Onboard Python API documentation

Release 1.0

Ted Alexander

Apr 29, 2022

CONTENTS

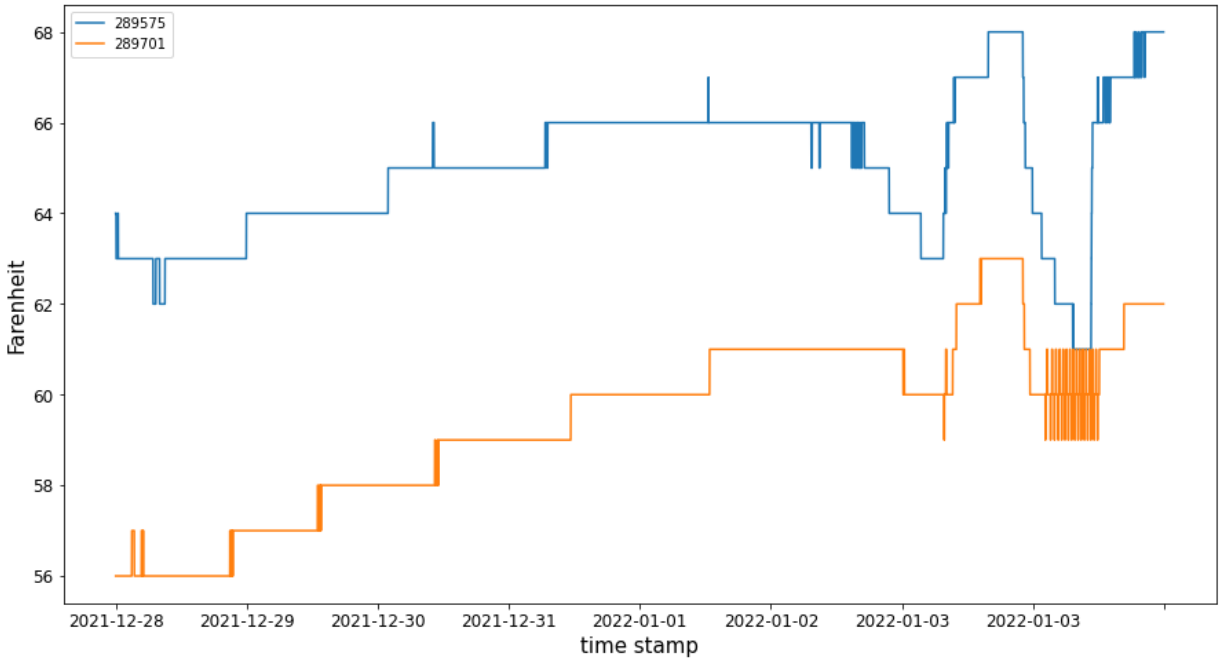
This package provides Python bindings to Onboard Data's building data API, allowing easy and lightweight access to building data.

For example, we can retrieve the last week of temperature data from all Zone Temperature points associated with FCUs in the Laboratory building:

```
import pandas as pd
from onboard.client import OnboardClient
from onboard.client.dataframes import points_df_from_streaming_timeseries
from onboard.client.models import PointSelector, TimeseriesQuery, PointData
from datetime import datetime, timezone, timedelta
from typing import List
import pytz
client = OnboardClient(api_key='your-api-key-here')

print(list(pd.DataFrame(client.get_all_buildings())['name'])) # returns list of␣
↪buildings that you have access to (you may not have 'Laboratory' in your set)

query = PointSelector()
query.point_types     = ['Zone Temperature'] # can list multiple point
query.equipment_types = ['fcu']              # types, equipment types,
query.buildings       = ['Laboratory']       # buildings, etc.
selection = client.select_points(query)

start = datetime.now(pytz.utc) - timedelta(days=7)
end   = datetime.now(pytz.utc)

timeseries_query = TimeseriesQuery(point_ids = selection['points'], start = start, end =␣
↪end)
sensor_data = points_df_from_streaming_timeseries(client.stream_point_
↪timeseries(timeseries_query))
```

and to plot:

```
import matplotlib.pyplot as plt
import numpy as np
# set the timestamp as the index and forward fill the data for plotting
sensor_data_clean = sensor_data.set_index('timestamp').astype(float).ffill()
# Edit the indexes just for visualization purposes
indexes = [i.split('T')[0] for i in list(sensor_data_clean.index)]
sensor_data_clean.index = indexes
fig = sensor_data_clean.plot(figsize=(15,8), fontsize = 12)
fig.set_ylabel('Farenheit',fontdict={'fontsize':15})
fig.set_xlabel('time stamp',fontdict={'fontsize':15})
plt.show()
```



For installation instructions, and to get set up with API access, refer to Initial Setup.

