carto-python Documentation

Release 1.6.0

CARTO

Jul 05, 2019

Contents

This section contains documentation on how to use the different *carto-python* APIs.

carto-python is a full, backwards incompatible rewrite of the deprecated *cartodb-python* SDK. Since the initial rewrite, *carto-python* has been loaded with a lot of new features, not present in old *cartodb-python*.

carto-python is a Python library to consume the CARTO APIs. You can integrate *carto-python* into your Python projects to:

- Import data from files, URLs or external databases to your user account or organization
- Execute SQL queries and get the results
- Run batch SQL jobs
- Create and instantiate named and anonymous maps
- Create, update, get, delete and list datasets, users, maps...
- etc.

You may find specially useful the *Examples* section for actual use cases of the CARTO Python library.

Please, refer to the *carto package API documentation* or the source code for further details about modules, methods and parameters.

Note: Code snippets provided in this developer guide are not intended to be executed since they may not contain API keys or USERNAME values needed to actually execute them. Take them as a guide on how to work with the modules and classes

---

CHAPTER 1

Quickstart

In order to use the CARTO Python client first you have to follow the *Installation* guide and then write a Python script that makes use of it.

As an example, next code snippet makes a SQL query to a dataset

```
from carto.auth import APIKeyAuthClient
from carto.exceptions import CartoException
from carto.sql import SQLClient

USERNAME="type here your username"
USR_BASE_URL = "https://{user}.carto.com/".format(user=USERNAME)
auth_client = APIKeyAuthClient(api_key="myapikey", base_url=USR_BASE_URL)

sql = SQLClient(auth_client)

try:
    data = sql.send('select * from mytable')
except CartoException as e:
    print("some error ocurred", e)

print data['rows']
```

CHAPTER 2

General Concepts

The CARTO Python module implements these public CARTO APIs:

- SQL API.
- Import API.
- Maps API.

As well as other non-public APIs. Non-public APIs may change in the future and will throw a warnings.warn message when used.

Please be aware if you plan to run them on a production environment.

Refer to the *carto package API documentation* for a list of non-public APIs implemented.

2.1 pyrestcli

The CARTO Python client relies on a Python REST client called pyrestcli

pyrestcli allows you to define data models, with a syntax that is derived from Django's model framework, that you can use directly against REST APIs

2.2 Resources and Managers

The CARTO Python client is built upon two main concepts: *Resource* and *Manager*

A *Resource* represent your model, according to the schema of the data available on the server for a given API. A *Manager* is a utility class to create *Resource* instances.

Each API implemented by the CARTO Python client provides a *Manager* and a *Resource*.

With a *Manager* instance you can:

- Get a resource given its id

```
resource = manager.get(resource_id)
```

- Create a new resource

```
resource = manager.create({id: "resource_id", prop_a: "test"})
```

- Retrieve all the resources

```
resources = manager.all()
```

- Get a filtered list of resources (search_args: To be translated into ?arg1=value1&arg2=value2...)

```
resources = manager.filter(**search_args)
```

With a *Resource* instance you can:

- Save the resource instance (equivalent to update the resource)

```
resource.save()
```

- Delete the resource instance

```
resource.delete()
```

- Refresh the resource instance

```
resource.refresh()
```

The CARTO Python client's Managers and Resources extend both classes, so please refer to the *carto package API documentation* for additional methods available.

2.3 Types of resources

The CARTO Python client provides three different types of Resources with different features:

- *AsyncResource*: Used for API requests that are asynchronous, as the Batch SQL API.

AsyncResources work in this way. First you create the asynchronous job in the server:

```
async_resource.run(**import_args)
```

Second, you start a loop refreshing the *async_resource* and checking the state of the job created in the server (depending on the API requested, the 'state' value may change):

```
while async_resource.state in ("enqueued", "pending", "uploading",
                               "unpacking", "importing", "guessing"):
    async_resource.refresh()
```

Finally, you check the state to know the status of the job in the server:

```
status = async_resource.state
# do what it takes depending on the status
```

- *WarnAsyncResource*: This type of *Resource* is an *AsyncResource* of a non-public API, so it will throw *warnings* whenever you try to use it.

- *WarnResource*: This type of *Resource* is a regular *Resource* of a non-public API, so it will throw *warnings* whenever you try to use it.

The use of *WarnAsyncResource* and *WarnResource* is totally discouraged for production environments, since non-public APIs may change without prior advice.

2.4 Fields

A *Field* class represent an attribute of a *Resource* class.

The *Field* class is meant to be subclassed every time a new specific data type wants to be defined.

Fields are a very handy way to parse a JSON coming from the REST API and store real Python objects on the *Resource*

The list of available fields is:

- *Field*: This default *Field* simply stores the value in the instance as it comes, suitable for basic types such as integers, chars, etc.
- *BooleanField*: Convenient class to make explicit that an attribute will store booleans
- *IntegerField*: Convenient class to make explicit that an attribute will store integers
- *FloatField*: Convenient class to make explicit that an attribute will store floats
- *CharField*: Convenient class to make explicit that an attribute will store chars
- *DateTimeField*: *Field* to store *datetimes* in resources
- *DictField*: Convenient class to make explicit that an attribute will store a dictionary
- *ResourceField*: *Field* to store resources inside other resources

The CARTO Python client provides additional instances of *ResourceField*:

- *VisualizationField*
- *TableField*
- *UserField*
- *EntityField*
- *PermissionField*

2.5 Exceptions

- *CartoException*: Generic exception class of the CARTO Python client. Most of the exceptions are wrapped into it.
- *CartoRateLimitException*: it is raised when a request is rate limited by SQL or Maps APIs (429 Too Many Requests HTTP error). More info about CARTO rate limits. It extends *CartoException* class with the rate limits info, so that any client can manage when to retry a rate limited request.

Please refer to the CARTO developer center for more information about concrete error codes and exceptions.

CHAPTER 3

Installation

You can install the CARTO Python client by using Pip.

```
pip install carto
```

If you want to use the development version, you can install directly from Github:

```
pip install -e git+git://github.com/CartoDB/carto-python.git#egg=carto
```

CHAPTER 4

Authentication

Before making API calls, we need to define how those calls are going to be authenticated. Currently, we support two different authentication methods: unauthenticated and API key based.

Therefore, we first need to create an *authentication client* that will be used when instantiating the Python classes that deal with API requests.

For unauthenticated requests, we need to create a *NoAuthClient* object:

```
from carto.auth import NoAuthClient

USERNAME="type here your username"
USR_BASE_URL = "https://{user}.carto.com/".format(user=USERNAME)
auth_client = NoAuthClient(base_url=USR_BASE_URL)
```

For API key authenticated requests, we need to create an *APIKeyAuthClient* instance:

```
from carto.auth import APIKeyAuthClient

USERNAME="type here your username"
USR_BASE_URL = "https://{user}.carto.com/".format(user=USERNAME)
auth_client = APIKeyAuthClient(api_key="myapikey", base_url=USR_BASE_URL)
```

API key is mandatory for all API requests except for sending SQL queries to public datasets.

The *base_url* parameter must include the *user* and or the *organization* with a format similar to these ones:

```
BASE_URL = "https://{organization}.carto.com/user/{user}/". \
    format(organization=ORGANIZATION,
           user=USERNAME)
USR_BASE_URL = "https://{user}.carto.com/".format(user=USERNAME)
```

For a detailed description of the rest of parameters both constructors accept, please take a look at the *carto.auth module* documentation.

Finally, you can use a *NonVerifiedAPIKeyAuthClient* instance if you are running CARTO on your premises and don't have a valid SSL certificate:

```
from carto.auth import NonVerifiedAPIKeyAuthClient

USERNAME="type here your username"
YOUR_ON_PREM_DOMAIN="myonprem.com"
USR_BASE_URL = "https://{domain}/user/{user}".format(domain=YOUR_ON_PREM_DOMAIN,␣
↪user=USERNAME)
auth_client = NonVerifiedAPIKeyAuthClient(api_key="myapikey", base_url=USR_BASE_URL)
```

CHAPTER 5

Auth API

Auth API is the piece of the CARTO platform that enables a consistent, uniform way of accessing data, datasets and APIs.

All requests to CARTO's APIs (Maps, SQL, etc.) require you to authenticate with an API key. API keys identify your project and provide a powerful and flexible primitive for managing access to CARTO's resources like APIs and Datasets.

These API keys can be provisioned, revoked and regenerated through the Auth API.

To learn how to create API keys from your CARTO dashboard, full Auth API reference or usage guides, please check the CARTO's help center

5.1 Types of API keys

In CARTO, you can find 3 types of API keys:

- Default public: Only valid to access for public data. It cannot be removed.
- Master: A super-user API key, it can do anything on your user account. And since a great power carries a great responsibility, you should use it for testing and development only! Keep it secret and use it sparingly.
- Regular: Regular API keys are the most common type of API keys. They provide access to APIs and database tables (aka Datasets) in a granular and flexible manner. They can be removed and have to be created using the Master API key.

5.2 API key format

Every API key consists on four main parts:

- name: You will choose it when creating the API key and it will be used for indexing your API keys.
- type: As mentioned before, there are three type of API keys: *default*, *master* and *regular* providing different levels of access.

- token: It will be used for authenticating your requests.
- grants: Describes which APIs this API key provides access to and to which tables. It consists on an array of two JSON objects. This object's *type* attribute can be *apis*, *database* or *dataservices*:
  - *apis*: Describes which APIs does this API key provide access to through apis attribute
  - *database*: Describes to which tables and which privleges on each table this API key grants access to though tables attribute
  - *dataservices*: Describes to which data services this API key grants access to though services attribute:

See the full API key format reference in the CARTO help center for more info about allowed table permissions, *dataservices*, etc.

5.3 API keys in the context of the Python SDK

To be able to access the CARTO APIs you need an API key.

In the context of the Python SDK, each API client needs an *auth_client* which contains the user account credentials.

Let's see an example:

Let's imagine I have a *tornadoes* dataset in my CARTO account and I want to get data about last year tornadoes.

