```python
# -*- coding: utf-8 -*-
# TODO: WIP WIP WIP WIP!
# I got another stage of this library aside, but this is perfectly usable with some restrictions :)
import json
import logging
import re
import requests
import simplejson
import six
import iso8601
from copy import copy
from distutils.version import LooseVersion
from functools import partial
from wait_for import wait_for

from .filters import Q


class APIException(Exception):
    pass


class ManageIQClient(object):
    def __init__(self, entry_point, auth, logger=None, verify_ssl=True, ca_bundle_path=None):
        """ If ca_bundle_path is specified it replaces the system's trusted root CAs"""
        self._entry_point = entry_point
        self._auth = auth
        self._verify_ssl = verify_ssl
        self._ca_bundle_path = ca_bundle_path
        self._session = requests.Session()
        self._session.headers.update({'Content-Type': 'application/json; charset=utf-8'})
        self._build_auth(auth)
        if not verify_ssl:
            self._session.verify = False
        elif ca_bundle_path:
            self._session.verify = ca_bundle_path
        self.logger = logger or logging.getLogger(__name__)
        self.response = None
        self._load_data()

    def _build_auth(self, auth):
        valid = False
        if isinstance(auth, dict):
            if set(("user", "password")) <= set(auth):
                self._session.auth = (auth["user"], auth["password"])
                valid = True
            if "token" in auth:
                self._session.headers.update({'X-Auth-Token': auth['token']})
                valid = True
        elif isinstance(auth, (tuple, list)):  # for backward compatibility
            self._session.auth = tuple(auth[:2])
            valid = True

        if not valid:
            raise ValueError("Unknown values provided for auth")

    def _load_data(self):
        data = self.get(self._entry_point)
        self.collections = CollectionsIndex(self, data.pop("collections", []))
        self._version = data.pop("version", None)
        self._versions = {}
        for version in data.pop("versions", []):
            self._versions[version["name"]] = version["href"]
        for key, value in data.items():
            setattr(self, key, value)

    @property
    def version(self):
        return self._version

    def _result_processor(self, result):
        # Save last full response
        self.response = result

        result_json = None
        try:
            result_json = result.json()
        except simplejson.scanner.JSONDecodeError:
            result_text = result.text.strip()
            # HTTP methods other than GET and OPTIONS are allowed to return empty result
            if result.request.method in ("GET", "OPTIONS") or result_text:
                raise APIException("JSONDecodeError: {}".format(result_text or "empty result"))

        if result_json and "error" in result_json:
            if isinstance(result_json["error"], dict):
                raise APIException(
```

```
                    "{}: {}".format(result_json["error"]["klass"], result_json["error"]["message"]))
            else:
                raise APIException(
                    "{}: {}".format(result_json.get("status", None), result_json["error"]))

        # Check HTTP response status
        if not result:
            raise APIException("The request failed with HTTP status {}: {}".format(
                result.status_code, result.reason))

        return result_json

    def _sending_request(self, func, retries=2):
        while retries:
            try:
                return func()
            except requests.ConnectionError as e:
                last_connection_exception = e
                retries -= 1
        raise last_connection_exception

    def get(self, api_endpoint_url=None, **get_params):
        if not api_endpoint_url:
            if "url" not in get_params:
                raise ValueError(
                    "url must be specified, as positional parameter, "
                    "or (deprecated) 'url' keyword parameter"
                )
            api_endpoint_url = get_params.pop("url")

        self.logger.info("[RESTAPI] GET %s %r", api_endpoint_url, get_params)
        data = self._sending_request(
            partial(self._session.get, api_endpoint_url, params=get_params))
        return self._result_processor(data)

    def post(self, api_endpoint_url=None, **payload):
        if not api_endpoint_url:
            if "url" not in payload:
                raise ValueError(
                    "url must be specified, as positional parameter, "
                    "or (deprecated) 'url' keyword parameter"
                )
            api_endpoint_url = payload.pop("url")

        self.logger.info("[RESTAPI] POST %s %r", api_endpoint_url, payload)
        data = self._sending_request(
            partial(self._session.post, api_endpoint_url, data=json.dumps(payload)))