Important note for RTEM Hackathon entrants: the above sample won't work out of the box, because you need to use the RTEM version of the client (see Initial Setup), and because you have access to different buildings. Please see our Kaggle series to get started with the RTEM data!

Note: While we are committed to backwards-compatibility, this project is under active development. If you discover a feature that would be helpful, or any unexpected behavior, please contact us at support@onboarddata.io. If you are participating in the NYSERDA RTEM Hackathon and have any questions regarding the use of the API or the competition, please email us at support@rtemhackathon.com.

CHAPTER

ONE

CONTENTS

1.1 Initial Setup

1.1.1 Installation

The Python Onboard API client is available to install through pip:

```
>>> pip install onboard.client
```

or by cloning our Github repo:

```
git clone git@github.com:onboard-data/client-py
```

Please note, the client requires Python >= 3.7.

1.1.2 Setting up API access

You'll need an active API Key with the appropriate scopes in order to use this python client.

If you are an existing Onboard user you can head over to the accounts page and generate a new key and grant scopes for "general" and "buildings:read".

If you would like to get access to Onboard and start prototyping against an example building please request access here.

If you're participating in the NYSERDA hackathon, please check your confirmation email after signing up for API access instructions.

You can test if your API key is working with the following code:

```
>>> from onboard.client import OnboardClient
>>> client = OnboardClient(api_key='your-api-key-here')
>>>
>>> # Verify access & connectivity
>>> client.whoami()
{'result': 'ok', 'apiKeyInHeader': True, ... 'authLevel': 4}
```

You can also retrieve a list of your currently authorized scopes with `client.whoami()['apiKeyScopes']`.

NYSERDA hackathon participants will use an alternate client, which is identical except for some minor differences in available columns:

```
>>> from onboard.client import RtemClient
>>> client = RtemClient(api_key='your-api-key-here')
```

1.1.3 Note about data structure

By default, calls to the API (such as `client.get_equipment_types()`) return JSON objects. In this documentation, we will be converting these objects to pandas dataframes by wrapping our API calls in pd.DataFrame().

1.2 Querying Data Model

Onboard's data model contains both equipment types (e.g. fans, air handling units) and point types (e.g. zone temperature). We can query the full data model within our API.

Data model column definitions for each of the below tables can be found in *data model columns page*.

1.2.1 Equipment types

First, we make an API call with client.get_equipment_types(). This returns a JSON object, which we will convert to a dataframe using Pandas:

```
>>> from onboard.client import OnboardClient
>>> client = OnboardClient(api_key='')
>>> import pandas as pd
>>> # Get all equipment types from the Data Model
>>> equip_type = pd.json_normalize(client.get_equipment_types())
>>> equip_type.columns
['id', 'tag_name', 'name_long', 'name_abbr', 'active', 'flow_order',
'critical_point_types', 'sub_types', 'tags']
```

equip_type now contains a dataframe listing all equipment types in our data model, along with associated attributes (e.g. tags, full names, associated point types, and sub-equipment types).

The sub-equipment types are nested as dataframes within each row, and can be listed for an equipment type (e.g. 'fan') like so:

```
>>> sub_type = pd.DataFrame(equip_type[equip_type.tag_name == 'fan']['sub_types'].item())
   id  equipment_type_id         tag_name          name_long name_abbr
0  12                 26       exhaustFan        Exhaust Fan       EFN
1  13                 26        reliefFan         Relief Fan      RlFN
2  14                 26        returnFan         Return Fan       RFN
3  15                 26        supplyFan         Supply Fan       SFN
...
```

Note that not all equipment types have associated sub-types.

1.2.2 Point types

Accessing point types is very similar, and can be accessed through client.get_all_point_types():

```
>>> # Get all point types from the Data Model
>>> point_type = pd.DataFrame(client.get_all_point_types())
>>> point_type[['id', 'tag_name', 'tags']]
      id                                  tag_name                                ␣
↪     tags
```

(continues on next page)

(continued from previous page)

```
0    124                Occupied Heating Setpoint             [air, sp, temp, zone,␣
↪heating, occ]
1    118               Outside Air Carbon Dioxide                    [air, co2, sensor,
↪ outside]
2    130          Return Air Temperature Setpoint                        [air, sp,␣
↪temp, return]
3     84  Dual-Temp Coil Discharge Air Temperature  [air, discharge, dualTemp, sensor,␣
↪temp, coil]
...
```

point_type now contains a dataframe listing all the tags associated with each point type.

