connect-openapi-client

Release 21.0.4.dev0+gfb90689.d20201109

CloudBlue

Nov 09, 2020

CONTENTS:

Welcome to *Connect Python OpenAPI Client* a simple, concise, powerful and REPL-friendly ReST API client.

It has been designed following the fluent interface design pattern.

Due to its REPL-friendly nature, using the CloudBlue Connect OpenAPI specifications it allows developers to learn and play with the CloudBlue Connect API using a python REPL like jupyter or ipython.

CHAPTER

ONE

USER GUIDE

1.1 Getting started

1.1.1 Requirements

connect-openapi-client runs on python 3.6 or later and has the following dependencies:

- connect-markdown-renderer 1.*
- requests 2.*
- pyyaml 5.*

1.1.2 Install

connect-openapi-client is a small python package that can be installed from the pypi.org repository.

```
$ pip install connect-openapi-client
```

1.2 First steps with Connect Python OpenAPI Client

1.2.1 Create a client instance

To use *connect-openapi-client* first of all you have to create an instance of the `ConnectClient` object:

```
from cnct import ConnectClient

client = ConnectClient('ApiKey SU-000-000-000:xxxxxxxxxxxxxxxx')
```

1.2.2 Access to namespaces and collections

The `ConnectClient` instance allows access to collections of resources using the *`collection()`* method of the client:

```
products = client.collection('products')
```

The previous call to the *`collection()`* method returns a *`Collection`* object that allows working with the resources that contain.

Some collections of the CloudBlue Connect ReST API are grouped within a namespace.

To access a namespace the client exposes the *`ns()`* method:

```
subscriptions = client.ns('subscriptions')
```

Since *Connect Python OpenAPI Client* has been designed following the fluent interface design pattern, you can chain methods:

```
assets = client.ns('subscriptions').collection('assets')
```

By default, the `ConnectClient` object, when instantiated, downloads and parse the CloudBlue Connect OpenAPI specifications.

This allows you to write the previous expression in a more concise way:

```
assets = client('subscriptions').assets
```

Note: For collections that use a dash in their names, it is yet possible to use the concise form by replacing the dash character with an underscore.

1.3 Working with resources

1.3.1 Create a new resource

To create a new resource inside a collection you can invoke the *`create()`* method on the corresponding *`Collection`* instance:

```
payload = {
    'name': 'My Awesome Product',
    'category': {
        'id': 'CAT-00000',
    },
}

new_product = c.products.create(payload=payload)
```

This returns the newly created object json-decoded.

1.3.2 Access to a resource

To access a resource within a collection you can use the *resource()* method on the corresponding *Collection* instance:

```
product = client.products.resource('PRD-000-000-000')
```

The indexing operator allows to write the previous expression the following way:

```
product = client.products['PRD-000-000-000']
```

The previous expression returns a *Resource* object.

Caution: The `Resource` object does not make any HTTP calls to retrieve the resource identified by the index, to avoid unnecessary traffic if what you want is to update it, delete it, perform an action on it or access a nested collection of resources. This means that, if the resource does not exist, any operation on it or on its nested collection will fail.

1.3.3 Retrieve a resource

To retrieve a resource from within a collection you have to invoke the *get()* method of the *Resource* object as shown below:

```
product = client.products['PRD-000-000-000'].get()
```

This call returns the json-decoded object or raise an exception if it does not exist.

1.3.4 Update a resource

To update a resource of the collection using its primary identifier, you can invoke the *update()* method as shown below:

```
payload = {
    'short_description': 'This is the short description',
    'detailed_description': 'This is the detailed description',
}

product = client.products['PRD-000-000-000'].update(payload=payload)
```

1.3.5 Delete a resource

To delete a resource the *delete()* method is exposed:

```
client.products['PRD-000-000-000'].delete()
```

1.3.6 Access to an action

To access an action that can be performed on a resource you can use the *action()* method of the *Resource* object or directly using the call operator on the *Resource* class passing the name of the action:

```
endsale_action = client.products['PRD-000-000-000']('endsale')
```

This returns an `Action` object.

1.3.7 Execute an action on a resource

Depending on its nature, an action can be exposed using the HTTP method that best gives the sense of the action to perform. The `Action` object exposes the `get()`, `post()`, `put()`, and `delete()` methods.

For example, supose you want to execute the **endsale** action:

```
payload = {
    'replacement': {
        'id': 'PRD-111-111-111'
    },
    'end_of_sale_notes': 'stopped manufacturing',
}

result = client.products['PRD-000-000-000']('endsale').post(payload=payload)
```

1.3.8 Access nested collections

If you want to access a nested collection, you can do that both using the *collection()* method or the name of the nested collection on the *Resource* object:

```
product_item = client.products['PRD-000-000-000'].items
```

As for root collections, you can use the `create()` method to create new resources within the nested collection or you can use the indexing operator to access a resource of the nested collection by ID.

