carto-python Documentation

Release 1.0.0

CARTO

Jun 09, 2017

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

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

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.

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.

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*

Exceptions

All the Exceptions of the CARTO Python client are wrapped into the *CartoException* class.

Please refer to the CARTO API docs 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.

CHAPTER 5

SQL API

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

```
from carto.sql import SQLCLient

sql = SQLCLient(auth_client)

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

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

sql = SQLCLient(auth_client)

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

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.

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)
```

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

CHAPTER 6

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.

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

# 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()
```

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()
```

Get all the datasets

```
from carto.datasets import DatasetManager

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

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)
```

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.

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)
```

CHAPTER 7

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)
```

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)
```

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 8

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 9

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

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.

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!
```

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
```

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
```

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
```

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
```

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
```

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
```

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
```

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
```

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
```

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
```

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
```

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!
```

kill_query.py

Description: Kills a running query

Usage example:

```
python kill_query.py 999
```

Output:

```
Query killed
```

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
```

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'
}
```

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}
```

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)
```

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
+------------+---------------+----------------------+---------------------+-------
↪--------+
```

```
| 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           |
| update_the_geom_webmercator_trigger |
```

```
+-----------------------------------+
```

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

CHAPTER 10

carto-python API docs

carto package API documentation

carto.auth module

Module for authenticated access to CARTO's APIs

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

get_user_name(*base_url*)

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

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

Dataset.**active_child = None**

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

Dataset.**attributions = None**

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

Dataset.**children = None**

Dataset.**connector = None**

Dataset.**created_at**
Field to store datetimes in resources

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

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

Dataset.**external_source = None**

Dataset.**fields = ['liked', 'likes', 'active_layer_id', 'table', 'display_name', 'privacy', 'permission', 'id', 'parent_id',**

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

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

Dataset.**license = None**

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

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

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

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

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

Dataset.**next_id** = None

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

Dataset.**permission**
carto.permissions.Permission

Dataset.**prev_id** = None

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

Dataset.**source** = None

Dataset.**stats**
Field to store datetimes in resources

Dataset.**synchronization** = None

Dataset.**table**
carto.tables.Table

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

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

Dataset.**transition_options** = None

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

Dataset.**updated_at**
Field to store datetimes in resources

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

Dataset.**user**
carto.users.User

Dataset.**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 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

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

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

ExportJob.**created_at**
Field to store datetimes in resources

ExportJob.**fields = ['user_id', 'url', 'created_at', 'updated_at', 'state', 'visualization_id', 'id']**

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

ExportJob.**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

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

ExportJob.**updated_at**
Field to store datetimes in resources

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

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

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

carto.fields module

Module for defining response objects

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

carto.permissions.Entity

value_class = 'carto.permissions.Entity'

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

carto.permissions.Permission

value_class = 'carto.permissions.Permission'

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

carto.tables.Table

value_class = 'carto.tables.Table'

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'

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

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

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

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

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

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

FileImportJob.**fields = ['quoted_fields_guessing', 'data_type', 'queue_id', 'user_defined_limits', 'item_queue_id',**

FileImportJob.**get_error_text = None**

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

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

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

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

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

FileImportJob.**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

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

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

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

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

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

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

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

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

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

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

FileImportJob.**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 CartoPaginator

resource_class
alias of FileImportJob

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

AnonymousMap.**fields = []**

AnonymousMap.**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

AnonymousMap.**update_from_dict**(*attribute_dict*)

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

NamedMap.**fields = []**

NamedMap.**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

NamedMap.**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*

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*)

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

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 = []

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

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.**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*)
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

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

Return type object

Raise CartoException

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

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

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

SyncTableJob.**created_at**
Field to store datetimes in resources

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

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

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

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

SyncTableJob.**fields = ['interval', 'service_name', 'updated_at', 'run_at', 'type_guessing', 'service_item_id', 'id', '**

SyncTableJob.**force_sync**()
Forces to sync the SyncTableJob

Returns

Raise CartoException

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

SyncTableJob.**get_force_sync_endpoint**()
Get the relative path to the specific API resource

Returns Relative path to the resource

Raise CartoException

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

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

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

SyncTableJob.**modified_at**
Field to store datetimes in resources

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

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

SyncTableJob.**ran_at**
Field to store datetimes in resources

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

SyncTableJob.**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

SyncTableJob.**run_at**
Field to store datetimes in resources

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

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

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

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

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

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

SyncTableJob.**updated_at**
Field to store datetimes in resources

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

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

SyncTableJob.**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 CartoPaginator

resource_class
alias of *SyncTableJob*

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

Table.**dependent_visualizations = None**

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

Table.**fields = ['rows_counted', 'map_id', 'description', 'permission', 'geometry_types', 'updated_at', 'table_size', 'i**

Table.**geometry_types**
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.**map_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

Table.**non_dependent_visualizations** = None

Table.**permission**
carto.permissions.Permission

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

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

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

Table.**schema** = None

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

Table.**synchronization** = None

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

Table.**table_visualization**
carto.visualizations.Visualization

Table.**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 CartoPaginator

resource_class
alias of *Table*

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

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

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

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

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

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

User.**fields = ['username', 'avatar_url', 'table_count', 'public_visualization_count', 'soft_geocoding_limit', 'all_visua**

User.**get_collection_endpoint**()

User.**get_resource_endpoint**()

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

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

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

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

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

User.**username**
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 CartoPaginator

resource_class
alias of *User*

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

Visualization.**active_child = None**

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

Visualization.**attributions = None**

Visualization.**children = None**

Visualization.**created_at**
Field to store datetimes in resources

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

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

Visualization.**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*

Visualization.**external_source** = None

Visualization.**fields** = ['liked', 'likes', 'active_layer_id', 'table', 'id', 'display_name', 'map_id', 'description', 'upd

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

Visualization.**kind** = None

Visualization.**license** = None

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

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

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

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

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

Visualization.**next_id** = None

Visualization.**parent_id** = None

Visualization.**permission** = None

Visualization.**prev_id** = None

Visualization.**privacy** = None

Visualization.**related_tables**
carto.tables.Table

Visualization.**source** = None

Visualization.**stats** = None

Visualization.**synchronization** = None

Visualization.**table**
carto.tables.Table

Visualization.**tags** = None

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

Visualization.**transition_options** = None

Visualization.**type** = None

Visualization.**updated_at**
Field to store datetimes in resources

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

Visualization.**uses_builder_features = None**

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 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