We can extract the metadata associated with each tag in our data model like so:

```
>>> # Get all tags and their definitions from the Data Model
>>> pd.DataFrame(client.get_tags())
      id        name                                          definition def_source      ␣
↪                                  def_url
0    120     battery  A container that stores chemical energy that c...      brick ␣
↪https://brickschema.org/ontology/1.1/classes/B...
1    191  exhaustVAV  A device that regulates the volume of air bein...    onboard      ␣
↪                                       None
2    193         oil  A viscous liquid derived from petroleum, espec...      brick ␣
↪https://brickschema.org/ontology/1.2/classes/Oil/
3    114    fumeHood  A fume-collection device mounted over a work s...      brick ␣
↪https://brickschema.org/ontology/1.1/classes/F...
...
```

This returns a dataframe containing definitions for all tags in our data model, with attribution where applicable.

1.2.3 Unit types

```
>>> # Get all unit types from the Data Model
>>> unit_types = pd.DataFrame(client.get_all_units())
>>> unit_types[['id', 'name_long', 'qudt']]
   id             name_long                                  qudt
0  55                 Litre          http://qudt.org/vocab/unit/L
1  68             US Gallon     http://qudt.org/vocab/unit/GAL_US
2  75                   Bar        http://qudt.org/vocab/unit/BAR
3  76                 Watts          http://qudt.org/vocab/unit/W
...
```

1.2.4 Measurement types

```
>>> # Get all measurement types from the Data Model
>>> measurement_types = pd.DataFrame(client.get_all_measurements())
>>> measurement_types[['id', 'name', 'qudt_type']]
    id              name                                          qudt_type
0   20     Reactive Power   http://qudt.org/vocab/quantitykind/ReactivePower
1   27              Floor   http://qudt.org/vocab/quantitykind/Dimensionless
2   33       Power Factor   http://qudt.org/vocab/quantitykind/Dimensionless
3   31             Torque  http://qudt.org/vocab/quantitykind/Dimensionle...
...
```

1.3 Querying Building-Specific Data

For column definitions, see *Columns for Data Extracted from Buildings*.

1.3.1 Querying Equipment

Using the API, we can retrieve the data from all the buildings that belong to our organization:

```
>>> # Get a list of all the buildings under your Organization
>>> pd.json_normalize(client.get_all_buildings())
    id  org_id             name  ... point_count info.note info
0   66       6   T`Challa House  ...          81            NaN
1  427       6  Office Building  ...        4219       NaN  NaN
2  428       6       Laboratory  ...        2206       NaN  NaN
3  429       6         Hogwarts  ...        4394       NaN  NaN
```

The first column of this dataframe ('id') contains the building identifier number.

In order to retrieve the equipment for a particular building (e.g. Laboratory, id: 428), we use client.get_building_equipment():

```
>>> # Get a list of all equipment in a building
>>> all_equipment = pd.DataFrame(client.get_building_equipment(428))
>>> all_equipment[['id', 'building_id', 'equip_id',  'points', 'tags']]
        id  building_id        equip_id                                          ␣
↪points                     tags
0    27293          428      crac-T-105  [{'id': 291731, 'building_id': 428, 'last_upda..
↪.             [crac, hvac]
1    27294          428   exhaustFan-01  [{'id': 290783, 'building_id': 428, 'last_upda..
↪.  [fan, hvac, exhaustFan]
2    27295          428  exhaustFan-021  [{'id': 289684, 'building_id': 428, 'last_upda..
↪.  [fan, hvac, exhaustFan]
3    27296          428  exhaustFan-022  [{'id': 289655, 'building_id': 428, 'last_upda..
↪.  [fan, hvac, exhaustFan]
...
```

1.3.2 Querying Specific Points

In order to query specific points, first we need to import the PointSelector class:

```
>>> # Set parameters for querying sensor data
>>> from onboard.client.models import PointSelector
>>> query = PointSelector()
```

There are multiple ways to select points using the PointSelector. The user can select all the points that are associated with one or more lists containing any of the following:

```
'organizations', 'buildings', 'point_ids', 'point_names', 'point_hashes',
'point_ids', 'point_names', 'point_topics', 'equipment', 'equipment_types'
```