The Python SDK provides an *SQLClient* which allows to run SQL queries. For this specific case we could run a query of this type:

```
query = "SELECT * FROM tornadoes WHERE year = DATE_PART('year', NOW() - INTERVAL '1␣
↪year')"
```

To run this SQL query we need to do it via the *SQLClient*

```
from carto.sql import SQLClient
from carto.exceptions import CartoException

sql = SQLClient(auth_client)

try:
  query = "SELECT * FROM tornadoes WHERE year = DATE_PART('year', NOW() - INTERVAL '1␣
↪year')"
  data = sql.send(query)
except CartoException as e:
  print("some error ocurred", e)
```

And to use the *SQLClient* we need an *auth_client* instance which allows to authenticate with our user credentials via an API key. The full example would look as follows:

```
from carto.auth import APIKeyAuthClient
from carto.sql import SQLClient
from carto.exceptions import CartoException

API_KEY = "myapikey"
USERNAME="type here your username"
USR_BASE_URL = "https://{user}.carto.com/".format(user=USERNAME)
auth_client = APIKeyAuthClient(api_key=API_KEY, base_url=USR_BASE_URL)

sql = SQLClient(auth_client)
```

(continues on next page)

(continued from previous page)

```
try:
  query = "SELECT * FROM tornadoes WHERE year = DATE_PART('year', NOW() - INTERVAL '1␣
↪year')"
  data = sql.send(query)
except CartoException as e:
  print("some error ocurred", e)
```

And this is when the Auth API can be useful.

5.4 Creating a regular API key

Let's go back to our *tornadoes* example. We might want to create a Regular API key that only has access to the *tornadoes* dataset. This is how we'd do it:

```
from carto.auth import APIKeyAuthClient
from carto.api_keys import APIKeyManager

API_KEY = "mymasterapikey"
USERNAME="type here your username"
USR_BASE_URL = "https://{user}.carto.com/".format(user=USERNAME)
auth_client = APIKeyAuthClient(api_key=API_KEY, base_url=USR_BASE_URL)

api_key_mamager = APIKeyManager(auth_client)
tables = [{
            "schema": api_key_manager.client.username,
            "name": "tornadoes",
            "permissions": [
              "select"
            ]
          }]
api_key = api_key_manager.create(name="tornadoes api key", tables=tables)
print(api_key.token)

# Now we can use this API key `token` to get data from the `tornadoes` dataset
```

5.5 Regenerate token of an existing regular API key

This will regenerate the internal token of the API key instance in case it has been compromised. Regular and Master API keys tokens can be regenerated.

```
from carto.auth import APIKeyAuthClient
from carto.api_keys import APIKeyManager

API_KEY = "mymasterapikey"
USERNAME="type here your username"
USR_BASE_URL = "https://{user}.carto.com/".format(user=USERNAME)
auth_client = APIKeyAuthClient(api_key=API_KEY, base_url=USR_BASE_URL)

api_key_mamager = APIKeyManager(auth_client)
tornados_api_key = api_key_mamager.get("tornadoes api key")

tornados_api_key.regenerate_token()
```

5.6 Revoke access to your account to an API key

API keys cannot be edited, that means wherever you grant some privileges to an API key the only way to revoke those privileges is by deleting the API key.

```
from carto.auth import APIKeyAuthClient
from carto.api_keys import APIKeyManager

API_KEY = "mymasterapikey"
USERNAME="type here your username"
USR_BASE_URL = "https://{user}.carto.com/".format(user=USERNAME)
auth_client = APIKeyAuthClient(api_key=API_KEY, base_url=USR_BASE_URL)

api_key_mamager = APIKeyManager(auth_client)
tornados_api_key = api_key_mamager.get("tornadoes api key")

tornados_api_key.delete()
```

5.7 Get all my regular API keys

```
from carto.auth import APIKeyAuthClient
from carto.api_keys import APIKeyManager

API_KEY = "mymasterapikey"
USERNAME="type here your username"
USR_BASE_URL = "https://{user}.carto.com/".format(user=USERNAME)
auth_client = APIKeyAuthClient(api_key=API_KEY, base_url=USR_BASE_URL)

api_key_mamager = APIKeyManager(auth_client)
api_keys = api_key_manager.filter(type='regular')

# now you can do any operation on those api_keys
```

5.8 Grant access to Data services

Regular API keys can also be granted privileges to the Data Services API

```
from carto.auth import APIKeyAuthClient
from carto.api_keys import APIKeyManager

API_KEY = "mymasterapikey"
USERNAME="type here your username"
USR_BASE_URL = "https://{user}.carto.com/".format(user=USERNAME)
auth_client = APIKeyAuthClient(api_key=API_KEY, base_url=USR_BASE_URL)

api_key_mamager = APIKeyManager(auth_client)
dataservices = ["geocoding", "routing", "isolines", "observatory"]
api_key = api_key_manager.create(name="tornadoes api key", services=dataservices)
```

Once we have created the regular API key we can run queries against the Data Services API

```
from carto.sql import SQLClient
from carto.exceptions import CartoException

# Create a new auth_client with the token of the regular API key previously created
auth_client = APIKeyAuthClient(api_key=api_key.token, base_url=USR_BASE_URL)
sql = SQLClient(auth_client)

try:
  query = "SELECT cdb_geocode_admin0_polygon('USA')"
  data = sql.send(query)
except CartoException as e:
  print("some error ocurred", e)
```

CHAPTER 6

SQL API

Making requests to the *SQL API* is pretty straightforward:

```
from carto.sql import SQLClient
from carto.exceptions import CartoException

sql = SQLClient(auth_client)

try:
    data = sql.send('select * from mytable')
except CartoException as e:
    print("some error ocurred", e)

print data['rows']
```

6.1 POST and GET

The CARTO SQL API is setup to handle both GET and POST requests.

By default all requests are sent via *POST*, anyway you still can send requests via *GET*:

```
from carto.sql import SQLClient
from carto.exceptions import CartoException

sql = SQLClient(auth_client)

try:
   data = sql.send('select * from mytable', do_post=False)
except CartoException as e:
   print("some error ocurred", e)

print data['rows']
```

6.2 Response formats

The SQL API accepts many output formats that can be useful to export data, such as:

- CSV
- SHP
- SVG
- KML
- SpatiaLite
- GeoJSON

By default, requests are sent in *JSON* format, but you can specify a different format like this:

```
from carto.sql import SQLClient

sql = SQLClient(auth_client)

try:
    result = sql.send('select * from mytable', format='csv')
    # here you have a CSV, proceed to do what it takes with it
except CartoException as e:
    print("some error ocurred", e)
```

Please refer to the *carto package API documentation* to find out about the rest of the parameters accepted by the constructor and the *send* method.

6.3 Batch SQL requests

For long lasting SQL queries you can use the batch SQL API.

```
from carto.sql import BatchSQLClient

LIST_OF_SQL_QUERIES = []

batchSQLClient = BatchSQLClient(auth_client)
createJob = batchSQLClient.create(LIST_OF_SQL_QUERIES)

print(createJob['job_id'])
```

The *BatchSQLClient* is asynchronous, but it offers methods to check the status of a job, update it or cancel it:

```
# check the status of a job after it has been created and you have the job_id
readJob = batchSQLClient.read(job_id)

# update the query of a batch job
updateJob = batchSQLClient.update(job_id, NEW_QUERY)

# cancel a job given its job_id
cancelJob = batchSQLClient.cancel(job_id)
```

6.4 COPY queries

COPY queries allow you to use the PostgreSQL COPY command for efficient streaming of data to and from CARTO.

Here is a basic example of its usage:

```
from carto.sql import SQLClient
from carto.sql import CopySQLClient

sql_client = SQLClient(auth_client)
copy_client = CopySQLClient(auth_client)

# Create a destination table for the copy with the right schema
sql_client.send("""
    CREATE TABLE IF NOT EXISTS copy_example (
      the_geom geometry(Geometry,4326),
      name text,
      age integer
    )
    """)
sql_client.send("SELECT CDB_CartodbfyTable(current_schema, 'copy_example')")

# COPY FROM a csv file in the filesytem
from_query = 'COPY copy_example (the_geom, name, age) FROM stdin WITH (FORMAT csv,␣
↪HEADER true)'
result = copy_client.copyfrom_file_path(from_query, 'copy_from.csv')

# COPY TO a file in the filesystem
to_query = 'COPY copy_example TO stdout WITH (FORMAT csv, HEADER true)'
copy_client.copyto_file_path(to_query, 'export.csv')
```

Here's an equivalent, more pythonic example of the COPY FROM, using a `file` object:

```
with open('copy_from.csv', 'rb') as f:
    copy_client.copyfrom_file_object(from_query, f)
```

And here is a demonstration of how to generate and stream data directly (no need for a file at all):

```
def rows():
    # note the \n to delimit rows
    yield bytearray(u'the_geom,name,age\n', 'utf-8')
    for i in range(1,80):
        row = u'SRID=4326;POINT({lon} {lat}),{name},{age}\n'.format(
            lon = i,
            lat = i,
            name = 'fulano',
            age = 100 - i
        )
        yield bytearray(row, 'utf-8')
copy_client.copyfrom(from_query, rows())
```

For more examples on how to use the SQL API, please refer to the **examples** folder or the *carto package API documentation*.

CHAPTER 7

Import API

You can import local or remote datasets into CARTO via the *Import API* like this:

```
from carto.datasets import DatasetManager

# write here the path to a local file or remote URL
LOCAL_FILE_OR_URL = ""

dataset_manager = DatasetManager(auth_client)
dataset = dataset_manager.create(LOCAL_FILE_OR_URL)
```

The Import API is asynchronous, but the *DatasetManager* waits a maximum of 150 seconds for the dataset to be uploaded, so once it finishes the dataset has been created in CARTO.