1.4 Querying collections

You can perform queries on a collection to retrieve a set of resources that match the filters specified in the query.

The Connect ReST API use the Resource Query Language or RQL, to perform queries on a collection.

Note: This guide assumes you are somewhat familiar with RQL. If not, take a look at the RQL video tutorial here.

The `ResourceSet` object helps both to express RQL queries and to manipulate the resulting set of resources.

1.4.1 Create a `ResourceSet` object

A *cnct.client.models.ResourceSet* object can be created through the corresponding o:class:*cnct.client.models.Collection* object using the *cnct.client.models.Collection.all()* method to access all the resources of the collection:

```
products = client.products.all()
```

Or applying filter using the *cnct.client.models.Collection.filter()* method:

```
products = client.products.filter(status='published')
```

The `ResourceSet` will not be evaluated until you need the resources data, i.e. it does not make any HTTP call until needed, to help express more complex queries using method chaining like in the following example:

```
products = client.products.filter(status='published').order_by('-created')
```

1.4.2 Count results

To get the total number of resources represented by a `ResourceSet` you can use the `cnct.client.models.Collection.count()` method.

```
no_of_published = client.products.filter(status='published').count()
```

or

```
no_of_published = client.products.all().count()
```

1.4.3 First result

To get the first resource represented by a `ResourceSet` you can use the `cnct.client.models.Collection.first()` method.

```
first = client.products.all().first()
```

or

```
first = client.products.filter(status='published').first()
```

1.4.4 Filtering resources

The *cnct.client.models.ResourceSet* object offers three way to define your RQL query filters:

Using raw RQL filter expressions

You can express your filters using raw RQL expressions like in this example:

```
products = client.products.filter('ilike(name,*awesome*)', 'in(status,(draft,
↪published))')
```

Arguments will be joined using the `and` logical operator.

Using kwargs and the __ (double underscore) notation

You can use the __ notation at the end of the name of the keyword argument to specify which RQL operator to apply:

```
products = client.products.filter(name__ilike='*awesome*', status__in=('draft',
↪'published'))
```

The lookups expressed through keyword arguments are `and`-ed togheter.

Chaning the filter method combine filters using `and`. Equivalent to the previous expression is to write:

```
products = client.products.filter(name__ilike='*awesome*').filter(status__in=('draft',
↪ 'published'))
```

The __ notation allow also to specify nested fields for lookups like:

```
products = client.products.filter(product__category__name__ilike='"*saas services*"')
```

Using the `R` object

The *`R`* object allows to create complex RQL filter expression.

The *`R`* constructor allows to specify lookups as keyword arguments the same way you do with the *`filter()`* method.

But it allows also to specify nested fields using the `.` notation:

```
flt = R().product.category.name.ilike('"*saas services*"')

products = client.products.filter(flt)
```

So an expression like:

```
flt = R().product.category.name.ilike('"*saas services*"')

products = client.products.filter(flt, status__in=('draft', 'published'))
```

will result in the following RQL query:

```
and(ilike(product.category.name,"*saas services*"),in(status,(draft,published)))
```

The `R` object also allows to join filter expressions using logical `and` and `or` and `not` using the `&`, `|` and and `~` bitwise operators:

```
query = (
    R(status='published') | R().category.name.ilike('*awesome*')
) & ~R(description__empty=True)
```

1.5 Other RQL operators

1.5.1 Searching

For endpoints that supports the RQL search operator you can specify your search term has shown below:

1.5.2 Ordering

To apply ordering you can specify the fields that have to be used to order the results:

```
ordered = rs.order_by('+field1', '-field2')
```

Any subsequent calls append other fields to the previous one.

So the previous statement can also be expressed with chaining:

```
ordered = rs.order_by('+field1').order_by('-field2')
```

1.5.3 Apply RQL `select`

For collections that supports the `select` RQL operator you can specify the object to be selected/unselected the following way:

```
with_select = rs.select('+object1', '-object2')
```

Any subsequent calls append other select expression to the previous.

So the previous statement can also be expressed with chaining:

```
with_select = rs.select('+object1').select('-object2')
```

CHAPTER

TWO

API REFERENCE

2.1 Client

class cnct.client.fluent.**ConnectClient**(*api_key*, *endpoint=None*, *use_specs=True*, *specs_location=None*, *validate_using_specs=True*, *default_headers=None*, *default_limit=100*)

Connect ReST API client.

__getattr__(*name*)

Returns a collection object called `name`.

Parameters **name** (*str*) – The name of the collection to retrieve.

Returns a collection called `name`.

Return type *Collection*

__call__(*name*)

Call self as a function.

ns(*name*)

Returns the namespace called `name`.