For example, here we make a query that returns all the points of the type 'Real Power' OR of the type 'Zone Temperature' that belong to the 'Laboratory' building:

```
>>> query = PointSelector()
>>> query.point_types = ['Real Power', 'Zone Temperature']
>>> query.buildings = ['Laboratory']
>>> selection = client.select_points(query)
```

We can add to our query to e.g. further require that returned points must be associated with the 'fcu' equipment type:

```
>>> query = PointSelector()
>>> query.point_types = ['Real Power', 'Zone Temperature']
>>> query.equipment_types = ['fcu']
>>> query.buildings = ['Laboratory']
>>> selection = client.select_points(query)
>>> selection
{'buildings': [428],
'equipment': [27356, 27357],
'equipment_types': [9],
'orgs': [6],
'point_types': [77],
'points': [289701, 289575]}
```

In this example, the points with ID=162801, and 162795 are the only ones that satisfy the requirements of our query.

We can get more information about these points by calling the function get_points_by_ids() on selection['points']:

```
>>> # Get Metadata for the sensors you would like to query
>>> sensor_metadata = client.get_points_by_ids(selection['points'])
>>> sensor_metadata_df = pd.DataFrame(sensor_metadata)
>>> sensor_metadata_df[['id', 'building_id', 'first_updated', 'last_updated', 'type',
↪'value', 'units']]
       id  building_id  first_updated  last_updated              type value            ␣
↪units
0  289575          428   1.626901e+12  1.641928e+12  Zone Temperature  66.0 ␣
↪degreesFahrenheit
1  289701          428   1.626901e+12  1.641928e+12  Zone Temperature  61.0 ␣
↪degreesFahrenheit
```

sensor_metadata_df now contains a dataframe with rows for each point. Based on the information about these points, we can observe that none of the points of our list belongs to the point type 'Real Power', but only to the point type 'Zone Temperature'

1.3.3 Exporting Data to .csv

Data extracted using the API can be exported to a .csv or excel file using Pandas:

```
>>> # Save Metadata to .csv file
>>> sensor_metadata_df.to_csv('~/metadata_query.csv')
```

1.3.4 Querying Time-Series Data

To query time-series data first we need to import modules from datetime, models and dataframes.

```
>>> from datetime import datetime, timezone, timedelta
>>> import pytz
>>> from onboard.client.models import TimeseriesQuery, PointData
>>> from onboard.client.dataframes import points_df_from_streaming_timeseries
```

We select the range of dates we want to query, in UTC format:

```
>>> # Enter Start & End Time Stamps in UTC
>>> # Example "2018-06-03T12:00:00Z"
>>>
>>> # get data from the past week
>>> start = datetime.now(pytz.utc) - timedelta(days=7)
>>> end = datetime.now(pytz.utc)
```

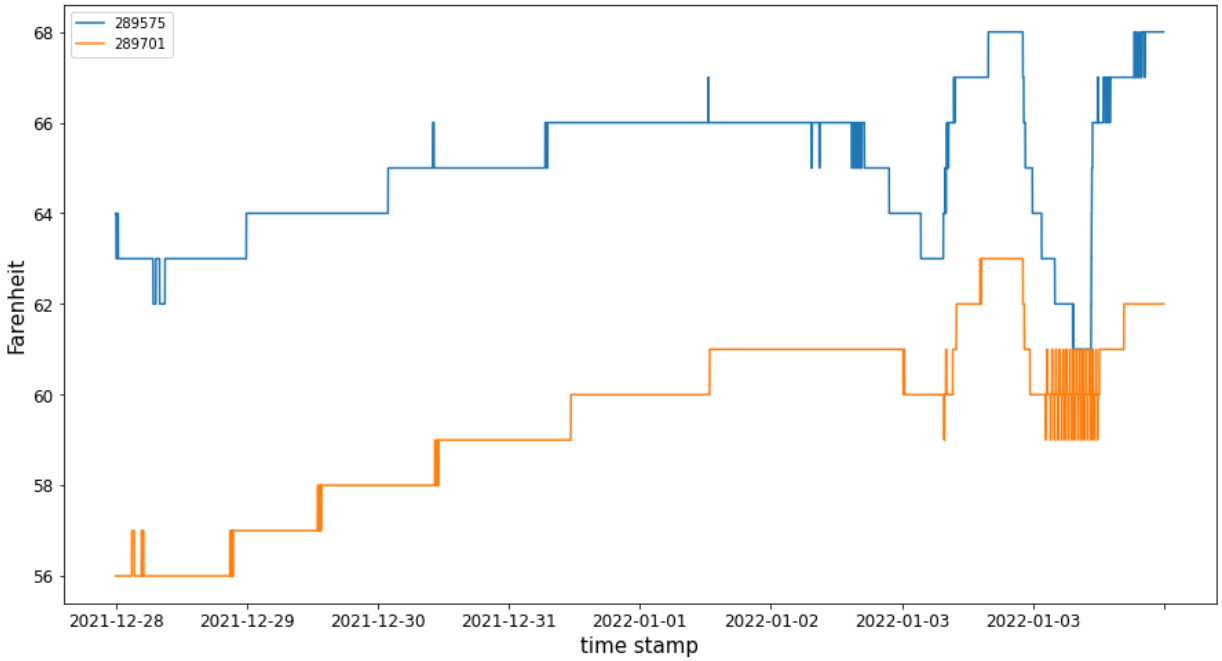
Now we are ready to query the time-series data for the points we previously selected in the specified time-period

```
>>> # Get time series data for the sensors you would like to query
>>> timeseries_query = TimeseriesQuery(point_ids = selection['points'], start = start,␣
↪end = end)
>>> sensor_data = points_df_from_streaming_timeseries(client.stream_point_
↪timeseries(timeseries_query))
>>> sensor_data
                       timestamp 289575 289701
0      2022-01-04T19:34:11.741000Z   68.0   None
1      2022-01-04T19:34:19.143000Z   None   62.0
2      2022-01-04T19:35:12.133000Z   68.0   None
...
```