7.1 Import a sync dataset

You can do it in the same way as a regular dataset, just include a sync_time parameter with a value >= 900 seconds

```
from carto.datasets import DatasetManager

# how often to sync the dataset (in seconds)
SYNC_TIME = 900
# write here the URL for the dataset to sync
URL_TO_DATASET = ""

dataset_manager = DatasetManager(auth_client)
dataset = dataset_manager.create(URL_TO_DATASET, SYNC_TIME)
```

Alternatively, if you need to do further work with the sync dataset, you can use the *SyncTableJobManager*

```
from carto.sync_tables import SyncTableJobManager
import time

```

(continues on next page)

(continued from previous page)

```
# how often to sync the dataset (in seconds)
SYNC_TIME = 900
# write here the URL for the dataset to sync
URL_TO_DATASET = ""

syncTableManager = SyncTableJobManager(auth_client)
syncTable = syncTableManager.create(URL_TO_DATASET, SYNC_TIME)

# return the id of the sync
sync_id = syncTable.get_id()

while(syncTable.state != 'success'):
    time.sleep(5)
    syncTable.refresh()
    if (syncTable.state == 'failure'):
        print('The error code is: ' + str(syncTable.error_code))
        print('The error message is: ' + str(syncTable.error_message))
        break

# force sync
syncTable.refresh()
syncTable.force_sync()
```

7.2 Get a list of all the current import jobs

```
from carto.file_import import FileImportJobManager

file_import_manager = FileImportJobManager(auth_client)
file_imports = file_import_manager.all()
```

7.3 Get all the datasets

```
from carto.datasets import DatasetManager

dataset_manager = DatasetManager(auth_client)
datasets = dataset_manager.all()
```

7.4 Get a specific dataset

```
from carto.datasets import DatasetManager

# write here the ID of the dataset to retrieve
DATASET_ID = ""

dataset_manager = DatasetManager(auth_client)
dataset = dataset_manager.get(DATASET_ID)
```

7.5 Delete a dataset

```
from carto.datasets import DatasetManager

# write here the ID of the dataset to retrieve
DATASET_ID = ""

dataset_manager = DatasetManager(auth_client)
dataset = dataset_manager.get(DATASET_ID)
dataset.delete()
```

Please refer to the *carto package API documentation* and the **examples** folder to find out about the rest of the parameters accepted by constructors and methods.

7.6 External database connectors

The CARTO Python client implements the database connectors feature of the Import API

The database connectors allow importing data from an external database into a CARTO table by using the *connector* parameter.

There are several types of database connectors that you can connect to your CARTO account.

Please refer to the database connectors documentation for supported external databases.

As an example, this code snippets imports data from a Hive table into CARTO:

```
from carto.datasets import DatasetManager

dataset_manager = DatasetManager(auth_client)

connection = {
  "connector": {
    "provider": "hive",
    "connection": {
      "server": "YOUR_SERVER_IP",
      "database": "default",
      "username": "YOUR_USER_NAME",
      "password": "YOUR_PASSWORD"
    },
    "schema": "default",
    "table": "YOUR_HIVE_TABLE"
  }
}

table = dataset_manager.create(None, None, connection=connection)
```

You still can configure a sync external database connector, by providing the *interval* parameter:

```
table = dataset_manager.create(None, 900, connection=connection)
```

7.7 DatasetManager vs FileImportJobManager and SyncTableJobManager

The *DatasetManager* is conceptually different from both *FileImportJobManager* and *SyncTableJobManager*. These later ones are *JobManagers*, that means that they create and return a job using the CARTO Import API. It's responsibility of the developer to check the *state* of the job to know whether the dataset import job is completed, or has failed, errored, etc.

As an example, this code snippet uses the *FileImportJobManager* to create an import job:

```
# write here the URL for the dataset or the path to a local file (local to the server.
↪..)
LOCAL_FILE_OR_URL = "https://academy.cartodb.com/d/tornadoes.zip"

file_import_manager = FileImportJobManager(auth_client)
file_import = file_import_manager.create(LOCAL_FILE_OR_URL)

# return the id of the import
file_id = file_import.get_id()

file_import.run()
while(file_import.state != "complete" and file_import.state != "created"
            and file_import.state != "success"):
    time.sleep(5)
    file_import.refresh()
    if (file_import.state == 'failure'):
        print('The error code is: ' + str(file_import))
        break
```

Note that with the *FileImportJobManager* we are creating an import job and we check the *state* of the job.

On the other hand the *DatasetManager* is an utility class that works at the level of *Dataset*. It creates and returns a *Dataset* instance. Internally, it uses a *FileImportJobManager* or a *SyncTableJobManager* depending on the parameters received and is able to automatically *check* the *state* of the job it creates to properly return a *Dataset* instance once the job finishes successfully or a *CartoException* in any other case.

As an example, this code snippet uses the *DatasetManager* to create a dataset:

```
# write here the path to a local file (local to the server...) or remote URL
LOCAL_FILE_OR_URL = "https://academy.cartodb.com/d/tornadoes.zip"

# to use the DatasetManager you need an enterprise account
auth_client = APIKeyAuthClient(BASE_URL, API_KEY)

dataset_manager = DatasetManager(auth_client)
dataset = dataset_manager.create(LOCAL_FILE_OR_URL)

# the create method will wait up to 10 minutes until the dataset is uploaded.
```

In this case, you don't have to check the *state* of the import job, since it's done automatically by the *DatasetManager*. On the other hand, you get a *Dataset* instance as a result, instead of a *FileImportJob* instance.

CHAPTER 8

Maps API

The *Maps API* allows to create and instantiate named and anonymous maps:

```
from carto.maps import NamedMapManager, NamedMap
import json

# write the path to a local file with a JSON named map template
JSON_TEMPLATE = ""

named_map_manager = NamedMapManager(auth_client)
named_map = NamedMap(named_map_manager.client)

with open(JSON_TEMPLATE) as named_map_json:
    template = json.load(named_map_json)

# Create named map
named = named_map_manager.create(template=template)
```

```
from carto.maps import AnonymousMap
import json

# write the path to a local file with a JSON named map template
JSON_TEMPLATE = ""

anonymous = AnonymousMap(auth_client)
with open(JSON_TEMPLATE) as anonymous_map_json:
    template = json.load(anonymous_map_json)

# Create anonymous map
anonymous.instantiate(template)
```

8.1 Instantiate a named map

```
from carto.maps import NamedMapManager, NamedMap
import json

# write the path to a local file with a JSON named map template
JSON_TEMPLATE = ""

# write here the ID of the named map
NAMED_MAP_ID = ""

# write here the token you set to the named map when created
NAMED_MAP_TOKEN = ""

named_map_manager = NamedMapManager(auth_client)
named_map = named_map_manager.get(NAMED_MAP_ID)

with open(JSON_TEMPLATE) as template_json:
    template = json.load(template_json)

named_map.instantiate(template, NAMED_MAP_TOKEN)
```

8.2 Work with named maps

```
from carto.maps import NamedMapManager, NamedMap

# write here the ID of the named map
NAMED_MAP_ID = ""

# get the named map created
named_map = named_map_manager.get(NAMED_MAP_ID)

# update named map
named_map.view = None
named_map.save()

# delete named map
named_map.delete()

# list all named maps
named_maps = named_map_manager.all()
```

For more examples on how to use the *Maps API*, please refer to the **examples** folder or the *carto package API documentation*.

CHAPTER 9

Custom visualizations (aka Kuviz)

Warning: Non-public API. It may change with no previous notice

9.1 Create a Kuviz Manager

```
from carto.auth import APIKeyAuthClient
from carto.kuvizs import KuvizManager

auth_client = APIKeyAuthClient(api_key=API_KEY, base_url=BASE_URL)
km = KuvizManager(auth_client)
```

9.2 Create a Kuviz

```
html = "<html><body><h1>Working with CARTO Kuviz</h1></body></html>"
public_kuviz = km.create(html=html, name="kuviz-public-test")
```

9.3 Create a Kuviz with password

```
html = "<html><body><h1>Working with CARTO Kuviz</h1></body></html>"
password_kuviz = km.create(html=html, name="kuviz-password-test", password="1234")
```

9.4 List all Kuviz

```
kuvizs = km.all()
```

9.5 Update a kuviz

```
new_html = "<html><body><h1>Another HTML</h1></body></html>"
public_kuviz.data = new_html
public_kuviz.save()
```

9.6 Adding a password

```
public_kuviz.password = "1234"
public_kuviz.save()
```

9.7 Removing a password

```
public_kuviz.password = None
public_kuviz.save()
```

9.8 Delete

```
public_kuviz.delete()
```

CHAPTER 10

Non-public APIs

Non-public APIs may change in the future and will thrown a *warnings.warn* message when used.

Please be aware if you plan to run them on a production environment.

Refer to the *carto package API documentation* for a list of non-public APIs

CHAPTER 11

Examples

This developer guide is not intended to be an extensive list of usage examples and use cases. For that, inside the *examples* folder of the carto-python Github repository there are sample code snippets of the carto-python client.

To run examples, you should need to install additional dependencies:

```
pip install -r examples/requirements.txt
```

carto-python examples need to setup environment variables.

- CARTO_ORG: The name of your organization
- CARTO_API_URL: The *base_url* including your user and/or organization
- CARTO_API_KEY: Your user API key

Please refer to the examples source code for additional info about parameters of each one

11.1 List of examples

Find below a list of provided examples of the *carto-python* library.

Take into account that the examples are not intended to provide a comprehensive list of the capabilities of *carto-python* but only some of its use cases.

11.1.1 *change_dataset_privacy.py*

Description: Changes the privacy of a user's dataset to 'LINK', 'PUBLIC' or 'PRIVATE'

Usage example:

```
python change_dataset_privacy.py tornados LINK
```

Output:

```
12:17:01 PM - INFO - Done!
```

11.1.2 *check_query.py*

Description: Analyzes an SQL query to check if it can be optimized

Usage example:

```
python check_query.py "select version()"
```

Output:

```
12:25:18 PM - INFO - {u'QUERY PLAN': u'Result  (cost=0.00..0.00 rows=1 width=0)␣
↪(actual time=0.002..0.002 rows=1 loops=1)'}
12:25:18 PM - INFO - {u'QUERY PLAN': u'Planning time: 0.006 ms'}
12:25:18 PM - INFO - {u'QUERY PLAN': u'Execution time: 0.008 ms'}
12:25:18 PM - INFO - time: 0.002
```

11.1.3 *create_anonymous_map.py*

Description: Creates an anonymous map

Usage example:

```
python create_anonymous_map.py "files/anonymous_map.json"
```

Output:

```
Anonymous map created with layergroupid:␣
↪50b159d8a635c94fdfd100bdc7d8fb08:1493192847307
```

11.1.4 *create_named_map.py*

Description: Creates an anonymous map

Usage example:

```
python create_named_map.py "files/named_map.json"
```

Output:

```
Named map created with ID: python_sdk_test_map
```

11.1.5 *export_create_tables.py*

Description: Runs a SQL query to export the *CREATE TABLE* scripts of the user's datasets

Usage example:

```
python export_create_datasets.py
```

Output:

```
...
Found dataset: test_12
Found dataset: tornados_24

Script exported
```

11.1.6 *export_dataset.py*

Description: Exports a *dataset* in a given *format*

Usage example:

```
python export_dataset.py --dataset=tornados --format=csv
```

Output:

```
File saved: tornados.csv
```

11.1.7 *export_map.py*

Description: Exports a map visualization as a .carto file

Usage example:

```
python export_map.py "Untitled map"
```

Output:

```
URL of .carto file is: http://s3.amazonaws.com/com.cartodb.imports.production/ ... .
↪carto
```

11.1.8 *import_and_merge.py*

Description: Import a folder with CSV files (same structure) and merge them into one dataset. Files must be named as file1.csv, file2.csv, file3.csv, etc.

