# api2db

*Release 0.9.9*

**Tristen Harr**

**Jun 02, 2021**

# CONTENTS

```
pip install api2db
```

CHAPTER

# ONE

# BECAUSE **** MESSY DATA AND SCOPE CREEP

**Note: Project requirements changed? No problem.**

**API got updated? Easy.**

**Changing databases? Change 1 line of code.**

**Multiple storage targets? Done.**

**Stop focusing on collecting data, and start focusing on using it.**

**Use api2db to:**

- Collect data from **any** API
- Perform pre-processing on data
- Clean and extract features from data
- Perform post-processing on data
- Store data

**api2db supports any/all of the following concurrently:**

- Streaming data live to
  - Local storage of data in files using Parquet, pickle, JSON, and CSV format
  - MySQL
  - Bigquery
  - MariaDB
  - PostgreSQL
  - Omnisci
- Storing data periodically to
  - MySQL
  - Bigquery
  - MariaDB
  - PostgreSQL
  - Omnisci

**api2db is currently adding support for:**

- Oracle

- Amazon Aurora
- Microsoft SQL Server
- Firebase RTDB
- Don't see your database? Submit a feature request.. or even better add it. api2db is open-source.

CHAPTER

TWO

# CONTENTS

## 2.1 Installation and Quickstart

### 2.1.1 Installation

**Install the library**

```
pip install api2db
```

**To add MySQL support**

```
pip install mypysql
```

**To add MariaDB support**

```
pip install mariadb
```

**To add PostgreSQL support**

```
pip install psycopg2
```

**To add Omnisci support**

```
pip install pymapd==0.25.0
pip install pyarrow==3.0.0
pip install pandas --upgrade
```

### 2.1.2 Quickstart

**Create a project with the `pmake` shell command**

Initial directory structure

```
project_dir/
```

```
path/to/project_dir/> pmake
```

New project directory structure

```
project_dir------/
                 |
                 apis------/                                    # Each API will get its␣
↪own file
                 |         |- __init__.py
                 |
```

(continues on next page)

(continued from previous page)

```
                AUTH-----/                                          # These templates are␣
↪used for adding database targets
                |        |- bigquery_auth_template.json
                |        |- omnisci_auth_template.json
                |        |- sql_auth_template.json
                |
                CACHE/                                              # Application cache files␣
↪will be stored and used here.
                |
                STORE/                                              # This is where incoming␣
↪data can be stored locally.
                |
                LOGS/                                               # Each collector will␣
↪receive it's own log file.
                |
                helpers.py                                          # Common helper functions␣
↪can be written here.
                |
                main.py                                             # This is the application␣
↪entry point.
```

**Choose an API** This example will use the CoinCap API as it is free, does not require an API key, and seems to have good uptime. (This project has no affiliation with CoinCap)

**Create a collector with the `cadd` shell command**

```
path/to/project_dir/> cadd coincap
```

```
project_dir-----/
                |
                apis-----/
                |        |- __init__.py
                |        |- coincap.py                             # This is where you'll␣
↪write code!
                |
                AUTH-----/
                |        |- bigquery_auth_template.json
                |        |- omnisci_auth_template.json
                |        |- sql_auth_template.json
                |
                CACHE/
                |
                STORE/
                |
                LOGS/
                |
                helpers.py
                |
                main.py
```

**Understanding `project_dir/apis/some_api_collector.py`**

Each collector has 4 parts.

1. **Data Import**
    - Performs a call to an API to request data.
2. **Data Processing**

- Processes and cleans incoming data, making it useful.

3. **Data Streams**
    - Streams data as it arrives to a storage target.
4. **Data Stores**
    - Stores data periodically to a storage target.

**The base template for the coincap collector looks like this**

```
from api2db.ingest import *
from api2db.stream import *
from api2db.store import *
from helpers import *

def coincap_import():
    return None

def coincap_form():
    pre_process = [
        # Preprocessing Here
    ]

    data_features = [
        # Data Features Here
    ]

    post_process = [
        # Postproccesing Here
    ]

    return ApiForm(name="coincap", pre_process=pre_process, data_features=data_
↪features, post_process=post_process)

def coincap_streams():
    streams = [

    ]
    return streams

def coincap_stores():
    stores = [

    ]
    return stores

coincap_info = Collector(name="coincap",
                         seconds=0,                         # Import frequency of 0␣
↪disables collector
                         import_target=coincap_import,
                         api_form=coincap_form,
                         streams=coincap_streams,
                         stores=coincap_stores,
                         debug=True                         # Set to False for production
                         )
```

### 2.1.3 Using the lab to build ApiForms

**To simplify setting up the data import and data-processing first run the `mlab` shell command**

```
path/to/project_dir/> mlab
```

```
project_dir-----/
                |
                apis------/
                |          |- __init__.py
                |          |- coincap.py
                |
                AUTH------/
                |          |- bigquery_auth_template.json
                |          |- omnisci_auth_template.json
                |          |- sql_auth_template.json
                |
                CACHE/
                |
                laboratory/
                |          |- lab.py      # This is where you can experiment with pre-
↪processing
                |
                STORE/
                |
                LOGS/
                |
                helpers.py
                |
                main.py
```