        self.logger.info("[RESTAPI] RESPONSE %s", data)
        return self._result_processor(data)

    def put(self, api_endpoint_url=None, **payload):
        if not api_endpoint_url:
            if "url" not in payload:
                raise ValueError(
                    "url must be specified, as positional parameter, "
                    "or (deprecated) 'url' keyword parameter"
                )
            api_endpoint_url = payload.pop("url")

        self.logger.info("[RESTAPI] PUT %s %r", api_endpoint_url, payload)
        data = self._sending_request(
            partial(self._session.put, api_endpoint_url, data=json.dumps(payload)))
        self.logger.info("[RESTAPI] RESPONSE %s", data)
        return self._result_processor(data)

    def patch(self, url, *payload):
        self.logger.info("[RESTAPI] PATCH %s %r", url, payload)
        data = self._sending_request(
            partial(self._session.patch, url, data=json.dumps(payload)))
        self.logger.info("[RESTAPI] RESPONSE %s", data)
        return self._result_processor(data)

    def delete(self, api_endpoint_url=None, **payload):
        if not api_endpoint_url:
            if "url" not in payload:
                raise ValueError(
                    "url must be specified, as positional parameter, "
                    "or (deprecated) 'url' keyword parameter"
                )
            api_endpoint_url = payload.pop("url")

        self.logger.info("[RESTAPI] DELETE %s %r", api_endpoint_url, payload)
        data = self._sending_request(
            partial(self._session.delete, api_endpoint_url, data=json.dumps(payload)))
        self.logger.info("[RESTAPI] RESPONSE %s", data)
        return self._result_processor(data)
```

```
173
174     def options(self, api_endpoint_url=None, **opt_params):
175         if not api_endpoint_url:
176             if "url" not in opt_params:
177                 raise ValueError(
178                     "url must be specified, as positional parameter, "
179                     "or (deprecated) 'url' keyword parameter"
180                 )
181             api_endpoint_url = opt_params.pop("url")
182
183         self.logger.info("[RESTAPI] OPTIONS %s %r", api_endpoint_url, opt_params)
184         data = self._sending_request(
185             partial(self._session.options, api_endpoint_url, params=opt_params))
186         return self._result_processor(data)
187
188     def get_entity(self, collection_or_name, entity_id, attributes=None):
189         if not isinstance(collection_or_name, Collection):
190             collection = Collection(
191                 self, "{}/{}".format(self._entry_point, collection_or_name), collection_or_name)
192         else:
193             collection = collection_or_name
194         entity = Entity(
195             collection,
196             {"href": "{}/{}".format(collection._href, entity_id)},
197             attributes=attributes)
198         return entity
199
200     def api_version(self, version):
201         return type(self)(
202             self._versions[version],
203             self._auth,
204             logger=self.logger,
205             verify_ssl=self._verify_ssl,
206             ca_bundle_path=self._ca_bundle_path)
207
208     @property
209     def versions(self):
210         return sorted(self._versions.keys(), reverse=True, key=LooseVersion)
211
212     @property
213     def latest_version(self):
214         return self.versions[0]
215
216     @property
217     def on_latest_version(self):
218         return self.version == self.latest_version
219
220
221 class CollectionsIndex(object):
222     def __init__(self, api, data):
223         self._api = api
224         self._data = data
225         self._collections = []
226         self._load_data()
227
228     def _load_data(self):
229         for collection in self._data:
230             c = Collection(
231                 self._api, collection["href"], collection["name"], collection["description"])
232             setattr(self, collection["name"], c)
233             self._collections.append(c)
234
235     @property
236     def all(self):
237         return self._collections
238
239     @property
240     def all_names(self):
241         return [c.name for c in self.all]
242
243     def __contains__(self, collection):
244         if isinstance(collection, six.string_types):
245             return collection in self.all_names
246         else:
247             return collection in self.all
248
249
250 class SearchResult(object):
251     def __init__(self, collection, data):
252         self.collection = collection
253         self.count = data.pop("count", 0)
254         self.subcount = data.pop("subcount", 0)
255         self.name = data.pop("name")
256         self.resources = []
257         for resource in data["resources"]:
258             self.resources.append(Entity(collection, resource))
```