Usage example:

```
python import_and_merge.py "files/*.csv"
```

Output:

```
12:37:42 PM - INFO - Table imported: barris_barcelona_1_part_1
12:37:53 PM - INFO - Table imported: barris_barcelona_1_part_2
12:38:05 PM - INFO - Table imported: barris_barcelona_1_part_3
12:38:16 PM - INFO - Table imported: barris_barcelona_1_part_4
12:38:27 PM - INFO - Table imported: barris_barcelona_1_part_5
12:38:38 PM - INFO - Table imported: barris_barcelona_1_part_6
12:38:49 PM - INFO - Table imported: barris_barcelona_1_part_7
12:39:22 PM - INFO - Tables merged

URL of dataset is:       https://YOUR_ORG.carto.com/u/YOUR_USER/dataset/barris_
↪barcelona_1_part_1_merged
```

11.1.9 *import_from_database.py*

Description: External database connector

Usage example:

```
python import_from_database.py --connection='{
  "connector": {
    "provider": "hive",
    "connection": {
      "server":"YOUR_SERVER_IP",
      "database":"default",
      "username":"cloudera",
      "password":"cloudera"
    },
    "schema": "default",
    "table": "YOUR_TABLE"
  }
}'
```

Output:

```
Table imported: YOUR_TABLE
```

11.1.10 *import_standard_table.py*

Description: Creates a CARTO dataset from a URL

Usage example:

```
python import_standard_table.py files/barris_barcelona_1_part_1.csv
```

Output:

```
12:46:00 PM - INFO - Name of table: barris_barcelona_1_part_1
URL of dataset:       https://YOUR_ORG.carto.com/u/YOUR_USER/dataset/barris_barcelona_
↪1_part_1
```

11.1.11 *import_sync_table_as_dataset.py*

Description: Creates a CARTO sync dataset from a URL

Usage example:

```
python import_sync_table_as_dataset.py "https://academy.cartodb.com/d/tornadoes.zip"␣
↪900
```

Output:

```
12:48:08 PM - INFO - Name of table: tornados
URL of dataset is:       https://YOUR_ORG.carto.com/u/YOUR_USER/dataset/tornados
```

11.1.12 *import_sync_table.py*

Description: Creates a CARTO sync dataset from a URL

Usage example:

```
python import_sync_table.py "https://academy.cartodb.com/d/tornadoes.zip" 900
```

11.1.13 *instantiate_named_map.py*

Description: Instantiates a named map

Usage example:

```
python instantiate_named_map.py "python_sdk_test_map" "files/instantiate_map.json"
↪"example_token"
```

Output:

```
Done!
```

11.1.14 *kill_query.py*

Description: Kills a running query

Usage example:

```
python kill_query.py 999
```

Output:

```
Query killed
```

11.1.15 *list_tables.py*

Description: Returns graph of tables ordered by size and indicating if they are cartodbfied or not

Usage example:

```
python list_tables.py
```

Output:

```
...
analysis_a08f3b6124_a49b778b1e146f4bc7e5e670f5edcb027513ddc5 NO:       | 0.01 MB;
analysis_971639c870_c0421831d5966bcff0731772b21d6835294c4b0a NO:       | 0.01 MB;
analysis_9e88a1147e_5da714d5786b61509da4ebcd1409aae05ea8704d NO:       | 0.01 MB;
testing_moving                                               NO:       | 0.0 MB;
analysis_7530d60ffc_868bfea631fa1dc8c212ad2a8a950e050607aa6c NO:       | 0.0 MB;

There are: 338 datasets in this account
```

11.1.16 *map_info.py*

Description: Return the names of all maps or display information from a specific map

Usage example:

```
python map_info.py
```

Output:

```
12:58:28 PM - INFO - data_2_1_y_address_locations map 1
12:58:28 PM - INFO - Untitled Map 2
12:58:28 PM - INFO - Untitled map
12:58:28 PM - INFO - Untitled Map
12:58:28 PM - INFO - cartodb_germany 1
12:58:28 PM - INFO - cb_2013_us_county_500k 1
```

Usage example:

```
python map_info.py --map="Untitled map"
```

Output:

```
{ 'active_layer_id': u'5a89b00d-0a86-4a8d-a359-912458ad05c9',
  'created_at': u'2016-07-11T08:50:15+00:00',
  'description': None,
  'display_name': None,
  'id': u'7cb87e6a-4744-11e6-9b1b-0e3ff518bd15',
  'liked': False,
  'likes': 0,
  'locked': False,
  'map_id': u'7820995a-98b8-4465-9c3d-607fd5f6fa67',
  'name': u'Untitled map',
  'related_tables': [<carto.tables.Table object at 0x10aece5d0>],
  'table': <carto.tables.Table object at 0x10acb6c90>,
  'title': None,
  'updated_at': u'2016-07-11T08:50:19+00:00',
  'url': u'https://YOUR_ORG.carto.com/u/YOUR_USER/viz/7cb87e6a-4744-11e6-9b1b-
↪0e3ff518bd15/map'
}
```

11.1.17 *running_queries.py*

Description: Returns the running queries of the account

Usage example:

```
python running_queries.py
```

Output:

```
01:00:49 PM - INFO - {u'query': u'select pid, query from pg_stat_activity  WHERE␣
↪usename = current_user', u'pid': 2810}
```

11.1.18 *sql_batch_api_jobs.py*

Description: Works with a Batch SQL API job

Usage example:

```
python sql_batch_api_jobs.py create --query="select CDB_CreateOverviews('my_table
↪'::regclass)"
```

Output:

```
01:03:07 PM - INFO - status: pending
01:03:07 PM - INFO - job_id: 3a73d74d-cc7a-4faf-9c37-1bec05f4835e
01:03:07 PM - INFO - created_at: 2017-06-06T11:03:07.746Z
01:03:07 PM - INFO - updated_at: 2017-06-06T11:03:07.746Z
01:03:07 PM - INFO - user: YOUR_USER
01:03:07 PM - INFO - query: select CDB_CreateOverviews('my_table'::regclass)
```

Usage example:

```
python sql_batch_api_jobs.py read --job_id=3a73d74d-cc7a-4faf-9c37-1bec05f4835e
```

Output:

```
01:04:03 PM - INFO - status: done
01:04:03 PM - INFO - job_id: 3a73d74d-cc7a-4faf-9c37-1bec05f4835e
01:04:03 PM - INFO - created_at: 2017-06-06T11:03:07.746Z
01:04:03 PM - INFO - updated_at: 2017-06-06T11:03:08.328Z
01:04:03 PM - INFO - user: YOUR_USER
01:04:03 PM - INFO - query: select CDB_CreateOverviews('my_table'::regclass)
```

Usage example:

```
python sql_batch_api_jobs.py cancel --job_id=3a73d74d-cc7a-4faf-9c37-1bec05f4835e
```

Output:

```
01:04:03 PM - INFO - status: cancelled
01:04:03 PM - INFO - job_id: 3a73d74d-cc7a-4faf-9c37-1bec05f4835e
01:04:03 PM - INFO - created_at: 2017-06-06T11:03:07.746Z
01:04:03 PM - INFO - updated_at: 2017-06-06T11:03:08.328Z
01:04:03 PM - INFO - user: YOUR_USER
01:04:03 PM - INFO - query: select CDB_CreateOverviews('my_table'::regclass)
```

11.1.19 *table_info.py*

Description: Return columns and its types, indexes, functions and triggers of a specific table

Usage example:

```
python table_info.py tornados
```

Output:

```
General information
+------------+----------------+------------------------+----------------------+-------
↪--------+
```

(continues on next page)

(continued from previous page)

```
| Table name | Number of rows | Size of the table (MB) | Privacy of the table |␣
↪Geometry type |
+------------+----------------+------------------------+----------------------+-------
↪--------+
|  tornados  |     14222      |          2.03          |        PUBLIC        | [u'ST_
↪Point'] |
+------------+----------------+------------------------+----------------------+-------
↪--------+

The columns and their data types are:

+----------------------+-------------------+
| Column name          |        Data type |
+----------------------+-------------------+
| cartodb_id           |            bigint |
| the_geom             |     USER-DEFINED |
| the_geom_webmercator |     USER-DEFINED |
| latitude             | double precision |
| longitude            | double precision |
| damage               |           numeric |
| _feature_count       |           integer |
+----------------------+-------------------+

Indexes of the tables:

+----------------------------------+-----------------------------------------------
↪-----------------------------------------------+
| Index name                       |                                              ␣
↪                            Index definition |
+----------------------------------+-----------------------------------------------
↪-----------------------------------------------+
| _auto_idx_tornados_damage        |                      CREATE INDEX _auto_idx_
↪tornados_damage ON tornados USING btree (damage) |
| tornados_the_geom_webmercator_idx | CREATE INDEX tornados_the_geom_webmercator_idx␣
↪ON tornados USING gist (the_geom_webmercator) |
| tornados_the_geom_idx            |                        CREATE INDEX tornados_
↪the_geom_idx ON tornados USING gist (the_geom) |
| tornados_pkey                    |                       CREATE UNIQUE INDEX␣
↪tornados_pkey ON tornados USING btree (cartodb_id) |
+----------------------------------+-----------------------------------------------
↪-----------------------------------------------+

Functions of the account:

+---------------+
| Function name |
+---------------+
+---------------+

Triggers of the account:

+-------------------------------------+
|             Trigger Name            |
+-------------------------------------+
|              test_quota             |
|          test_quota_per_row         |
|            track_updates            |
```