**A blank lab.py file will look like this**

```
from api2db.ingest import *
CACHE=True      # Caches API data so that only a single API call is made if True

def import_target():
    return None

def pre_process():
    return None

def data_features():
    return None

def post_process():
    return None

if __name__ == "__main__":
    api_form = ApiForm(name="lab",
                       pre_process=pre_process(),
                       data_features=data_features(),
                       post_process=post_process()
                       )
    api_form.experiment(CACHE, import_target)
```

### 2.1.4 Importing data

**Perform a data import by writing the code for the `import_target` function**

lab.py

```
.
.
.

import requests
import logging

def import_target():
    """
    Data returned by the import target must be an array of dicts.
    This allows for either a single API call to be returned, or an array of them.
    """
    data = None
    url = "https://api.coincap.io/v2/assets/"
    try:
        data = [requests.get(url).json()]
    except Exception as e:
        logging.exception(e)
    return data

.
.
.
```

**Use the `rlab` shell command to run the lab**

---

**Note:** Watch the laboratory directory closely. Data will be dumped into JSON files at different points during data-processing to provide the programmer with an easier to read format.

---

```
path/to/project_dir/> rlab
```

Output:

```
data:
{
    "data": [
        {
            "id": "bitcoin",
            "rank": "1",
            "symbol": "BTC",
            "name": "Bitcoin",
            "supply": "18698850.0000000000000000",
            "maxSupply": "21000000.0000000000000000",
            "marketCapUsd": "1041388865130.8623213956691350",
            "volumeUsd24Hr": "12822561919.6746830356589619",
            "priceUsd": "55692.6690748822693051",
            "changePercent24Hr": "-4.1033665252363403",
            "vwap24Hr": "57708.7312639442977184",
            "explorer": "https://blockchain.info/",
        },
        .
```

(continues on next page)

(continued from previous page)

```
        .
        .
    ],

    "timestamp": 1620100433183,
}

data keys:
dict_keys(['data', 'timestamp'])

pre_process must return a list of 0 or more pre-processors.
pre_process:
None
```

### 2.1.5 Performing pre-processing on data

**Perform pre-processing on data by writing the code for the `pre_process` function**

lab.py

```
.
.
.

def pre_process():
    """
    Pre-processors are applied sequentially.
    In this example, we will:

        1. Extract the timestamp and make it a global feature using GlobalExtract
        2. Perform a ListExtract to extract the list of data which will become the␣
↪rows in the storage target table
    """
    return [
        GlobalExtract(key="timestamp",
                      lam=lambda x: x["timestamp"],
                      dtype=int
                      ),

        ListExtract(lam=lambda x: x["data"])
    ]


.
.
.
```

**Use the `rlab` shell command to run the lab**

```
path/to/project_dir/> rlab
```

Output:

```
data point 1:
{'id': 'bitcoin', 'rank': '1', 'symbol': 'BTC', 'name': 'Bitcoin', 'supply':
↪'18698850.0000000000000000', 'maxSupply': '21000000.0000000000000000', 'marketCapUsd
↪': '1
```

(continues on next page)

(continued from previous page)

```
041388865130.8623213956691350', 'volumeUsd24Hr': '12822561919.6746830356589619',
↪'priceUsd': '55692.6690748822693051', 'changePercent24Hr': '-4.1033665252363403',
↪'vw
ap24Hr': '57708.7312639442977184', 'explorer': 'https://blockchain.info/'}

data point 2:
{'id': 'ethereum', 'rank': '2', 'symbol': 'ETH', 'name': 'Ethereum', 'supply':
↪'115729464.3115000000000000', 'maxSupply': None, 'marketCapUsd': '376411190202.
↪66581272
13330461', 'volumeUsd24Hr': '17656637086.6618270054805080', 'priceUsd': '3252.
↪5095699873722881', 'changePercent24Hr': '6.4420494833790460', 'vwap24Hr': '3234.
↪41835079
37765772', 'explorer': 'https://etherscan.io/'}

data point 3:
{'id': 'binance-coin', 'rank': '3', 'symbol': 'BNB', 'name': 'Binance Coin', 'supply
↪': '153432897.0000000000000000', 'maxSupply': '170532785.0000000000000000', 'marke
tCapUsd': '98431624817.6777436959489247', 'volumeUsd24Hr': '254674805.8210425908376882
↪', 'priceUsd': '641.5288164550379551', 'changePercent24Hr': '1.1504585233985471'
, 'vwap24Hr': '653.0516845642682435', 'explorer': 'https://etherscan.io/token/
↪0xB8c77482e45F1F44dE1745F52C74426C631bDD52'}

data_features must return a list of data-features.
data_features:
None
```

### 2.1.6 Extracting features from data

**Extract data-features from data by writing the code for the `data_features` function**

---

**Note:** Pick and choose which data-features you wish to extract from your data. This example will extract the `id`, `rank`, `symbol`, `name`, `priceUsd`, and `volumeUsd24Hr`

Feature extraction will handle null data and data of the wrong type automatically.