```
259
260     def __iter__(self):
261         for resource in self.resources:
262             resource.reload()
263             yield resource
264
265     def __getitem__(self, position):
266         entity = self.resources[position]
267         entity.reload()
268         return entity
269
270     def __len__(self):
271         return self.subcount
272
273     def __repr__(self):
274         return "<SearchResult for {!r}>".format(self.collection)
275
276
277 class Collection(object):
278     def __init__(self, api, href, name, description=None):
279         self._api = api
280         self._href = href
281         self._data = None
282         self._subcollections = None
283         self._subcollections_loaded = False
284         self.action = ActionContainer(self)
285         self.name = name
286         self.description = description
287
288     @property
289     def api(self):
290         return self._api
291
292     @property
293     def subcollections(self):
294         # it's enought to try to load the subcollections list once
295         if not self._subcollections_loaded:
296             self._subcollections_loaded = True
297             try:
298                 opts = self._api.options(self._href)
299                 self._subcollections = opts['subcollections']
300             except Exception:
301                 # OPTIONS are supported only in new versions of API and subcollections are
302                 # returned only in even newer versions. Many things can go wrong here,
303                 # hence this broad except.
304                 self._subcollections = None
305                 self._api.logger.warning(
306                     "[RESTAPI] failed to get subcollections list for %s", self._href)
307         return self._subcollections
308
309     def reload(self, expand=False, attributes=None):
310         if expand is True:
311             kwargs = {"expand": "resources"}
312         elif expand:
313             kwargs = {"expand": expand}
314         else:
315             kwargs = {}
316         if attributes is not None:
317             if isinstance(attributes, six.string_types):
318                 attributes = [attributes]
319             kwargs.update(attributes=",".join(attributes))
320         self._data = self._api.get(self._href, **kwargs)
321         self._resources = self._data["resources"]
322         self._count = self._data.get("count", 0)
323         self._subcount = self._data.get("subcount", 0)
324         self._actions = self._data.pop("actions", [])
325         if self._data["name"] != self.name:
326             raise ValueError("Name mishap!")
327
328     def reload_if_needed(self):
329         if self._data is None:
330             self.reload()
331
332     def query_string(self, **params):
333         """Specify query string to use with the collection.
334
335     Returns: :py:class:`SearchResult`
336     """
337         return SearchResult(self, self._api.get(self._href, **params))
338
339     def raw_filter(self, filters):
340         """Sends all filters to the API.
341
342     No fancy, just a wrapper. Any advanced functionality shall be implemented as another method.
343
344     Args:
```

```
345         filters: List of filters (strings)
346
347       Returns: :py:class:`SearchResult`
348       """
349           return SearchResult(self, self._api.get(self._href, **{"filter[]": filters}))
350
351       def filter(self, q):
352           """Access the ``filter[]`` functionality of ManageIQ.
353
354       Args:
355         q: An instance of :py:class:`filters.Q`
356
357       Returns: :py:class:`SearchResult`
358       """
359           return self.raw_filter(q.as_filters)
360
361       def find_by(self, **params):
362           """Searches in ManageIQ using the ``filter[]`` get parameter.
363
364       This method only supports logical AND so all key/value pairs are considered as equality
365       comparision and all are logically anded.
366       """
367           return self.filter(Q.from_dict(params))
368
369       def get(self, **params):
370           try:
371               return self.find_by(**params)[0]
372           except IndexError:
373               raise ValueError("No such '{}' matching query {!r}!".format(self.name, params))
374
375       def options(self, **params):
376           return self._api.options(self._href, **params)
377
378       @property
379       def count(self):
380           self.reload_if_needed()
381           return self._count
382
383       @property
384       def subcount(self):
385           self.reload_if_needed()
386           return self._subcount
387
388       @property
389       def all(self):
390           self.reload(expand=True)
391           return [Entity(self, r) for r in self._resources]
392
393       def all_include_attributes(self, attributes):
394           """Returns all entities present in the collection with ``attributes`` included."""