(continues on next page)

(continued from previous page)

```
| update_the_geom_webmercator_trigger |
+-------------------------------------+
```

11.1.20 *user_info.py*

Description: Returns information from a specific user

Usage example:

```
export CARTO_USER=YOUR_USER
python user_info.py
```

Output:

```
The attributes of the user are:

+---------------------------+-------------------------------------------------------------
↪-----------------------------------------------+
| Attribute                 | Value
↪                                              |
+---------------------------+-------------------------------------------------------------
↪-----------------------------------------------+
| username                  | YOUR_USER
↪                                              |
| avatar_url                | //cartodb-libs.global.ssl.fastly.net/cartodbui/assets/
↪unversioned/images/avatars/avatar_pacman_green.png |
| quota_in_bytes            | 20198485636
↪                                              |
| public_visualization_count | 0
↪                                              |
| base_url                  | https://YOUR_ORG.carto.com/u/YOUR_USER
↪                                              |
| table_count               | 217
↪                                              |
| all_visualization_count   | 80
↪                                              |
| client                    | <carto.auth.APIKeyAuthClient object at 0x102eac710>
↪                                              |
| soft_geocoding_limit      | True
↪                                              |
| db_size_in_bytes          | 13867610112
↪                                              |
| email                     | XXX@yyy.zzz
↪                                  |
+---------------------------+-------------------------------------------------------------
↪-----------------------------------------------+

The quotas of the user are:

+----------------+----------+------------------+------------+----------------+
|        Service | Provider |       Soft limit | Used quota | Monthly quota |
+----------------+----------+------------------+------------+----------------+
|       isolines |       37 |         heremaps | False      |         100000 |
| hires_geocoder |    20238 |         heremaps | False      |         100000 |
|        routing |        0 |           mapzen | False      |         200000 |
```

(continues on next page)

(continued from previous page)

```
|    observatory |   482896 | data observatory | False      |       1000000 |
+----------------+----------+------------------+------------+---------------+
```

CHAPTER 12

carto-python API docs

12.1 carto package API documentation

12.1.1 carto.auth module

Module for authenticated access to CARTO's APIs

class carto.auth.**APIKeyAuthClient**(*base_url*, *api_key*, *organization=None*, *session=None*, *client_id=None*, *user_agent=None*)

Bases: carto.auth._UsernameGetter, carto.auth._BaseUrlChecker, carto.auth._ClientIdentifier, pyrestcli.auth.BaseAuthClient

This class provides you with authenticated access to CARTO's APIs using your API key.

You can find your API key by clicking on the API key section of the user dropdown menu

prepare_send(*http_method*, ***requests_args*)

send(*relative_path*, *http_method*, ***requests_args*)

Makes an API-key-authorized request

Parameters

- **relative_path** (*str*) – URL path relative to self.base_url
- **http_method** (*str*) – HTTP method
- **requests_args** (*kwargs*) – kwargs to be sent to requests

Returns A request response object

Raise CartoException

class carto.auth.**AuthAPIClient**(*base_url*, *api_key*, *organization=None*, *session=None*)

Bases: carto.auth._UsernameGetter, carto.auth._BaseUrlChecker, pyrestcli.auth.BasicAuthClient

This class provides you with authenticated access to CARTO's APIs using your API key at Basic authentication header, as provided by Auth API.

Auth API is still under development. You might want to take a look at APIKeyAuthClient for missing features or an stable API.

You can find your API key by clicking on the API key section of the user dropdown menu

is_valid_api_key()
Checks validity. Right now, an API key is considered valid if it can list user API keys and the result contains that API key. This might change in the future.

Returns True if the API key is considered valid for current user.

class carto.auth.**NonVerifiedAPIKeyAuthClient**(*base_url*, *api_key*, *organization=None*, *session=None*)
Bases: *carto.auth.APIKeyAuthClient*

This class provides you with authenticated access to CARTO's APIs using your API key but avoids verifying SSL certificates. This is useful for onpremises instances of CARTO

You can find your API key by clicking on the API key section of the user dropdown menu

send(*relative_path*, *http_method*, ***requests_args*)
Makes an API-key-authorized request not verifying SSL certs

Parameters

- **relative_path** (*str*) – URL path relative to self.base_url
- **http_method** (*str*) – HTTP method
- **requests_args** (*kwargs*) – kwargs to be sent to requests

Returns A request response object

Raise CartoException

12.1.2 carto.api_keys module

Module for working with CARTO Auth API

https://carto.com/developers/auth-api/

class carto.api_keys.**APIKey**(*auth_client*, ***kwargs*)
Bases: pyrestcli.resources.Resource

Represents an API key in CARTO. API keys are used to grant permissions to tables and maps. See the Auth API reference for more info: https://carto.com/developers/auth-api/

class Meta

collection_endpoint = 'api/v3/api_keys/'

id_field = 'id'

json_data = True

name_field = 'name'

parse_json = True

created_at
Field to store datetimes in resources

fields = ['name', 'type', 'created_at', 'updated_at', 'token', 'grants']

grants
carto.api_keys.Grants

name
Convenient class to make explicit that an attribute will store chars

regenerate_token()
Regenerates the associated token

Returns

Raise CartoException

token
Convenient class to make explicit that an attribute will store chars

type
Convenient class to make explicit that an attribute will store chars

updated_at
Field to store datetimes in resources

class carto.api_keys.**APIKeyManager**(*auth_client*)
Bases: *carto.resources.Manager*

Manager for the APIKey class.

create(*name, apis=['sql', 'maps'], tables=None, services=None*)
Creates a regular APIKey.

Parameters

- **name** (*str*) – The API key name
- **apis** (*list*) – Describes which APIs does this API Key provide access to
- **tables** (*TableGrant or dict*) – Describes to which tables and which privleges on each table this API Key grants access to
- **services** (*list*) – Describes to which data services this API Key grants access to

Returns An APIKey instance with a token

json_collection_attribute = 'result'

paginator_class
alias of *carto.paginators.CartoPaginator*

resource_class
alias of *APIKey*

class carto.api_keys.**Grants**(*auth_client*, ***kwargs*)
Bases: pyrestcli.resources.Resource

apis
Convenient class to make explicit that an attribute will store chars

fields = ['tables', 'apis', 'services']

get_id()

services
Convenient class to make explicit that an attribute will store chars

tables
carto.api_keys.TableGrant

class carto.api_keys.**TableGrant**(*auth_client*, ***kwargs*)
Bases: pyrestcli.resources.Resource

Describes to which tables and which privleges on each table this API Key grants access to trough tables attribute. This is an internal data type, with no specific API endpoints

See https://carto.com/developers/auth-api/reference/#section/API-Key-format

Example:

```
{
    "type": "database",
    "tables": [
        {
            "schema": "public",
            "name": "my_table",
            "permissions": [
                "insert",
                "select",
                "update"
            ]
        }
    ]
}
```

fields = ['name', 'permissions', 'schema']

name
Convenient class to make explicit that an attribute will store chars

permissions
Convenient class to make explicit that an attribute will store chars

schema
Convenient class to make explicit that an attribute will store chars

to_json()

12.1.3 carto.datasets module

Module for working with CARTO datasets

class carto.datasets.**Dataset**(*auth_client*, ***kwargs*)
Bases: *carto.resources.WarnResource*

Represents a dataset in CARTO. Typically, that means there is a table in the PostgreSQL server associated to this object.

> **Warning:** Non-public API. It may change with no previous notice

class Meta

collection_endpoint = 'api/v1/viz/'

id_field = 'id'

json_data = True

name_field = 'name'

parse_json = True

active_child
Convenient class to make explicit that an attribute will store chars

active_layer_id
Convenient class to make explicit that an attribute will store chars

attributions
Convenient class to make explicit that an attribute will store chars

auth_tokens
Convenient class to make explicit that an attribute will store chars

children
Convenient class to make explicit that an attribute will store chars

connector
Convenient class to make explicit that an attribute will store chars

created_at
Field to store datetimes in resources

delete()
Deletes the resource from the server; Python object remains untouched :return:

dependent_visualizations
carto.visualizations.Visualization

dependent_visualizations_count
Convenient class to make explicit that an attribute will store integers

description
Convenient class to make explicit that an attribute will store chars

display_name
Convenient class to make explicit that an attribute will store chars

external_source
Convenient class to make explicit that an attribute will store chars

fields = ['dependent_visualizations', 'liked', 'prev_id', 'likes', 'active_layer_id',

force_delete()

id
Convenient class to make explicit that an attribute will store chars

kind
Convenient class to make explicit that an attribute will store chars

license
Convenient class to make explicit that an attribute will store chars

liked
Convenient class to make explicit that an attribute will store booleans

likes
Convenient class to make explicit that an attribute will store integers

locked
Convenient class to make explicit that an attribute will store booleans

map_id
Convenient class to make explicit that an attribute will store chars

name
Convenient class to make explicit that an attribute will store chars

next_id
Convenient class to make explicit that an attribute will store chars

parent_id
Convenient class to make explicit that an attribute will store chars

permission
carto.permissions.Permission

prev_id
Convenient class to make explicit that an attribute will store chars

privacy
Convenient class to make explicit that an attribute will store chars

source
Convenient class to make explicit that an attribute will store chars

stats
Field to store datetimes in resources

synchronization
carto.synchronizations.Synchronization

table
carto.tables.Table

tags
Convenient class to make explicit that an attribute will store chars

title
Convenient class to make explicit that an attribute will store chars

transition_options
Convenient class to make explicit that an attribute will store a dictionary

type
Convenient class to make explicit that an attribute will store chars

updated_at
Field to store datetimes in resources

url
Convenient class to make explicit that an attribute will store chars

user
carto.users.User

uses_builder_features
Convenient class to make explicit that an attribute will store booleans

class carto.datasets.**DatasetManager**(*auth_client*)
Bases: *carto.resources.Manager*

Manager for the Dataset class.