---

lab.py

```
.
.
.

def data_features():
    return [
        Feature(key="id",
                lam=lambda x: x["id"],
                dtype=str),

        Feature(key="rank",
                lam=lambda x: x["rank"],
                dtype=int),

        Feature(key="symbol",
                lam=lambda x: x["symbol"],
                dtype=str),
```

(continues on next page)

(continued from previous page)

```
        Feature(key="name",
                lam=lambda x: x["name"],
                dtype=str),

        Feature(key="price_usd",                    # Keys support renaming
                lam=lambda x: x["priceUsd"],
                dtype=float),

        Feature(key="volume_usd_24_hr",
                lam=lambda x: x["volumeUsd24Hr"],
                dtype=float)
    ]

.
.
.
```

**Use the `rlab` shell command to run the lab**

```
path/to/project_dir/> rlab
```

Output:

```
data point 1:
{'id': 'bitcoin', 'rank': '1', 'symbol': 'BTC', 'name': 'Bitcoin', 'supply':
↪'18698850.0000000000000000', 'maxSupply': '21000000.0000000000000000', 'marketCapUsd
↪': '1
041388865130.8623213956691350', 'volumeUsd24Hr': '12822561919.6746830356589619',
↪'priceUsd': '55692.6690748822693051', 'changePercent24Hr': '-4.1033665252363403',
↪'vw
ap24Hr': '57708.7312639442977184', 'explorer': 'https://blockchain.info/'}

data point 2:
{'id': 'ethereum', 'rank': '2', 'symbol': 'ETH', 'name': 'Ethereum', 'supply':
↪'115729464.3115000000000000', 'maxSupply': None, 'marketCapUsd': '376411190202.
↪66581272
13330461', 'volumeUsd24Hr': '17656637086.6618270054805080', 'priceUsd': '3252.
↪5095699873722881', 'changePercent24Hr': '6.4420494833790460', 'vwap24Hr': '3234.
↪41835079
37765772', 'explorer': 'https://etherscan.io/'}

data point 3:
{'id': 'binance-coin', 'rank': '3', 'symbol': 'BNB', 'name': 'Binance Coin', 'supply
↪': '153432897.0000000000000000', 'maxSupply': '170532785.0000000000000000', 'marke
tCapUsd': '98431624817.6777436959489247', 'volumeUsd24Hr': '254674805.8210425908376882
↪', 'priceUsd': '641.5288164550379551', 'changePercent24Hr': '1.1504585233985471'
, 'vwap24Hr': '653.0516845642682435', 'explorer': 'https://etherscan.io/token/
↪0xB8c77482e45F1F44dE1745F52C74426C631bDD52'}

data:
                id  rank symbol            name     price_usd    volume_usd_24_hr
↪ timestamp
0          bitcoin     1    BTC         Bitcoin  55692.669075  12822561919.674683
↪1620100433183
1         ethereum     2    ETH        Ethereum    3252.50957  17656637086.661827
↪1620100433183
```

(continues on next page)

(continued from previous page)

```
2     binance-coin     3     BNB    Binance Coin    641.528816    254674805.821043 ␣
↪1620100433183
3              xrp     4     XRP             XRP      1.461734   1969092162.016667 ␣
↪1620100433183
4         dogecoin     5    DOGE        Dogecoin      0.419828   2694025432.110168 ␣
↪1620100433183
..             ...   ...     ...             ...           ...                ...    ␣
↪       ...
95       abbc-coin    96    ABBC       ABBC Coin      0.755244        355316.252287 ␣
↪1620100433183
96          status    97     SNT          Status      0.169848       5966843.243043 ␣
↪1620100433183
97             nxm    98     NXM             NXM     90.764252       7577199.874023 ␣
↪1620100433183
98  ocean-protocol    99   OCEAN  Ocean Protocol      1.357968       9131449.423728 ␣
↪1620100433183
99           iotex   100    IOTX           IoTeX      0.057802        576658.038699 ␣
↪1620100433183

[100 rows x 7 columns]

data dtypes:
id                   string
rank                  Int64
symbol               string
name                 string
price_usd           Float64
volume_usd_24_hr    Float64
timestamp             Int64
dtype: object
```

### 2.1.7 Performing post-processing on data

**Perform post-processing on data by writing the code for the `post_process` function**

**Note:** Post-processors can be applied to alter the data, or extract new information from the data.

lab.py

```
.
.
.

import time
def post_process():
    """
    In this example we will add a timestamp for the arrival time of the data.
    """
    return [
        ColumnAdd(key="arrival_time",
                  lam=lambda: int(time.time()*1000),
                  dtype=int
                  )
```

(continues on next page)