395           self.reload(expand=True, attributes=attributes)
396           entities = [Entity(self, r, attributes=attributes) for r in self._resources]
397           self.reload()
398           return entities
399
400       def __repr__(self):
401           return "<Collection {!r} ({!r})>".format(self.name, self.description)
402
403       def __call__(self, entity_id, attributes=None):
404           return self._api.get_entity(self, entity_id, attributes=attributes)
405
406       def __iter__(self):
407           self.reload(expand=True)
408           for resource in self._resources:
409               yield Entity(self, resource)
410
411       def __getitem__(self, position):
412           self.reload_if_needed()
413           entity = Entity(self, self._resources[position])
414           entity.reload()
415           return entity
416
417       def __len__(self):
418           return self.subcount
419
420
421   class Entity(object):
422       # TODO: Extend these fields
423       TIME_FIELDS = {
424           "updated_on", "created_on", "last_scan_attempt_on", "state_changed_on", "lastlogon",
425           "updated_at", "created_at", "last_scan_on", "last_sync_on", "last_refresh_date",
426           "retires_on"}
427       COLLECTION_MAPPING = dict(
428           ems_id="providers",
429           storage_id="data_stores",
430           zone_id="zones",
```

```
431             host_id="hosts",
432             current_group_id="groups",
433             miq_user_role_id="roles",
434             evm_owner_id="users",
435             task_id="tasks",
436         )
437         EXTENDED_COLLECTIONS = dict(
438             roles={"features"},
439         )
440
441         def __init__(self, collection, data, incomplete=False, attributes=None):
442             self.collection = collection
443             self.action = ActionContainer(self)
444             self._data = data
445             self._incomplete = incomplete
446             self._attributes = attributes
447             self._href = None
448             self._load_data()
449
450         def _load_data(self):
451             if "id" in self._data:  # We have complete data
452                 self.reload(get=False)
453             elif "href" in self._data:  # We have only href
454                 self._href = self._data["href"]
455                 # self._data = None
456             else:  # Malformed
457                 raise ValueError("Malformed data: {!r}".format(self._data))
458
459         def reload(self, expand=None, get=True, attributes=None):
460             kwargs = {}
461             if expand:
462                 if isinstance(expand, (list, tuple)):
463                     expand = ",".join(map(str, expand))
464                 kwargs.update(expand=expand)
465             if attributes is None:
466                 attributes = self._attributes
467             if attributes:
468                 if isinstance(attributes, six.string_types):
469                     attributes = [attributes]
470                 kwargs.update(attributes=",".join(attributes))
471             if "href" in self._data:
472                 self._href = self._data["href"]
473             if get and self._href:
474                 new = self.collection._api.get(self._href, **kwargs)
475                 if self._data is None:
476                     self._data = new
477                 else:
478                     self._data.update(new)
479             self._actions = self._data.pop("actions", [])
480             for key, value in self._data.items():
481                 if value is None:
482                     continue
483                 if key in self.TIME_FIELDS:
484                     setattr(self, key, iso8601.parse_date(value))
485                 elif key in self.COLLECTION_MAPPING.keys():
486                     setattr(
487                         self,
488                         re.sub(r"_id$", "", key),
489                         self.collection._api.get_entity(self.COLLECTION_MAPPING[key], value)
490                     )
491                     setattr(self, key, value)
492                 elif (isinstance(value, dict) and self._href and
493                       "count" in value and "resources" in value):
494                     href = self._href
495                     if not href.endswith("/"):
496                         href += "/"
497                     subcol = Collection(self.collection._api, href + key, key)
498                     setattr(self, key, subcol)
499                 elif (isinstance(value, list) and self._href and
500                       key in self.EXTENDED_COLLECTIONS.get(self.collection.name, set([]))):
501                     href = self._href
502                     if not href.endswith("/"):
503                         href += "/"
504                     subcol = Collection(self.collection._api, href + key, key)
505                     setattr(self, key, subcol)
506                 else:
507                     setattr(self, key, value)
508
509         @property
510         def exists(self):
511             try:
512                 self.reload()
513             except APIException:
514                 return False
515             else:
516                 return True
```

```
517
518     @property
519     def subcollections(self):
520         return self.collection.subcollections
521
522     def wait_for_existence(self, existence, **kwargs):
523         return wait_for(
524             lambda: self.exists, fail_condition=not existence, **kwargs)
525
526     def wait_exists(self, **kwargs):
527         return self.wait_for_existence(True, **kwargs)
528
529     def wait_not_exists(self, **kwargs):
530         return self.wait_for_existence(False, **kwargs)
531
532     def reload_if_needed(self):
533         if self._data is None or self._incomplete or not hasattr(self, "_actions"):
534             self.reload()
535             self._incomplete = False
536
537     def __getattr__(self, attr):
538         self.reload()
539         if attr in self.__dict__:
540             # It got loaded
541             return self.__dict__[attr]
542         if self.subcollections is not None:
543             if attr not in self.subcollections:
544                 raise AttributeError("No such attribute/subcollection {}".format(attr))
545         if not self._href:
546             raise AttributeError("Can't get URL of attribute/subcollection {}".format(attr))
547         # Try to get subcollection
548         href = self._href
549         if not href.endswith("/"):
550             href += "/"
551         subcol = Collection(self.collection._api, href + attr, attr)
552         try:
553             subcol.reload()
554         except APIException:
555             raise AttributeError("No such attribute/subcollection {}".format(attr))
556         else:
557             return subcol
558
559     def __getitem__(self, item):
560         # Backward compatibility
561         return getattr(self, item)
562
563     def __repr__(self):
564         return "<Entity {!r}>".format(self._href if self._href else self._data["id"])
565
566     def _ref_repr(self):
567         return {"href": self._href} if self._href else {"id": self._data["id"]}
568
569
570 class ActionContainer(object):
571     def __init__(self, obj):
572         self._obj = obj
573
574     def reload(self):
575         self._obj.reload_if_needed()
576         reloaded_actions = []
577         for action in self._obj._actions:
578
579             def _add_method(method=None):
580                 """Adds HTTP method to Action, e.g. ``.delete.POST()``."""
581                 method = method or action["method"]
582                 new_name = method.upper()
583                 base_action_obj = self.__dict__[action["name"]]
584                 if not hasattr(base_action_obj, new_name):
585                     action_obj = base_action_obj if method == base_action_obj._method else Action(
586                         self, action["name"], method, action["href"])
587                     setattr(base_action_obj, new_name, action_obj)
588
589             # There can be multiple actions with the same name and different HTTP methods
590             # (e.g. action "delete" with HTTP methods POST or DELETE).
591             # Create action ``.name()`` with default (first) method.
592             # For each method, create action ``.name.METHOD()``.
593             # E.g. default action ``.delete()`` with method POST and actions
594             # ``.delete.POST()`` and ``.delete.DELETE()``.
595             if action["name"] not in reloaded_actions:
596                 reloaded_actions.append(action["name"])
597                 action_obj = Action(self, action["name"], action["method"], action["href"])
598                 setattr(self, action["name"], action_obj)
599             _add_method()
600
601             # Edit actions on entities can be performed using PATCH and PUT methods as well.
602             # These methods are not listed in "actions", therefore adding
```

```
603             # them explicitly - see https://bugzilla.redhat.com/show_bug.cgi?id=1491336
604             if action["name"] == "edit" and isinstance(self._obj, Entity):
605                 for edit_method in ("patch", "put"):
606                     _add_method(method=edit_method)
607
608     def execute_action(self, action_name, *args, **kwargs):
609         # To circumvent bad method names, like 'import', you can use this one directly
610         action = getattr(self, action_name)
611         action_method = kwargs.pop('action_method', None)
612         if action_method:
613             action = getattr(action, action_method)
614         return action(*args, **kwargs)
615
616     @property
617     def all(self):
618         self.reload()
619         return [a["name"] for a in self._obj._actions]
620
621     @property
622     def collection(self):
623         if isinstance(self._obj, Collection):
624             return self._obj
625         elif isinstance(self._obj, Entity):
626             return self._obj.collection
627         else:
628             raise ValueError("ActionContainer assigned to wrong object!")