Parameters **name** (*str*) – The name of the namespace to access.

Returns The namespace called `name`.

Return type *NS*

collection(*name*)

Returns the collection called `name`.

Parameters **name** (*str*) – The name of the collection to access.

Returns The collection called `name`.

Return type *Collection*

2.2 Models

class cnct.client.models.**NS**(*client*, *path*)
A namespace is a group of related collections.

__getattr__(*name*)
Returns a collection object by its name.

Parameters **name** (*str*) – the name of the Collection object.

Returns The Collection named name.

Return type *Collection*

collection(*name*)
Returns the collection called name.

Parameters **name** (*str*) – The name of the collection.

Raises

- **TypeError** – if the name is not a string.
- **ValueError** – if the name is blank.
- **NotFoundError** – if the name does not exist.

Returns The collection called name.

Return type *Collection*

help()
Output the namespace documentation to the console.

Returns self

Return type *NS*

class cnct.client.models.**Collection**(*client*, *path*)
A collection is a group of operations on a resource.

__getitem__(*resource_id*)
Return a Resource object representing the resource identified by resource_id.

Parameters **resource_id** (*str, int*) – The identifier of the resource

Returns the Resource instance identified by resource_id.

Return type *Resource*

all()
Return a ResourceSet instance.

Returns a ResourceSet instance.

Return type *ResourceSet*

filter(**args*, ***kwargs*)
Returns a ResourceSet object. The returned ResourceSet object will be filtered based on the arguments and keyword arguments.

Arguments can be RQL filter expressions as strings or R objects.

Ex.

```
rs = collection.filter('eq(field,value)', 'eq(another.field,value2)')
rs = collection.filter(R().field.eq('value'), R().another.field.eq('value2'))
```

All the arguments will be combined with logical `and`.

Filters can be also specified as keyword argument using the ___ (double underscore) notation.

Ex.

```
rs = collection.filter(
    field=value,
    another__field=value,
    field2__in=('a', 'b'),
    field3__null=True,
)
```

Also keyword arguments will be combined with logical `and`.

Raises **TypeError** – If arguments are neither strings nor R objects.

Returns A ResourceSet with the filters applied.

Return type *ResourceSet*

create(*payload=None*, ***kwargs*)

Create a new resource within this collection.

Parameters **payload** (*dict, optional*) – JSON payload of the resource to create, defaults to None.

Returns The newly created resource.

Return type dict

resource(*resource_id*)

Returns an Resource object.

Parameters **resource_id** (*str, int*) – The resource identifier.

Returns The Resource identified by `resource_id`.

Return type *Resource*

help()

Output the collection documentation to the console.

Returns self

Return type *Collection*

class cnct.client.models.**Resource**(*client*, *path*)

Represent a generic resource.

__getattr__(*name*)

Returns a nested Collection object called `name`.

Parameters **name** (*str*) – The name of the Collection to retrieve.

Returns a Collection called `name`.

Return type *Collection*

__call__(*name*)

Call self as a function.

collection(*name*)
Returns the collection called `name`.

Parameters **name** (*str*) – The name of the collection.

Raises

- **TypeError** – if the `name` is not a string.
- **ValueError** – if the `name` is blank.
- **NotFoundError** – if the `name` does not exist.

Returns The collection called `name`.

Return type *Collection*

action(*name*)
Returns the action called `name`.

Parameters **name** (*str*) – The name of the action.

Raises

- **TypeError** – if the `name` is not a string.
- **ValueError** – if the `name` is blank.
- **NotFoundError** – if the `name` does not exist.

Returns The action called `name`.

Return type Action

get(***kwargs*)
Execute a http GET to retrieve this resource. The http GET can be customized passing kwargs that will be forwarded to the underlying GET of the `requests` library.

Returns The resource data.

Return type dict

update(*payload=None*, ***kwargs*)
Execute a http PUT to update this resource. The http PUT can be customized passing kwargs that will be forwarded to the underlying PUT of the `requests` library.

Parameters **payload** (*dict, optional*) – the JSON payload of the update request, defaults to None

Returns The updated resource.

Return type dict

delete(***kwargs*)
Execute a http DELETE to delete this resource. The http DELETE can be customized passing kwargs that will be forwarded to the underlying DELETE of the `requests` library.

values(**fields*)
Returns a flat dictionary containing only the fields passed as arguments. Nested field can be specified using dot notation.

Ex.

```
values = resource.values('field', 'nested.field')
```

Returns A dictionary containing field,value pairs.

Return type dict

help()
Output the resource documentation to the console.

Returns self

Return type *Resource*

class cnct.client.models.**ResourceSet**(*client*, *path*, *query=None*)
Represent a set of resources.