(continued from previous page)

```
    ]

.
.
.
```

**Use the `rlab` shell command to run the lab**

```
path/to/project_dir/> rlab
```

Output:

```
data point 1:
{'id': 'bitcoin', 'rank': '1', 'symbol': 'BTC', 'name': 'Bitcoin', 'supply':
↪'18698850.0000000000000000', 'maxSupply': '21000000.0000000000000000', 'marketCapUsd
↪': '1
041388865130.8623213956691350', 'volumeUsd24Hr': '12822561919.6746830356589619',
↪'priceUsd': '55692.6690748822693051', 'changePercent24Hr': '-4.1033665252363403',
↪'vw
ap24Hr': '57708.7312639442977184', 'explorer': 'https://blockchain.info/'}

data point 2:
{'id': 'ethereum', 'rank': '2', 'symbol': 'ETH', 'name': 'Ethereum', 'supply':
↪'115729464.3115000000000000', 'maxSupply': None, 'marketCapUsd': '376411190202.
↪66581272
13330461', 'volumeUsd24Hr': '17656637086.6618270054805080', 'priceUsd': '3252.
↪5095699873722881', 'changePercent24Hr': '6.4420494833790460', 'vwap24Hr': '3234.
↪41835079
37765772', 'explorer': 'https://etherscan.io/'}

data point 3:
{'id': 'binance-coin', 'rank': '3', 'symbol': 'BNB', 'name': 'Binance Coin', 'supply
↪': '153432897.0000000000000000', 'maxSupply': '170532785.0000000000000000', 'marke
tCapUsd': '98431624817.6777436959489247', 'volumeUsd24Hr': '254674805.8210425908376882
↪', 'priceUsd': '641.5288164550379551', 'changePercent24Hr': '1.1504585233985471'
, 'vwap24Hr': '653.0516845642682435', 'explorer': 'https://etherscan.io/token/
↪0xB8c77482e45F1F44dE1745F52C74426C631bDD52'}

finalized data:
                id  rank symbol            name     price_usd    volume_usd_24_hr
↪ timestamp   arrival_time
0          bitcoin     1    BTC         Bitcoin  55692.669075  12822561919.674683
↪1620100433183  1620104839526
1         ethereum     2    ETH        Ethereum    3252.50957  17656637086.661827
↪1620100433183  1620104839526
2     binance-coin     3    BNB    Binance Coin    641.528816    254674805.821043
↪1620100433183  1620104839526
3              xrp     4    XRP             XRP      1.461734   1969092162.016667
↪1620100433183  1620104839526
4         dogecoin     5   DOGE        Dogecoin      0.419828   2694025432.110168
↪1620100433183  1620104839526
..             ...   ...    ...             ...           ...                ...
↪       ...             ...
95       abbc-coin    96   ABBC       ABBC Coin      0.755244       355316.252287
↪1620100433183  1620104839526
96          status    97    SNT          Status      0.169848      5966843.243043
↪1620100433183  1620104839526
```

(continues on next page)

(continued from previous page)

```
97             nxm     98     NXM              NXM      90.764252       7577199.874023 ␣
↪1620100433183  1620104839526
98  ocean-protocol     99  OCEAN  Ocean Protocol       1.357968       9131449.423728 ␣
↪1620100433183  1620104839526
99           iotex    100    IOTX           IoTeX       0.057802        576658.038699 ␣
↪1620100433183  1620104839526

[100 rows x 8 columns]

finalized data dtypes:
id                   string
rank                  Int64
symbol               string
name                 string
price_usd           Float64
volume_usd_24_hr    Float64
timestamp             Int64
arrival_time          Int64
dtype: object
```

### 2.1.8 Exporting data from the lab to a collector

**Note: Once the lab has been used to build the form fields for an ApiForm, move the data to the collector**

It is not necessary to use the lab feature of the library to perform data-extraction, it just makes things a bit easier.

**Move the code from `lab.py` to `coincap.py`**

coincap.py

```
.
.
.

import requests
import logging
import time


def coincap_import():
    data = None
    url = "https://api.coincap.io/v2/assets/"
    try:
        data = [requests.get(url).json()]
    except Exception as e:
        logging.exception(e)
    return data

def coincap_form():
    pre_process = [
        GlobalExtract(key="timestamp",
                      lam=lambda x: x["timestamp"],
                      dtype=int
                      ),
```

(continues on next page)

(continued from previous page)

```
        ListExtract(lam=lambda x: x["data"])
    ]

    data_features = [
        Feature(key="id",
                lam=lambda x: x["id"],
                dtype=str),

        Feature(key="rank",
                lam=lambda x: x["rank"],
                dtype=int),

        Feature(key="symbol",
                lam=lambda x: x["symbol"],
                dtype=str),

        Feature(key="name",
                lam=lambda x: x["name"],
                dtype=str),

        Feature(key="price_usd",            # Keys support renaming
                lam=lambda x: x["priceUsd"],
                dtype=float),

        Feature(key="volume_usd_24_hr",
                lam=lambda x: x["volumeUsd24Hr"],
                dtype=float)
    ]

    post_process = [
        ColumnAdd(key="arrival_time",
                  lam=lambda: int(time.time()*1000),
                  dtype=int
                  )
    ]

    return ApiForm(name="coincap", pre_process=pre_process, data_features=data_
↪features, post_process=post_process)

.
.
.
```

**Once the lab has been moved over, you can optionally run the `clab` shell command to delete the lab**

### 2.1.9 Setting up an authentication file for database targets

1. Create a JSON file in the AUTH directory
2. Copy the template for the database target you wish to use
3. Fill out the template