629
630     def __getattr__(self, attr):
631         self.reload()
632         if attr not in self.__dict__:
633             raise AttributeError("No such action {}".format(attr))
634         return self.__dict__[attr]
635
636     def __contains__(self, action):
637         return action in self.all
638
639
640 class Action(object):
641     def __init__(self, container, name, method, href):
642         self._container = container
643         self._method = method
644         self._href = href
645         self._name = name
646
647     @property
648     def collection(self):
649         return self._container.collection
650
651     @property
652     def api(self):
653         return self.collection.api
654
655     def __call__(self, *args, **kwargs):
656         # TODO: for backwards compatibility - can be removed when no longer used
657         # possibility to override HTTP method that will be used with the action
658         # (e.g. force_method='delete')
659         force_method = kwargs.pop('force_method', None)
660         if force_method:
661             return getattr(self, force_method.upper())(*args, **kwargs)
662
663         if self._method == "post":
664             resources = []
665             # We got resources to post
666             for res in args:
667                 if isinstance(res, Entity):
668                     resources.append(res._ref_repr())
669                 else:
670                     resources.append(res)
671             query_dict = {"action": self._name}
672             if resources:
673                 query_dict["resources"] = []
674                 for resource in resources:
675                     new_res = dict(resource)
676                     if kwargs:
677                         new_res.update(kwargs)
678                     query_dict["resources"].append(new_res)
679             else:
680                 if kwargs:
681                     query_dict["resource"] = kwargs
682
683             result = self.api.post(self._href, **query_dict)
684         elif self._method == "patch":
685             result = self.api.patch(self._href, *args)
686         elif self._method in ("delete", "put"):
687             result = getattr(self.api, self._method)(self._href, **kwargs)
688         else:
```

```
689                 raise NotImplementedError
690             if result is None:
691                 return None
692             # Make sure that HTTP response from action is not overriden during result processing
693             action_response = self.api.response
694             try:
695                 if "results" in result:
696                     outcome = [self._process_result(r) for r in result["results"]]
697                 else:
698                     outcome = self._process_result(result)
699             finally:
700                 self.api.response = action_response
701             return outcome
702
703         def _get_entity_from_href(self, result):
704             """Returns entity in correct collection.
705
706         If the "href" value in result doesn't match the current collection,
707         try to find the collection that the "href" refers to.
708         """
709             href_result = result['href']
710
711             if self.collection._href.startswith(href_result):
712                 return Entity(self.collection, result, incomplete=True)
713
714             href_match = re.match(r"(https?://.+/api[^?]*)/([a-z_-]+)", href_result)
715             if not href_match:
716                 raise ValueError("Malformed href: {}".format(href_result))
717             collection_name = href_match.group(2)
718             entry_point = href_match.group(1)
719             new_collection = Collection(
720                 self.collection.api,
721                 "{}/{}".format(entry_point, collection_name),
722                 collection_name
723             )
724             return Entity(new_collection, result, incomplete=True)
725
726         def _process_result(self, result):
727             if result is None:
728                 return None
729             elif "href" in result:
730                 return self._get_entity_from_href(result)
731             elif "id" in result:
732                 d = copy(result)
733                 d["href"] = "{}/{}".format(self.collection._href, result["id"])
734                 return Entity(self.collection, d, incomplete=True)
735             elif "task_href" in result:
736                 collection = self.api.collections.tasks
737                 # reuse task_href and task_id, no other data is relevant
738                 d = {"href": result.get("task_href"), "id": result.get("task_id")}
739                 return Entity(collection, d, incomplete=True)
740             elif "message" in result:
741                 return result
742             else:
743                 raise NotImplementedError
744
745         def __repr__(self):
746             return "<Action {} {}#{}>".format(self._method, self._container._obj._href, self._name)
```