__iter__()
Returns an iterator to iterate over the set of resources.

Returns A resources iterator.

Return type *ResourceSet*

__bool__()
Return True if the ResourceSet contains at least a resource otherwise return False.

Returns True if contains a resource otherwise False.

Return type bool

__getitem__(*key*)
If called with and integer index, returns the item at index `key`.

If key is a slice, set the pagination limit and offset accordingly.

Parameters **key** (*int, slice*) – index or slice.

Raises **TypeError** – If `key` is neither an integer nor a slice.

Returns The resource at index `key` or self if `key` is a slice.

Return type dict, ResultSet

configure(***kwargs*)
Set the keyword arguments that needs to be forwarded to the underlying `requests` call.

Returns This ResourceSet object.

Return type *ResourceSet*

limit(*limit*)
Set the number of results to be fetched from the remote endpoint at once.

Parameters **limit** (*int*) – maximum number of results to fetch in a batch.

Raises

- **TypeError** – if *limit* is not an integer.
- **ValueError** – if *limit* is not positive non-zero.

Returns A copy of this ResourceSet class with the new limit.

Return type *ResourceSet*

order_by(**fields*)
Add fields for ordering.

Returns This ResourceSet object.

Return type *ResourceSet*

select(**fields*)
Apply the RQL `select` operator to this ResourceSet object.

Returns This ResourceSet object.

Return type *ResourceSet*

filter(**args*, ***kwargs*)
Applies filters to this ResourceSet object.

Arguments can be RQL filter expressions as strings or R objects.

Ex.

```
rs = rs.filter('eq(field,value)', 'eq(another.field,value2)')
rs = rs.filter(R().field.eq('value'), R().another.field.eq('value2'))
```

All the arguments will be combined with logical `and`.

Filters can be also specified as keyword argument using the ___ (double underscore) notation.

Ex.

```
rs = rs.filter(
    field=value,
    another__field=value,
    field2__in=('a', 'b'),
    field3__null=True,
)
```

Also keyword arguments will be combined with logical `and`.

Raises **TypeError** – If arguments are neither strings nor R objects.

Returns This ResourceSet object.

Return type *ResourceSet*

count()
Returns the total number of resources within this ResourceSet object.

Returns The total number of resources present.

Return type int

first()
Returns the first resource that belongs to this ResourceSet object or None if the ResourceSet doesn't contains resources.

Returns The first resource.

Return type dict, None

all()
Returns a copy of the current ResourceSet.

Returns A copy of this ResourceSet.

Return type *ResourceSet*

search(*term*)
Create a copy of the current ResourceSet with the search set to *term*.

Parameters **term** (*str*) – The term to search for.

Returns A copy of the current ResourceSet.

Return type *ResourceSet*

values_list(**fields*)
Returns a flat dictionary containing only the fields passed as arguments for each resource that belongs to this ResourceSet.

Nested field can be specified using dot notation.

Ex.

```
values = rs.values_list('field', 'nested.field')
```

Returns A list of dictionaries containing field,value pairs.

Return type list

help()
Output the ResourceSet documentation to the console.

Returns self

Return type *ResourceSet*

2.3 RQL utility

cnct.rql.base.**R**
alias of *cnct.rql.base.RQLQuery*

class cnct.rql.base.**RQLQuery**(**, *_op='expr'*, *_children=None*, *_negated=False*, *_expr=None*, ***kwargs*)
Helper class to construct complex RQL queries.

__len__()
Returns the length of this R object. It will be 1 if represent a single RQL expression or the number of expressions this logical operator (and, or) applies to.

Returns The length of this R object.

Return type int

__bool__()
Returns False if it is an empty R object otherwise True.

Returns False if it is an empty R object otherwise True.

Return type bool

__eq__(*other*)
Returns True if self == other.

Parameters **other** (*R*) – Another R object.

Returns True if the `other` object is equal to self, False otherwise.

Return type bool

__and__(*other*)
Combine this R object with `other` using a logical `and`.

Parameters **other** (*R*) – Another R object.

Returns The R object representing a logical `and` between this and `other`.

Return type R

__or__(*other*)
Combine this R object with `other` using a logical `or`.

Parameters **other** (*R*) – Another R object.

Returns The R object representing a logical `or` between this and `other`.

Return type R

__invert__()
Apply the RQL `not` operator to this R object.

Returns The R object representing this R object negated.

Return type R

n(*name*)
Set the current field for this R object.

Parameters **name** (*str*) – name of the field

Raises **AttributeError** – if this R object has already been evaluated.

Returns This R object.

Return type R

2.4 Exceptions

class cnct.client.exceptions.**ClientError**(*message=None*, *status_code=None*, *error_code=None*, *errors=None*)

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

A

C

D

F

G

H

L

N

O

R

S

U

V