### 2.1.10 Setting up a stream target for live data

**The following code will set up live streaming both to a local file location, and to a MySQL database**

coincap.py

```
.
.
.

def coincap_streams():
    """
    In this example, we will stream data live into a local file, and directly into a␣
↪MySQL database.
    """
    streams = [
        Stream2Local(name="coincap",
                     path="STORE/coincap/live"
                     ),

        Stream2Sql(name="coincap",
                   auth_path="AUTH/mysql_auth.json",
                   db_name="stream_coincap",
                   dialect="mysql",
                   port="3306"
                   )
    ]
    return streams

.
.
.
```

**Yes it is that easy, no you do not have to build the tables.**

### 2.1.11 Setting up a store target for data

**The following will set up a storage target that will pull data from `STORE/coincap/live` and store it to a MariaDB database periodically**

coincap.py

```
.
.
.

def coincap_stores():
    """
```

(continues on next page)

(continued from previous page)

```
    In this example, we will store data every 10 minutes to a MariaDB database.
    The files we store will then be composed into a single file, and stored in a␣
↪different storage location.
    """
    stores = [
        Store2Sql(name="coincap",
                  seconds=600,
                  path="STORE/coincap/live",
                  db_name="store_coincap",
                  auth_path="AUTH/mariadb_auth.json",
                  port="3306",
                  dialect="mariadb",
                  move_composed_path="STORE/coincap/ten_minute_intervals/"
        )
    ]
    return stores


.
.
.
```

### 2.1.12 Registering a collector to run

**To register a collector, all that needs to be done is set the import frequency by changing the `seconds` parameter**

coincap.py

```
.
.
.

coincap_info = Collector(name="coincap",
                         seconds=30,                          # Import data from the API␣
↪every 30 seconds
                         import_target=coincap_import,
                         api_form=coincap_form,
                         streams=coincap_streams,
                         stores=coincap_stores,
                         debug=True                           # Set to False for␣
↪production
                         )


.
.
.
```

### 2.1.13 Running the application

**Run main.py**

Info Log Outputs:

```
2021-05-04 01:01:14 stream.py                  INFO  stream starting -> (local.parquet)
2021-05-04 01:01:14 stream.py                  INFO  stream starting -> (sql.mysql)
2021-05-04 01:01:14 api2db.py                  INFO  import scheduled: [30 seconds]␣
↪(api request data) -> (streams)
2021-05-04 01:01:14 api2db.py                  INFO  storage refresh scheduled: [30␣
↪seconds] -> (check stores)
2021-05-04 01:01:15 api2db.py                  INFO  storage scheduled: [600 seconds]␣
↪(STORE/coincap/live) -> (store)
2021-05-04 01:01:15 stream2sql.py              INFO  establishing connection to mysql:/
↪/***/stream_coincap
2021-05-04 01:01:15 stream2sql.py              INFO  database not found mysql://***.
↪com/stream_coincap... creating database
2021-05-04 01:01:15 stream2sql.py              INFO  connection established mysql://
↪***/stream_coincap
2021-05-04 01:01:25 store.py                   INFO  storage files composed,␣
↪attempting to store 3600 rows to mariadb://***/store_coincap
2021-05-04 01:01:25 stream2sql.py              INFO  establishing connection to␣
↪mariadb://***/store_coincap
2021-05-04 01:01:25 stream2sql.py              INFO  database not found mariadb://***/
↪store_coincap... creating database
2021-05-04 01:01:25 stream2sql.py              INFO  connection established mariadb://
↪***/store_coincap
```

Debug Log Outputs:

```
.
.
.
2021-05-04 01:01:24 stream2sql.py              DEBUG 100 rows inserted into mysql://
↪***/stream_coincap
2021-05-04 01:01:24 stream2local.py            DEBUG storing 100 rows to STORE/coincap/
↪live
2021-05-04 01:01:24 stream2sql.py              DEBUG 100 rows inserted into mysql://
↪***/stream_coincap
2021-05-04 01:01:25 stream2local.py            DEBUG storing 100 rows to STORE/coincap/
↪live
2021-05-04 01:01:25 stream2sql.py              DEBUG 100 rows inserted into mysql://
↪***/stream_coincap
2021-05-04 01:01:25 stream2sql.py              DEBUG 3600 rows inserted into mariadb://
↪***/store_coincap
2021-05-04 01:01:25 stream2local.py            DEBUG storing 100 rows to STORE/coincap/
↪live
2021-05-04 01:01:25 stream2sql.py              DEBUG 100 rows inserted into mysql://
↪***/stream_coincap
2021-05-04 01:01:26 stream2local.py            DEBUG storing 100 rows to STORE/coincap/
↪live
2021-05-04 01:01:26 stream2sql.py              DEBUG 100 rows inserted into mysql://
↪***/stream_coincap
2021-05-04 01:01:26 stream2local.py            DEBUG storing 100 rows to STORE/coincap/
↪live
2021-05-04 01:01:26 stream2sql.py              DEBUG 100 rows inserted into mysql://
↪***/stream_coincap
```

(continues on next page)

(continued from previous page)

```
.
.
.
```

## 2.2 Examples

### 2.2.1 Shell Commands

#### pmake

This shell command is used for initial creation of the project structure.