> **Warning:** Non-public API. It may change with no previous notice

create(*archive*, *interval=None*, ***import_args*)
Creating a table means uploading a file or setting up a sync table

Parameters

- **archive** (*str*) – URL to the file (both remote URLs or local paths are supported) or StringIO object
- **interval** (*int*) – Interval in seconds. If not None, CARTO will try to set up a sync table against the (remote) URL
- **import_args** (*kwargs*) – Arguments to be sent to the import job when run

Returns New dataset object

Return type *Dataset*

Raise CartoException

is_sync_table(*archive*, *interval*, ***import_args*)
Checks if this is a request for a sync dataset.

The condition for creating a sync dataset is to provide a URL or a connection to an external database and an interval in seconds

Parameters

- **archive** – URL to the file (both remote URLs or local paths are supported) or StringIO object
- **interval** (*int*) – Interval in seconds.
- **import_args** (*kwargs*) – Connection parameters for an external database

Returns True if it is a sync dataset

json_collection_attribute = 'visualizations'

paginator_class
alias of *carto.paginators.CartoPaginator*

resource_class
alias of *Dataset*

send(*url*, *http_method*, ***client_args*)
Sends an API request, taking into account that datasets are part of the visualization endpoint.

Parameters

- **url** (*str*) – Endpoint URL
- **http_method** (*str*) – The method used to make the request to the API
- **client_args** (*kwargs*) – Arguments to be sent to the auth client

Returns A request response object

Raise CartoException

12.1.4 carto.exceptions module

Module for carto-python exceptions definitions

exception carto.exceptions.**CartoException**
Bases: exceptions.Exception

Any Exception produced by carto-python should be wrapped around this class

exception carto.exceptions.**CartoRateLimitException**(*response*)
Bases: *carto.exceptions.CartoException*

This exception is raised when a request is rate limited by SQL or Maps APIs (429 Too Many Requests HTTP error). More info about CARTO rate limits: https://carto.com/developers/fundamentals/limits/#rate-limits

It extends CartoException with the rate limits info, so that any client can manage when to retry a rate limited request.

static is_rate_limited(*response*)
Checks if the response has been rate limited by CARTO APIs

Parameters **response** (*requests.models.Response class*) – The response rate limited by CARTO APIs

Returns Boolean

12.1.5 carto.export module

Module for exporting visualizations

class carto.export.**ExportJob**(*client*, *visualization_id*)
Bases: *carto.resources.WarnAsyncResource*

Equivalent to a .carto export in CARTO.

Allows a CARTO export to be created using a visualization in the user's CARTO account

> **Warning:** Non-public API. It may change with no previous notice

class Meta

collection_endpoint = 'api/v3/visualization_exports/'

id_field = 'id'

json_data = True

name_field = 'id'

parse_json = True

created_at
Field to store datetimes in resources

fields = ['user_id', 'url', 'created_at', 'updated_at', 'state', 'visualization_id', '

id
Convenient class to make explicit that an attribute will store chars

run(***export_params*)
Make the actual request to the Import API (exporting is part of the Import API).

Parameters **export_params** (*kwargs*) – Any additional parameters to be sent to the Import API

Returns

Note: The export is asynchronous, so you should take care of the progression, by calling the *carto.resources.AsyncResource.refresh()* method and check the export job *state* attribute. See *carto.visualizations.Visualization.export()* method implementation for more details

state
Convenient class to make explicit that an attribute will store chars

updated_at
Field to store datetimes in resources

url
Convenient class to make explicit that an attribute will store chars

user_id
Convenient class to make explicit that an attribute will store chars

visualization_id
Convenient class to make explicit that an attribute will store chars

12.1.6 carto.fields module

Module for defining response objects

class carto.fields.**Base64EncodedField**(*many=False*)
Bases: pyrestcli.fields.CharField

class carto.fields.**EntityField**(*many=False*)
Bases: pyrestcli.fields.ResourceField

carto.permissions.Entity

value_class = 'carto.permissions.Entity'

class carto.fields.**GrantsField**(*many=False*)
Bases: pyrestcli.fields.ResourceField

carto.api_keys.Grants

type_field = {'apis': 'apis', 'services': 'dataservices', 'tables': 'database'}

value_class = 'carto.api_keys.Grants'

class carto.fields.**PasswordAndPrivacyFields**(*many=False*)
Bases: pyrestcli.fields.CharField

class carto.fields.**PermissionField**(*many=False*)
Bases: pyrestcli.fields.ResourceField

carto.permissions.Permission

value_class = 'carto.permissions.Permission'

class carto.fields.**SynchronizationField**(*many=False*)
Bases: pyrestcli.fields.ResourceField

carto.synchronizations.Synchronization

value_class = 'carto.synchronizations.Synchronization'

class carto.fields.**TableField**(*many=False*)
Bases: pyrestcli.fields.ResourceField

carto.tables.Table

value_class = 'carto.tables.Table'

class carto.fields.**TableGrantField**(*many=False*)
Bases: pyrestcli.fields.ResourceField

carto.api_keys.TableGrant

value_class = 'carto.api_keys.TableGrant'

class carto.fields.**UserField**(*many=False*)
Bases: pyrestcli.fields.ResourceField

carto.users.User

value_class = 'carto.users.User'

class carto.fields.**VisualizationField**(*many=False*)
Bases: pyrestcli.fields.ResourceField

carto.visualizations.Visualization

value_class = 'carto.visualizations.Visualization'

12.1.7 carto.file_import module

Module for importing remote and local files into CARTO

class carto.file_import.**FileImportJob**(*archive*, *auth_client*)
Bases: *carto.resources.AsyncResource*

This class provides support for one-time uploading and importing of remote and local files into CARTO

class Meta

collection_endpoint = 'api/v1/imports/'

id_field = 'item_queue_id'

json_data = True

name_field = 'id'

parse_json = True

content_guessing
Convenient class to make explicit that an attribute will store booleans

create_visualization
Convenient class to make explicit that an attribute will store booleans

data_type
Convenient class to make explicit that an attribute will store chars

display_name
Convenient class to make explicit that an attribute will store chars

error_code
Convenient class to make explicit that an attribute will store integers

fields = ['quoted_fields_guessing', 'data_type', 'queue_id', 'user_defined_limits', 'i

get_error_text = None

id
Convenient class to make explicit that an attribute will store chars

is_raster
Convenient class to make explicit that an attribute will store booleans

item_queue_id
Convenient class to make explicit that an attribute will store chars

queue_id
Convenient class to make explicit that an attribute will store chars

quoted_fields_guessing
Convenient class to make explicit that an attribute will store booleans

run(***import_params*)
Actually creates the import job on the CARTO server

Parameters **import_params** (*kwargs*) – To be send to the Import API, see CARTO's docs on Import API for an updated list of accepted params

Returns

Note: The import job is asynchronous, so you should take care of the progression, by calling the *carto.resources.AsyncResource.refresh()* method and check the import job *state* attribute. See *carto.datasets.DatasetManager.create()* for a unified method to import files into CARTO

---

state
Convenient class to make explicit that an attribute will store chars

success
Convenient class to make explicit that an attribute will store booleans

synchronization_id
Convenient class to make explicit that an attribute will store chars

table_id
Convenient class to make explicit that an attribute will store chars

table_name
Convenient class to make explicit that an attribute will store chars

tables_created_count
Convenient class to make explicit that an attribute will store integers

type_guessing
Convenient class to make explicit that an attribute will store booleans

user_defined_limits
Convenient class to make explicit that an attribute will store chars

user_id
Convenient class to make explicit that an attribute will store chars

visualization_id
Convenient class to make explicit that an attribute will store chars

warnings = None

class carto.file_import.**FileImportJobManager**(*auth_client*)
Bases: *carto.resources.Manager*

create(*archive*, ***kwargs*)
Creates a file import on the server

Parameters

- **archive** (*str*) – archive can be a pointer to a remote location, a path to a local file or a StringIO object
- **kwargs** (*kwargs*) – Attributes (field names and values) of the new resource

Returns The *carto.file_import.FileImportJob*

filter()
Get a filtered list of file imports

Returns A list of file imports, with only the id set (you need to refresh them if you want all the attributes to be filled in)

Return type list of *carto.file_import.FileImportJob*

Raise CartoException

json_collection_attribute = 'imports'

paginator_class
alias of *carto.paginators.CartoPaginator*

resource_class
alias of *FileImportJob*

12.1.8 carto.maps module

Module for working with named and anonymous maps

class carto.maps.**AnonymousMap**(*auth_client*)
Bases: *carto.maps.BaseMap*

Equivalent to creating an anonymous map in CARTO.

class Meta

collection_endpoint = 'api/v1/map/'

id_field = 'id'

json_data = True

name_field = 'id'

parse_json = True

fields = []

instantiate(*params*)
Allows you to fetch the map tiles of a created map

Parameters **params** (*dict*) – The json with the styling info for the named map

Returns

Raise CartoException

update_from_dict(*attribute_dict*)
Update the fields of the resource out of a data dictionary taken out of an API response :param attribute_dict: Dictionary to be mapped into object attributes :return:

class carto.maps.**BaseMap**(*auth_client*)
Bases: pyrestcli.resources.Resource

Base class for NamedMap and AnonymousMap

fields = []

get_tile_url(*x*, *y*, *z*, *layer_id=None*, *feature_id=None*, *filter=None*, *extension='png'*)
Prepares a URL to get data (raster or vector) from a NamedMap or AnonymousMap

Parameters

- **x** (*int*) – The x tile
- **y** (*int*) – The y tile
- **z** (*int*) – The zoom level
- **layer_id** (*str*) – Can be a number (referring to the # layer of your map), all layers of your map, or a list of layers. To show just the basemap layer, enter the value 0 To show the first layer, enter the value 1 To show all layers, enter the value 'all' To show a list of layers, enter the comma separated layer value as '0,1,2'
- **feature_id** (*str*) – The id of the feature
- **filter** (*str*) – The filter to be applied to the layer
- **extension** (*str*) – The format of the data to be retrieved: png, mvt, ...