This returns a dataframe containing columns for the timestamp and for each requested point.

Here, we set the timestamp as the index and forward fill the data for plotting

```
>>> sensor_data_clean = sensor_data.set_index('timestamp').astype(float).ffill()
>>>
>>> # Edit the indexes just for visualization purposes
>>> indexes = [i.split('T')[0] for i in list(sensor_data_clean.index)]
>>> sensor_data_clean.index = indexes
>>>
>>> fig = sensor_data_clean.plot(figsize=(15,8), fontsize = 12)
>>>
>>> # Adding some formatting
>>> fig.set_ylabel('Farenheit',fontdict={'fontsize':15})
>>> fig.set_xlabel('time stamp',fontdict={'fontsize':15})
```



1.4 Columns for Data Model

1.4.1 Equipment types

Accessed with `pd.json_normalize(client.get_equipment_types())`

id: unique integer associated with the given type/tag

tag_name: this is the equip type tag associated with a class

name_long: longer, human-readable name (e.g. tag "cogen" => "Cogeneration Plant"). This is the class name you will find in the ontology

name_abbr: common abbreviated form (e.g. "FCU", "CHWS")

active: True if this class in the latest version of the ontology

critical_point_types: id numbers of the associated point types that are expected to be observed (look up in client.get_all_point_types())

sub_types: embedded JSON of possible forms of the equipment super-type (e.g. 'fan' has the sub-types 'exhaustFan', 'reliefFan', 'returnFan', etc.)

tags: Haystack tags associated with equipment super-type

1.4.2 Sub-equipment types

Accessed for given equipment (e.g. 'fan') with `sub_type = pd.DataFrame(equip_type[equip_type.tag_name == 'fan']['sub_types'].item())`

id: unique integer associated with the given type/tag

equipment_type_id: id of the associated equipment tag in client.get_equipment_types()

tag_name: this is the sub-equip type tag associated with a class

name_long: longer, human-readable name. This is the class name you will find in the ontology.

name_abbr: common abbreviated form

1.4.3 Point types

Accessed with `client.get_all_point_types()`

id: unique integer associated with the given type/tag

tag_name: human-readable name. This is the class name you will find in the ontology.

active: True if this class in the latest version of the ontology

measurement_id: id of the associated measurement type in `client.get_all_measurements()`

tags: Haystack tags associated with point type

1.4.4 Unit types

Accessed with `pd.DataFrame(client.get_all_units())`

id: unique integer associated with the given type/tag

name_long: human-readable unit name (e.g. 'Cubic Meter per Hour')

name_abbr: abbreviated form (e.g. 'm3/h')

data_type: form of associated data. Can be 'Binary', 'Continuous', 'Enum', 'None', or 'Ordinal'

raw_encoding: for Binary and Enum data types, contains dictionary matching number to interpretation.