Given a blank project directory

```
project_dir-----/
```

**Shell Command:** `path/to/project_dir> pmake FooCollector BarCollector`

```
project_dir-----/
                |
                apis-----/
                |        |- __init__.py
                |        |- FooCollector.py
                |        |- BarCollector.py
                |
                AUTH-----/
                |        |- bigquery_auth_template.json
                |        |- omnisci_auth_template.json
                |        |- sql_auth_template.json
                |
                CACHE/
                |
                STORE/
                |
                helpers.py
                |
                main.py
```

**Note:** This command can also be used without any collector arguments, and collectors can be added using the `cadd` shell command.

### cadd

This shell command is used to add a collector to an existing api2db project

Given the following project structure

```
project_dir-----/
                |
                apis-----/
                |        |- __init__.py
                |        |- FooCollector.py
                |
                AUTH-----/
                |        |- bigquery_auth_template.json
                |        |- omnisci_auth_template.json
                |        |- sql_auth_template.json
                |
                CACHE/
                |
                STORE/
                |
                helpers.py
                |
                main.py
```

**Shell Command:** `path/to/procect_dir> cadd BarCollector`

```
project_dir-----/
                |
                apis-----/
                |        |- __init__.py
                |        |- FooCollector.py
                |        |- BarCollector.py
                |
                AUTH-----/
                |        |- bigquery_auth_template.json
                |        |- omnisci_auth_template.json
                |        |- sql_auth_template.json
                |
                CACHE/
                |
                STORE/
                |
                helpers.py
                |
                main.py
```

### crem

This shell command is used to remove a collector registered with an existing api2db project

Given the following project

```
project_dir-----/
                |
                apis-----/
                |        |- __init__.py
```

(continues on next page)

(continued from previous page)

```
                |        |- FooCollector.py
                |        |- BarCollector.py
                |
                AUTH-----/
                |        |- bigquery_auth_template.json
                |        |- omnisci_auth_template.json
                |        |- sql_auth_template.json
                |
                CACHE/
                |
                STORE/
                |
                helpers.py
                |
                main.py
```

**Shell Command:** `path/to/project_dir> crem BarCollector`

```
project_dir-----/
                |
                apis-----/
                |        |- __init__.py
                |        |- FooCollector.py
                |
                AUTH-----/
                |        |- bigquery_auth_template.json
                |        |- omnisci_auth_template.json
                |        |- sql_auth_template.json
                |
                CACHE/
                |
                STORE/
                |
                helpers.py
                |
                main.py
```

### clist

This shell command is used to show a list of collectors registered with an existing api2db project

Given the following project

```
project_dir-----/
                |
                apis-----/
                |        |- __init__.py
                |        |- FooCollector.py
                |        |- BarCollector.py
                |
                AUTH-----/
                |        |- bigquery_auth_template.json
                |        |- omnisci_auth_template.json
                |        |- sql_auth_template.json
                |
```

(continues on next page)

(continued from previous page)

```
                CACHE/
                |
                STORE/
                |
                helpers.py
                |
                main.py
```

**Shell Command:** `path/to/procect_dir> clist`

**Out:** `["FooCollector", "BarCollector"]`

### pclear

This shell command is used to clear a project and should **ONLY** be used if a complete restart is required.

Given the following project

```
project_dir-----/
                |
                apis-----/
                |        |- __init__.py
                |        |- FooCollector.py
                |        |- BarCollector.py
                |
                AUTH-----/
                |        |- bigquery_auth_template.json
                |        |- omnisci_auth_template.json
                |        |- sql_auth_template.json
                |
                CACHE/
                |
                STORE/
                |
                helpers.py
                |
                main.py
```

**Shell Command:** `path/to/project_dir> pclear`

```
project_dir-----/
```

### mlab

This shell command is used for creation of a lab. Labs offer an easier way to design an ApiForm.

Given a project directory

```
project_dir-----/
                |
                apis-----/
                |        |- __init__.py
                |        |- FooCollector.py
                |        |- BarCollector.py
                |
```

(continues on next page)

(continued from previous page)

```
                AUTH-----/
                |        |- bigquery_auth_template.json
                |        |- omnisci_auth_template.json
                |        |- sql_auth_template.json
                |
                CACHE/
                |
                STORE/
                |
                helpers.py
                |
                main.py
```

**Shell Command:** `path/to/project_dir> mlab`

```
project_dir-----/
                |
                apis-------/
                |          |- __init__.py
                |          |- FooCollector.py
                |          |- BarCollector.py
                |
                AUTH-------/
                |          |- bigquery_auth_template.json
                |          |- omnisci_auth_template.json
                |          |- sql_auth_template.json
                |
                CACHE/
                |
                STORE/
                |
                laboratory-/
                |          |- lab.py    EDIT THIS FILE!
                |
                helpers.py
                |
                main.py
```

### rlab

This shell command is used to run a lab.