Returns A URL to download data

Return type str

Raise CartoException

class carto.maps.**NamedMap**(*auth_client*)
Bases: *carto.maps.BaseMap*

Equivalent to creating a named map in CARTO.

class Meta

collection_endpoint = 'api/v1/map/named/'

id_field = 'template_id'

json_data = True

name_field = 'name'

parse_json = True

fields = []

instantiate(*params*, *auth=None*)
Allows you to fetch the map tiles of a created map

Parameters

- **params** (*dict*) – The json with the styling info for the named map
- **auth** (*carto.auth.APIKeyAuthClient*) – The auth client

Returns

Raise CartoException

update_from_dict(*attribute_dict*)
Method overriden from the base class

class carto.maps.**NamedMapManager**(*auth_client*)
Bases: pyrestcli.resources.Manager

Manager for the NamedMap class

create(***kwargs*)
Creates a named map

Parameters **kwargs** (*kwargs*) – Attributes for creating the named map. Specifically an attribute *template* must contain the JSON object defining the named map

Returns New named map object

Return type *NamedMap*

Raise CartoException

json_collection_attribute = 'template_ids'

resource_class
alias of *NamedMap*

12.1.9 carto.paginators module

Used internally to retrieve results paginated

class carto.paginators.**CartoPaginator**(*json_collection_attribute*, *base_url*, *params=None*)
Bases: pyrestcli.paginators.Paginator

Used internally to retrieve results paginated

get_urls(*initial_url*)

process_response(*response*)

12.1.10 carto.permissions module

Entity classes for defining permissions

class carto.permissions.**Entity**(*auth_client*, ***kwargs*)
Bases: pyrestcli.resources.Resource

Represents an entity in CARTO. This is an internal data type, with no specific API endpoints

fields = ['type', 'id']

id
Convenient class to make explicit that an attribute will store chars

type
Convenient class to make explicit that an attribute will store chars

class carto.permissions.**Permission**(*auth_client*, ***kwargs*)
Bases: pyrestcli.resources.Resource

Represents a permission in CARTO. This is an internal data type, with no specific API endpoints

acl = None

created_at
Field to store datetimes in resources

entity
carto.permissions.Entity

fields = ['created_at', 'updated_at', 'entity', 'id', 'owner']

id
Convenient class to make explicit that an attribute will store chars

owner
carto.users.User

updated_at
Field to store datetimes in resources

12.1.11 carto.resources module

Extensions for pyrestcli Resource and Manager classes

class carto.resources.**AsyncResource**(*auth_client*, ***kwargs*)
Bases: pyrestcli.resources.Resource

fields = []

refresh()
Updates the information of the async job against the CARTO server. After calling the *refresh()* method you should check the *state* attribute of your resource

Returns

run(***client_params*)
Actually creates the async job on the CARTO server

Parameters **client_params** (*kwargs*) – To be send to the CARTO API. See CARTO's documentation depending on the subclass you are using

Returns

Raise CartoException

class carto.resources.**Manager**(*auth_client*)
Bases: pyrestcli.resources.Manager

Manager class for resources

class carto.resources.**WarnAsyncResource**(*auth_client*, ***kwargs*)
Bases: *carto.resources.AsyncResource*

AsyncResource class for resources that represent non-public CARTO APIs. You'll be warned not to used the in production environments

fields = []

class carto.resources.**WarnResource**(*auth_client*, ***kwargs*)
Bases: pyrestcli.resources.Resource

Resource class for resources that represent non-public CARTO APIs. You'll be warned not to used the in production environments

fields = []

12.1.12 carto.sql module

Module for the SQL API

class carto.sql.**BatchSQLClient**(*client*, *api_version='v2'*)
Bases: object

Allows you to send requests to CARTO's Batch SQL API

cancel(*job_id*)
Cancels a job

Parameters **job_id** (*str*) – The id of the job to be cancelled

Returns A status code depending on whether the cancel request was successful

Return type str

Raises ***CartoException*** –

create(*sql_query*)
Creates a new batch SQL query.

Batch SQL jobs are asynchronous, once created you should call *carto.sql.BatchSQLClient.read()* method given the *job_id* to retrieve the state of the batch query

Parameters **sql_query** (*str or list of str*) – The SQL query to be used

Returns Response data, either as json or as a regular response.content object

Return type object

Raise CartoException

create_and_wait_for_completion(*sql_query*)
Creates a new batch SQL query and waits for its completion or failure

Batch SQL jobs are asynchronous, once created this method automatically queries the job status until it's one of 'done', 'failed', 'canceled', 'unknown'

Parameters **sql_query** (*str or list of str*) – The SQL query to be used

Returns Response data, either as json or as a regular response.content object

Return type object

Raise CartoException when there's an exception in the BatchSQLJob execution or the batch job status is one of the BATCH_JOBS_FAILED_STATUSES ('failed', 'canceled', 'unknown')

read(*job_id*)
Reads the information for a specific Batch API request

Parameters **job_id** (*str*) – The id of the job to be read from

Returns Response data, either as json or as a regular response.content object

Return type object

Raise CartoException

send(*url*, *http_method*, *json_body=None*, *http_header=None*)
Executes Batch SQL query in a CARTO server

Parameters

- **url** (*str*) – Endpoint url
- **http_method** (*str*) – The method used to make the request to the API

- **json_body** (*dict*) – The information that needs to be sent, by default is set to None
- **http_header** (*str*) – The header used to make write requests to the API, by default is none

Returns Response data, either as json or as a regular response.content object

Return type object

Raise CartoException

update(*job_id*, *sql_query*)
Updates the sql query of a specific job

Parameters

- **job_id** (*str*) – The id of the job to be updated
- **sql_query** (*str*) – The new SQL query for the job

Returns Response data, either as json or as a regular response.content object

Return type object

Raise CartoException

update_from_dict(*data_dict*)

Parameters **data_dict** (*dict*) – Dictionary to be mapped into object attributes

Returns

class carto.sql.**CopySQLClient**(*client*, *api_version='v2'*)
Bases: object

Allows to use the PostgreSQL COPY command for efficient streaming of data to and from CARTO.

copyfrom(*query*, *iterable_data*, *compress=True*, *compression_level=1*)
Gets data from an iterable object into a table

Parameters

- **query** (*str*) – The "COPY table_name [(column_name[, ...])] FROM STDIN [WITH(option[,...])]" query to execute
- **iterable_data** (*object*) – An object that can be iterated to retrieve the data

Returns Response data as json

Return type str

Raises ***CartoException*** –

copyfrom_file_object(*query*, *file_object*, *compress=True*, *compression_level=1*)
Gets data from a readable file object into a table

Parameters

- **query** (*str*) – The "COPY table_name [(column_name[, ...])] FROM STDIN [WITH(option[,...])]" query to execute
- **file_object** (*file*) – A file-like object. Normally the return value of open('file.ext', 'rb')

Returns Response data as json

Return type str

Raises ***CartoException*** –

copyfrom_file_path(*query*, *path*, *compress=True*, *compression_level=1*)
Gets data from a readable file into a table

Parameters

- **query** (*str*) – The "COPY table_name [(column_name[, ...])] FROM STDIN [WITH(option[,...])]" query to execute
- **path** (*str*) – A path to a file

Returns Response data as json

Return type str

Raises ***CartoException*** –

copyto(*query*)
Gets data from a table into a Response object that can be iterated

Parameters **query** (*str*) – The "COPY { table_name [(column_name[, ...])] | (query) } TO STDOUT [WITH(option[,...])]" query to execute

Returns response object

Return type Response

Raises ***CartoException*** –

copyto_file_object(*query*, *file_object*)
Gets data from a table into a writable file object

Parameters

- **query** (*str*) – The "COPY { table_name [(column_name[, ...])] | (query) } TO STDOUT [WITH(option[,...])]" query to execute
- **file_object** (*file*) – A file-like object. Normally the return value of open('file.ext', 'wb')

Raises ***CartoException*** –

copyto_file_path(*query*, *path*, *append=False*)
Gets data from a table into a writable file

Parameters

- **query** (*str*) – The "COPY { table_name [(column_name[, ...])] | (query) } TO STDOUT [WITH(option[,...])]" query to execute
- **path** (*str*) – A path to a writable file
- **append** (*bool*) – Whether to append or not if the file already exists Default value is False

Raises ***CartoException*** –

copyto_stream(*query*)
Gets data from a table into a stream

Parameters **query** (*str*) – The "COPY { table_name [(column_name[, ...])] | (query) } TO STDOUT [WITH(option[,...])]" query to execute

Returns the data from COPY TO query

Return type raw binary (text stream)

Raise CartoException

class carto.sql.**SQLClient**(*auth_client*, *api_version='v2'*)
Bases: object

Allows you to send requests to CARTO's SQL API

send(*sql*, *parse_json=True*, *do_post=True*, *format=None*, ***request_args*)
Executes SQL query in a CARTO server

Parameters

- **sql** (*str*) – The SQL
- **parse_json** (*boolean*) – Set it to False if you want raw reponse
- **do_post** (*boolean*) – Set it to True to force post request
- **format** (*str*) – Any of the data export formats allowed by CARTO's SQL API
- **request_args** (*dictionary*) – Additional parameters to send with the request

Returns response data, either as json or as a regular response.content object

Return type object

Raise CartoException

12.1.13 carto.sync_tables module

Module for the IMPORT API with sync tables

class carto.sync_tables.**SyncTableJob**(*url*, *interval*, *auth_client*)
Bases: *carto.resources.AsyncResource*

This class provides support for creating Sync Tables into CARTO

class Meta

```
collection_endpoint = 'api/v1/synchronizations/'
id_field = 'id'
json_data = True
name_field = 'id'
parse_json = True
```

checksum
Convenient class to make explicit that an attribute will store chars

content_guessing
Convenient class to make explicit that an attribute will store booleans

created_at
Field to store datetimes in resources

enqueued
Convenient class to make explicit that an attribute will store booleans

error_code
Convenient class to make explicit that an attribute will store integers

error_message
Convenient class to make explicit that an attribute will store chars

etag
Convenient class to make explicit that an attribute will store chars

fields = ['interval', 'service_name', 'updated_at', 'run_at', 'type_guessing', 'servic

force_sync()
Forces to sync the SyncTableJob

Returns

Raise CartoException

from_external_source
Convenient class to make explicit that an attribute will store booleans

get_force_sync_endpoint()
Get the relative path to the specific API resource

Returns Relative path to the resource

Raise CartoException

id
Convenient class to make explicit that an attribute will store chars

interval
Convenient class to make explicit that an attribute will store integers

log_id
Convenient class to make explicit that an attribute will store chars

modified_at
Field to store datetimes in resources

name
Convenient class to make explicit that an attribute will store chars

quoted_fields_guessing
Convenient class to make explicit that an attribute will store booleans

ran_at
Field to store datetimes in resources

retried_times
Convenient class to make explicit that an attribute will store integers

run(***import_params*)
Actually creates the job import on the CARTO server

Parameters **import_params** (*kwargs*) – To be send to the Import API, see CARTO's docs on Import API for an updated list of accepted params