display_encoding: for Binary and Enum data types, contains dictionary showing how each reported number will be displayed. E.g., a 0 from an Occupancy sensor will be reported as 'Unoccupied'.

qudt: url for additional information about unit (e.g. 'Degrees Celsius') on qudt.org

unit_type: url for additional information about measurement type (e.g. 'Temperature') on qudt.org

1.4.5 Measurement types

Accessed with `pd.DataFrame(client.get_all_measurements())`

id: unique integer associated with the given measurement types

name: name of measurement type

default_unit_id: id of default associated unit type in client.get_all_units(). Note, pandas will cast this column as a float, but it can still be used to look up id

units_convertible: True if units of this measurement type can be interchangeably converted (generally True for continuous measurement types)

units: embedded JSON of possible units for given measurement type

qudt_type: url for additional information about measurement type (e.g. 'Temperature') on qudt.org

1.4.6 Tag metadata

Accessed with `pd.DataFrame(client.get_tags())`

id: unique integer associated with the given tag metadata

name: name of tag being described

definition: definition of tag

def_source: source of definition (either brick, haystack, or onboard)

def_url: url for source of definition (brick and haystack only)

category: category used to help sort point types in the ontology (see data model page). Can be 'Medium', 'Medium Property', 'Point Class', 'Quantity Modifier', or None

1.5 Columns for Data Extracted from Buildings

1.5.1 Building-Specific Equipment

id: unique integer associated with the given equipment in this building. Will be unique across all equipment in platform.

building_id: unique integer associated with the building. Will be unique across all buildings in platform.

equip_id: Name to identify individual equipment instances. Constructed as equipment name + identifying suffix

suffix: Just the identifying suffix part of the equip_id

equip_type_name: Relevant name in the ontology

equip_type_id: integer id of relevant equipment type

equip_type_abbr: abbreviation of relevant equipment type

equip_type_tag: tag name of relevant equipment type

equip_subtype_name: name of relevant equipment sub-type

equip_subtype_id: integer id of relevant equipment sub-type

equip_subtype_tag: tag name of relevant equipment sub-type

floor_num_physical: 4-digit code (see below) for floor where equipment is located. Can be integer or NaN if not available

1000: basement

1001: rooftop

1002: outside

1003: whole_buildings

1004: ground_floor

1005: penthouse

floor_num_served: 4-digit code for floor that equipment serves. Can be integer or NaN if not available

area_served_desc: Description of area that equipment serves

equip_dis: plain-text description of equipment from building documentation

parent_equip: integer id that links to parent equipment row(s)

child_equip: integer id that links to child equipment row(s)

points: embedded JSON containing associated points

tags: Haystack tags associated with equipment

1.5.2 Building-Specific Points

id: unique integer associated with the given point in this building. Will be unique across all points in platform.

building_id: unique integer associated with the building. Will be unique across all buildings in platform.

last_updated: Unix-formatted timestamp of most recent value reported from point

first_updated: Unix-formatted timestamp of earliest value reported from point

name: raw sensor metadata (from BACnet scan)

description: alternate location for raw sensor metadata (from BACnet scan)

units: Matches to unit abbreviation in units table

raw_unit_id: unit id as it appears in `client.get_all_units()`

value: Most recent reported value for point (from BACnet scan)

type: name of point type in the ontology

point_type_id: point type name as it appears in `client.get_all_point_types()`

measurement_id: measurement type id as it appears in `client.get_all_measurements())`

state_text: mapping between each state and text description of state

equip_id: unique integer associated with the associated equipment

1.5.3 Site-Level Data

Accessed with `client.get_all_buildings()`

Note: many fields will be blank for NYSERDA hackathon users.

id: Unique ID generated for a new site (primary key for the Site Table)

name: Site name (will be a number for NYSERDA hackathon users)

sq_ft: Total square-footage of the site address

equip_count: Number of equipment instances associated with the building

point_count: Number of points associated with the building

info.org: Site's main ownership organization

info.floors: Number of floors associated with the site's square footage

info.m2fend: Site scheduled weekday closing time

info.satend: Site scheduled Saturday closing time

info.sunend: Site scheduled Sunday closing time

info.geoCity: Name of the city where the site is located

info.geoState: Name of the state where the site is located (e.g. New York)

info.m2fstart: Site scheduled weekday opening time

info.satstart: Site scheduled Saturday opening time

info.sunstart: Site scheduled Sunday opening time

info.geoCountry: Name of the country where the site is located

info.weatherRef: The source of weather data

CHAPTER

TWO

LICENSE