### clab

This shell command is used to clear a lab.

```
project_dir-----/
                |
                apis-------/
                |          |- __init__.py
                |          |- FooCollector.py
                |          |- BarCollector.py
                |
                AUTH-------/
```

(continues on next page)

(continued from previous page)

```
                |          |- bigquery_auth_template.json
                |          |- omnisci_auth_template.json
                |          |- sql_auth_template.json
                |
                CACHE/
                |
                STORE/
                |
                laboratory-/
                |          |- lab.py    EDIT THIS FILE!
                |
                helpers.py
                |
                main.py
```

**Shell Command:** `path/to/project_dir> clab`

```
project_dir-----/
            |
            apis-----/
            |        |- __init__.py
            |        |- FooCollector.py
            |        |- BarCollector.py
            |
            AUTH-----/
            |        |- bigquery_auth_template.json
            |        |- omnisci_auth_template.json
            |        |- sql_auth_template.json
            |
            CACHE/
            |
            STORE/
            |
            helpers.py
            |
            main.py
```

### 2.2.2 Pre-processing

#### BadRowSwap

---

**Note:** BadRowSwap should not be used until **AFTER** ListExtract has been performed on the data, unless performing a list extract is not necessary on the data.

When using BadRowSwap, the following conditions must be met:

1. The value contained at location `key_1` must be able to be identified as valid, or in need of being swapped without any reference to the value at location `key_2`. (Typically using regex or performing type-checking)
2. `key_1` and `key_2` must be unique within their respective row of data. `data = {"key_1": {"key_1": 1, "key_2": 2}}` would be invalid.

BadRowSwap **will potentially drop rows of data**. Rows meeting the following conditions will be dropped:

- Any row that is missing `key_1` as a key will be dropped.

- Any row that evaluates as needing to be swapped based on `key_1` that is missing `key_2` will be dropped.

BadRowSwap will keep rows that meet the following conditions:

- Any row that evaluates as not needing to be swapped based on `key_1` will be kept, regardless of if `key_2` exists or not.
- Any row that evaluates as needing to be swapped based on `key_1` that also contains `key_2` will swap the values at the locations of the `key_1` and `key_2` and the row will be kept.

Performing BadRowSwap can be computationally expensive, since it walks all nested data until it finds the desired keys. So here are a few tips to help you determine if you should be using it or not.

Usage Tips for using BadRowSwap:

- If both `key_1` and `key_2` are unimportant fields, I.e. Nullable fields and keeping them does not add significant value to the data consider just allowing the collector to Null them if they do not match the types or consider allowing them to simply have the wrong values if they have the same data-types. Otherwise you risk both slowing down data-collection, and dropping rows that have good data other than those swapped rows.
- Always attempt to place the key at location `key_1` as the more important value to retain. If you need to swap data like a "uuid" and a "description", use the "uuid" as `key_1`
- If you cannot place the key at location `key_1` as the more important key, consider if the risk of losing data with a valid value for the more important key is worth it in instances where the less important key is missing
- Consider the frequency that BadRowSwap would need to be run. If 1 out of 1,000,000 data-points contains values with swapped keys, is it worth running the computation on all 1,000,000 rows to save just that 1 row?
- Analyze the data by hand. Pull it into a pandas DataFrame, and check it.
  - How often are is a row incorrect?
  - Are the erroneous rows ALWAYS the same key?
  - How often is one of the keys for the row missing when the rows have bad data?

---

**Summary of BadRowSwap usage:**

```
data = [
    {
        "id": "17.0",
        "size": "Foo",
    },
    {
        "id": "Bar",
        "size": "10.0"
    }
]

pre = BadRowSwap(key_1="id",
                 key_2="size",
                 lam=lambda x: re.match("[0-9][0-9]\.[0-9]+", x["id"]) is not None
                 )
```

**Example Usage of BadRowSwap:**

Occasionally when dealing with an API, the data is not always where it is supposed to be. Oftentimes this results in the rows containing the misplaced data being dropped altogether. In the instance that for some unknown reason the incoming data has keys that tend to occasionally have their values swapped so long as it is possible to check to see if the data has been swapped due to what the data *should* be, use BadRowSwap.

This example assumes that the API occasionally swaps the values for "id" and "latitude". BadRowSwap can handle any level of nested data in these instances, so long as the keys for the values that are occasionally swapped are **unique within a single row**

```
>>> import re
... data = [
...     {
...         "id": "16.53",                                    # NEEDS SWAP = True
...          "place": {
...              "coords": {
...                  "latitude": "ID_1",
...                  "longitude": "-20.43"
...              },
...              "name": "place_1"
...          },
...         "details": "Some details... etc"
...      },
...
...     {
...         "id": "ID_2",                                     # NEEDS SWAP = False
...          "place": {
...              "coords": {
...                  "latitude": "15.43",
...                  "longitude": "-20.43"
...              },
...              "name": "place_2"
...          },
...         "details": "Some details... etc"
...      },
...
...     {
...         "id": "10.21",                                    # NEEDS SWAP = True
...         "place": {
...             "coords": {
...                                                           # Missing "latitude" key,␣
↪results in row being skipped
...                 "longitude": "-20.43"
...             },
...             "name": "place_2"
...         },
...         "details": "Some details... etc"
...     },
...
...     {                                                     # Missing "id" key,␣
↪results in row being skipped
...         "place": {
...             "coords": {
...                 "latitude": "ID_4",
...                 "longitude": "-20.43"
...             },
...             "name": "place_2"
...         },
...         "details": "Some details... etc"
...     },
...
...     {
...         "id": "ID_5",                                     # NEEDS SWAP = False
```