Returns

Note: The sync table job is asynchronous, so you should take care of the progression, by calling the *carto.resources.AsyncResource.refresh()* method and check the import job *state* attribute. See *carto.datasets.DatasetManager.create()* for a unified method to import files into CARTO

---

run_at
Field to store datetimes in resources

service_item_id
Convenient class to make explicit that an attribute will store chars

service_name
Convenient class to make explicit that an attribute will store chars

state
Convenient class to make explicit that an attribute will store chars

success
Convenient class to make explicit that an attribute will store booleans

synchronization_id
Convenient class to make explicit that an attribute will store chars

type_guessing
Convenient class to make explicit that an attribute will store booleans

updated_at
Field to store datetimes in resources

url
Convenient class to make explicit that an attribute will store chars

user_id
Convenient class to make explicit that an attribute will store chars

visualization_id
Convenient class to make explicit that an attribute will store chars

class carto.sync_tables.**SyncTableJobManager**(*auth_client*)
Bases: *carto.resources.Manager*

Manager for the SyncTableJob class

create(*url*, *interval*, ***kwargs*)
Create a sync table on the server

Parameters

- **url** (*str*) – URL can be a pointer to a remote location or a path to a local file
- **interval** (*int*) – Sync interval in seconds
- **kwargs** (*kwargs*) – Attributes (field names and values) of the new resource

Returns SyncTableJob

json_collection_attribute = 'synchronizations'

paginator_class
alias of *carto.paginators.CartoPaginator*

resource_class
alias of *SyncTableJob*

12.1.14 carto.tables module

Module for working with tables

class carto.tables.**Table**(*auth_client*, ***kwargs*)
Bases: *carto.resources.WarnResource*

Represents a table in CARTO. This is an internal data type. Both Table and TableManager are not meant to be used outside the SDK

If you are looking to work with datasets / tables from outside the SDK, please look into the datasets.py file.

Warning: Non-public API. It may change with no previous notice

class Meta

collection_endpoint = 'api/v1/tables/'

id_field = 'id'

json_data = True

name_field = 'name'

parse_json = True

dependent_visualizations = None

description
Convenient class to make explicit that an attribute will store chars

fields = ['rows_counted', 'map_id', 'description', 'permission', 'geometry_types', 'up

geometry_types
Convenient class to make explicit that an attribute will store chars

id
Convenient class to make explicit that an attribute will store chars

map_id
Convenient class to make explicit that an attribute will store chars

name
Convenient class to make explicit that an attribute will store chars

non_dependent_visualizations = None

permission
carto.permissions.Permission

privacy
Convenient class to make explicit that an attribute will store chars

row_count
Convenient class to make explicit that an attribute will store integers

rows_counted
Convenient class to make explicit that an attribute will store integers

schema
Convenient class to make explicit that an attribute will store chars

size
Convenient class to make explicit that an attribute will store integers

synchronization
carto.synchronizations.Synchronization

table_size
Convenient class to make explicit that an attribute will store integers

table_visualization
carto.visualizations.Visualization

updated_at
Field to store datetimes in resources

class carto.tables.**TableManager**(*auth_client*)
Bases: *carto.resources.Manager*

Manager for the Table class.

> **Warning:** Non-public API. It may change with no previous notice

paginator_class
alias of *carto.paginators.CartoPaginator*

resource_class
alias of *Table*

12.1.15 carto.users module

Module for working with users

class carto.users.**User**(*auth_client*)
Bases: *carto.resources.WarnResource*

Represents an enterprise CARTO user, i.e. a user that belongs to an organization

Currently, CARTO's user API only supports enterprise users.

> **Warning:** Non-public API. It may change with no previous notice

class Meta

collection_endpoint = None

id_field = 'username'

json_data = True

name_field = 'username'

parse_json = True

all_visualization_count
Convenient class to make explicit that an attribute will store integers

available_for_hire
Convenient class to make explicit that an attribute will store booleans

avatar_url
Convenient class to make explicit that an attribute will store chars

base_url
Convenient class to make explicit that an attribute will store chars

db_size_in_bytes
Convenient class to make explicit that an attribute will store integers

description
Convenient class to make explicit that an attribute will store chars

disqus_shortname
Convenient class to make explicit that an attribute will store chars

email
Convenient class to make explicit that an attribute will store chars

fields = ['website', 'username', 'last_name', 'remove_logo', 'avatar_url', 'viewer', '

get_collection_endpoint()

get_resource_endpoint()

google_maps_query_string
Convenient class to make explicit that an attribute will store chars

last_name
Convenient class to make explicit that an attribute will store chars

location
Convenient class to make explicit that an attribute will store chars

name
Convenient class to make explicit that an attribute will store chars

org_admin
Convenient class to make explicit that an attribute will store booleans

org_user
Convenient class to make explicit that an attribute will store booleans

password
Convenient class to make explicit that an attribute will store chars

public_visualization_count
Convenient class to make explicit that an attribute will store integers

quota_in_bytes
Convenient class to make explicit that an attribute will store integers

remove_logo
Convenient class to make explicit that an attribute will store booleans

soft_geocoding_limit
Convenient class to make explicit that an attribute will store integers

table_count
Convenient class to make explicit that an attribute will store integers

twitter_username
Convenient class to make explicit that an attribute will store chars

username
Convenient class to make explicit that an attribute will store chars

viewer
Convenient class to make explicit that an attribute will store booleans

website
Convenient class to make explicit that an attribute will store chars

class carto.users.**UserManager**(*auth_client*)
Bases: *carto.resources.Manager*

Manager for the User class.

> **Warning:** Non-public API. It may change with no previous notice

filter(***search_args*)
Should get all the current users from CARTO, but this is currently not supported by the API

get_collection_endpoint()

get_resource_endpoint(*resource_id*)

paginator_class
alias of *carto.paginators.CartoPaginator*

resource_class
alias of *User*

12.1.16 carto.visualizations module

Module for working with map visualizations

class carto.visualizations.**Visualization**(*auth_client*, ***kwargs*)
Bases: *carto.resources.WarnResource*

Represents a map visualization in CARTO.

> **Warning:** Non-public API. It may change with no previous notice

class Meta

```
collection_endpoint = 'api/v1/viz/'
id_field = 'id'
json_data = True
name_field = 'name'
parse_json = True
```

active_child
Convenient class to make explicit that an attribute will store chars

active_layer_id
Convenient class to make explicit that an attribute will store chars

attributions
Convenient class to make explicit that an attribute will store chars

auth_tokens
Convenient class to make explicit that an attribute will store chars

children
Convenient class to make explicit that an attribute will store chars

created_at
Field to store datetimes in resources

description
Convenient class to make explicit that an attribute will store chars

display_name
Convenient class to make explicit that an attribute will store chars

export()
Make the actual request to the Import API (exporting is part of the Import API) to export a map visualization as a .carto file

Returns A URL pointing to the .carto file

Return type str

Raise CartoException

Warning: Non-public API. It may change with no previous notice

Note: The export is asynchronous, but this method waits for the export to complete. See *MAX_NUMBER_OF_RETRIES* and *INTERVAL_BETWEEN_RETRIES_S*

external_source
Convenient class to make explicit that an attribute will store a dictionary

fields = ['liked', 'prev_id', 'likes', 'active_layer_id', 'table', 'children', 'displa

id
Convenient class to make explicit that an attribute will store chars

kind
Convenient class to make explicit that an attribute will store chars

license
Convenient class to make explicit that an attribute will store chars

liked
Convenient class to make explicit that an attribute will store booleans

likes
Convenient class to make explicit that an attribute will store integers

locked
Convenient class to make explicit that an attribute will store booleans

map_id
Convenient class to make explicit that an attribute will store chars

name
Convenient class to make explicit that an attribute will store chars

next_id
Convenient class to make explicit that an attribute will store chars

parent_id
Convenient class to make explicit that an attribute will store chars

permission
carto.permissions.Permission

prev_id
Convenient class to make explicit that an attribute will store chars

privacy
Convenient class to make explicit that an attribute will store chars

related_tables
carto.tables.Table

source
Convenient class to make explicit that an attribute will store chars

stats
Convenient class to make explicit that an attribute will store a dictionary

synchronization
carto.synchronizations.Synchronization

table
carto.tables.Table

tags
Convenient class to make explicit that an attribute will store chars

title
Convenient class to make explicit that an attribute will store chars

transition_options
Convenient class to make explicit that an attribute will store a dictionary

type
Convenient class to make explicit that an attribute will store chars

updated_at
Field to store datetimes in resources

url
Convenient class to make explicit that an attribute will store chars

uses_builder_features
Convenient class to make explicit that an attribute will store booleans

version
Convenient class to make explicit that an attribute will store integers

class carto.visualizations.**VisualizationManager**(*auth_client*)
Bases: *carto.resources.Manager*

Manager for the Visualization class.

> **Warning:** Non-public API. It may change with no previous notice

create(***kwargs*)
Creating visualizations is better done by using the Maps API (named maps) or directly from your front end app if dealing with public datasets

json_collection_attribute = 'visualizations'

paginator_class
alias of *carto.paginators.CartoPaginator*

resource_class
alias of *Visualization*

send(*url*, *http_method*, ***client_args*)
Sends API request, taking into account that visualizations are only a subset of the resources available at the visualization endpoint

Parameters

- **url** (*str*) – Endpoint URL
- **http_method** (*str*) – The method used to make the request to the API
- **client_args** (*kwargs*) – Arguments to be sent to the auth client

Returns

Raise CartoException

Python Module Index

C

Index

D

E

F

G

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W