u'/home/freddie/champion.alf',
u'esttotalsecs': 0.0, u'starttime': u'2014-02-17 08:16:53.73-08', u'pausetime': None, u'minslots': 0,
u'deletetime': None, u'priority': 10.0}
```

Search Clause

The search clause is a boolean expression (the above example uses and) composed of comparisons and operators (*in*, <) of the entity's attributes (owner, spooltime). Accommodations are made to reduce the amount of typing. For example, string literals like freddie is not enclosed in quotes, and the string -8d is a short form meaning "one day ago".

More details on the search clause syntax can be found here.

Note that the broader the search, the more expensive it will be in terms of CPU, memory, and bandwidth for locating, transmitting, and storing the results. This can affect the client, engine, and backend database server.

Columns

By default, the query functions will retrieve all attributes for a given entity. However, this can waste memory and cpu on the client, engine, and backend database if all attributes are not required. The list of retrieved attributes can be specified with the columns= keyword parameter.

For example, this fetches only the job id, owner, title, and metadata columns:

```
>>> tq.jobs("active", columns=["jid", "owner", "title", "metadata"])
```

Sorting

Results can be ordered using the sortby= keyword parameter. Multiple attributes can be specified to indicate secondary sorting. Items are ordered in ascending order, unless the attribute is prepended with - to indicate descending order.

For example, to show the order in which similar jobs would be processed in a FIFO queue, we can list them in priority descending, spooltime ascending order as follows:

```
>>> tq.jobs("active or ready", sortby=["-priority", "spooltime"])
```

While item sorting could be done client side, it is not possible to do so if the result set was truncated with the limit= keyword parameter because truncation will happen server side before the client has a chance to sort the results.

Be aware that sorting could have an impact on the backend database server depending on the complexity and scope of the query.

Limiting

The limit= keyword parameter sets the upper bound on the number of items returned by the engine.

For example, this statement will fetch the ten jobs with the most number of active tasks on the farm:

```
>>> tq.jobs("active", sortby=["-numactive"], limit=10)
```

This statement lists the twenty hosts with the least amount of disk space.

```
>>> tq.blades("name like east", sortby=["availdisk"], limit=20)
```

Be aware that setting the limit to 0 places *NO LIMIT* on the number of records returned.

Note that there is a system default upper bound in effect when the limit= keyword parameter is not used. This upper bound is 2500, though admins may override this by setting SiteMaxListReplyCount in tractor.config.

Affiliation

In the above examples, attribute of items are used in search clauses, column specifications, and sorting. Attributes of affiliated entities can also be specified in all of these cases by prefixing the attribute with the entity name. For example, the following statement fetches the active tasks owned by walt.

```
>>> tq.tasks("state=active and Job.owner=walt")
```

Similarly, the following command displayed all error tasks from jobs spooled within the last 24 hours, including their jobs' titles and spooltimes, and sorted by job priority.

```
>>> tq.tasks("state=error and Job.spooltime > -24h", columns=["Job.spooltime", "Job.title"], sortby=["Job.
priority"])
```

Because spooltime and priority are attributes of no other entity than a job, the entity prefix is not required. The following is equivalent:

```
>>> tq.tasks("state=error and spooltime > -24h", columns=["spooltime", "title"], sortby=["priority"])
```

Archives

The archives= keyword parameter can be set to True in order to search the records of jobs that have been deleted (and its associated tasks, commands, and invocations).

Be aware that archives can be *much* bigger than the live data set, so queries can be *MUCH* more expensive to execute, transmit, and store in memory.

Such expense can be avoided by:

- using well-specified search clauses to ensure small result sets,
- not sorting on the server for larger sets, and
- using the limit argument and no sorting for potentially large result sets.

Operations

Not only does the query module have functions to fetch items, it has the ability to operate on them. **BE CAREFUL** using these operation functions because they are **VERY** powerful and can be **VERY VERY VERY DESTRUCTIVE**.

The first argument to an operation specifies which items are to be operated on, and can be any of the following:

- a search clause,
- a list of objects returned by a query function,
- a single object from a list returned by a list function,
- a dictionary specifying the required attributes for the operation, or
- a list of dictionaries specifying the required attributes for the operation

For example, all of the following are equivalent:

```
>>> tq.retry("jid=123 and tid=1")
>>> tq.retry(tq.tasks("jid=123 and tid=1"))
>>> tq.retry(tq.tasks("jid=123 and tid=1")[0])
>>> tq.retry({"jid": 123, "tid": 1})
>>> tq.retry([{"jid": 123, "tid": 1}])
```

The last two use cases permit authors to construct their own objects to identify which items to affect. Different attributes will be required in the dictionary, depending on the entity being operated on. Job operations require jid; task operations, jid and tid; command operations, jid and cid; blade operations, name.

This statement pauses all active jobs with errors:

```
>>> tq.pause("active and error")
```

This statement nimbies the ten blades with the least amount of disk space:

```
>>> tq.nimby("up and not nimby", sortby=["availdisk"], limit=10)
```

Some operations have additional keyword arguments. For example, when changing the priority of a job, the prioriy= keyword parameter used to specify the priority. The following statement changes the priority of all of george's to 500.

```
>>> tq.chpri("owner=george", priority=500)
```

Connectivity

The Python module silently handles your engine connectivity, lazily establishing a session with the engine.

The default engine hostname, port, connected user, password, and debug boolean flag can be changed using setEngineClientParam(). The default engine hostname will be the TRACTOR_ENGINE environment variable value, or tractor-engine if it is not set. The default port is 80. Depending on how EngineDiscovery has been configured, an existing engine may be automatically discovered without using this function.

The default user is the USER environment variable value, or root if it is not set. The default password is not set. The default debug flag is set to False.

The default password is the TRACTOR_PASSWORD environment variable value; otherwise, it is not set.

The default debug flag is the TRACTOR_DEBUG environment variable value; othewise, it is False.

This example overrides some of the default connection values. It is important to do this *before* running any list or operation functions.

```
>>> tq.setEngineClientParam(hostname="test-engine", port=8000, user="donovan", debug=True)
```

If your sites uses passwords, you can prompt the user for their password and set the used value as follows:

```
>>> the_password = getpass.getpass()  # the_password may be obtained by other means
>>> tq.setEngineClientParam(password=the_password)
```

You can avoid having the user enter their password each time the script runs by having session information cached in a file using thesessionFilename= parameter. The file name would typically be scoped by application name, user name, and host name.needsPassword() would be called to determine whether the cached session id is valid or whether the user needs to be prompted for their password. The following is a full example:

```
>>> import os
>>> import socket
>>> import getpass
>>> import tractor.api.query as tq
>>>
>>> home = os.path.expanduser("~")
>>> app = os.path.basename(__file__)
>>> user = getpass.getuser()
>>> host = socket.gethostname()
>>>
>>> # qualify the session filename by host if it will be stored in a location shared my multiple hosts
>>> sessionFilename = os.path.join(home, ".{app}.{user}.{host}.session".format(home=home, app=app, user=user,
host=host))
>>>
>>> # set the session filename
>>> tq.setEngineClientParam(sessionFilename=sessionFilename)
>>>
>>> # check if the password needs to be obtained; this block will be skipped if the session filen contains a
valid session id
>>> if tq.needsPassword():
>>>     password = getpass.getpass()
>>>     tq.setEngineClientParam(password=password)
>>>
>>> # the session file will be created when the first successful API call is made
>>> jobs = tq.jobs("error")
>>> print("There are {numjobs} jobs with errors.".format(numjobs=len(jobs)))
```

You can terminate the session by calling closeEngineClient(). This is helpful for reducing the number of sessions maintained by the engine. This forces the session to be no longer valid, so the password will need to be reacquired by the app the next time it is run.

```
>>> tq.closeEngineClient()
```

More Help

The internal module documentation will show all of the list and operation functions available in the query module.

```
>>> help(tq)
```