(continues on next page)

(continued from previous page)

```
...         "place": {
...             "coords": {
...                                                     # Missing "latitude" row␣
↪is kept, because no row swap needed
...                 "longitude": "-20.43"
...             },
...             "name": "place_2"
...         },
...         "details": "Some details... etc"
...     }
... ]
...
... pre = BadRowSwap(key_1="id",
...                  key_2="latitude",
...                  lam=lambda x: re.match("[0-9][0-9]\.[0-9]+", x["id"]) is not None
...                  )
...
... pre.lam_wrap(data)
[
    {
        "id": "ID_1",  # "id" and "latitude" have been swapped
        "place": {
            "coords": {
                "latitude": "16.53",
                "longitude": "-20.43"
            },
            "name": "place_1"
        },
        "details": "Some details... etc"
    },
    {
        "id": "ID_2",  # No changes required with this row
        "place": {
            "coords": {
                "latitude": "15.43",
                "longitude": "-20.43"
            },
            "name": "place_2"
        },
        "details": "Some details... etc"
    },
    # Row 3, and Row 4 have been dropped because they were missing key_1 or they␣
↪required a swap and were missing key_2
    {
        "id": "ID_5",  # No changes required with this row
        "place": {
            "coords": {
                # The latitude is still missing but that can be handled later, it may␣
↪be nullable, so it should be kept
                "longitude": "-20.43"
            },
            "name": "place_2"
        },
        "details": "Some details... etc"
    }
]
```

### FeatureFlatten

**Note:** FeatureFlatten should not be used until **AFTER** ListExtract has been performed on the data, unless performing a list extract is not necessary on the data.

**Summary of FeatureFlatten usage:**

```
data = [
    {
        "data_id": 1,
        "data_features": [
                                {
                                    "x": 5,
                                    "y": 10
                                },
                                {
                                    "x": 7,
                                    "y": 15
                                },
                                .
                                .
                                .
                            ]
    }
]
pre = FeatureFlatten(key="data_features")
```

**Example Usage of FeatureFlatten:**

```
>>> data = [
...     {
...         "data_id": 1,
...         "data_features": {
...                             "Foo": 5,
...                             "Bar": 10
...                          }
...     },
...
...     {
...         "data_id": 2,
...         "data_features": [
...                             {
...                                 "Foo": 5,
...                                 "Bar": 10
...                             },
...                             {
...                                 "Foo": 7,
...                                 "Bar": 15
...                             }
...                          ]
...     }
... ]
... pre = FeatureFlatten(key="data_features")
... pre.lam_wrap(data)
[
    {
```

(continues on next page)

(continued from previous page)

```
        "data_id": 1,
        "data_features": {
                            "Foo": 5,
                            "Bar": 10
                        }
    },
    {
        "data_id": 2,
        "data_features": {
                            "Foo": 5,
                            "Bar": 10
                        }
    },
    {
        "data_id": 2,
        "data_features": {
                            "Foo": 7,
                            "Bar": 15
                        }
    }
]
```

### GlobalExtract

**Summary of GlobalExtract usage:**

```
data = {"date": "2021-04-19", "data_array": [{"id": 1, "name": "Foo"}, {"id": 2, "name
↪": "Bar"}]}
pre = GlobalExtract(key="publish_time",
                    lam=lambda x: x["date"],
                    dtype=str
                    )
```

**Final DataFrame**

| id | name | publish_time |
|---|---|---|
| 1 | Foo | 2021-04-19 |
| 2 | Bar | 2021-04-19 |

**Example Usage of GlobalExtract:**

```
>>> # pre-processing operators
... pres = []
... # Dictionary that contains all globally extracted data
... pre_2_post_dict = {}
... # Incoming Data
... data = {"date": "2021-04-19", "data_array": [{"id": 1, "name": "Foo"}, {"id": 2,
↪"name": "Bar"}]}
... # GlobalExtract instance for extracting the "date" from data, but replacing its␣
↪key with "publish_time"
... pre = GlobalExtract(key="publish_time",
...                     lam=lambda x: x["date"],
...                     dtype=str
...                     )
```

(continues on next page)

(continued from previous page)

```
... # The preprocessor gets added to the list of preprocessors
... pres.append(pre)
... # Each preprocesser gets applied sequentially
... for p in pres:
...     if p.ctype == "global_extract":
...         pre_2_post_dict[p.key] = p.lam_wrap(data)
...     else:
...         pass # See other pre-processors
... pre_2_post_dict
{"publish_time": {"value": "2021-04-19", "dtype": str}}
```

**Later after the data has been extracted to a DataFrame df**

```
# Assume df = DataFrame containing extracted data
# Assume dtype_convert is a function that maps a python native type to a pandas dtype

# For each globally extracted item
for k, v in pre_2_post_dict.items():
    # Add the item to the DataFrame -> These are GLOBAL values shared amongst ALL rows
    df[k] = v["value"]
    # Typecast the value to ensure it is the correct dtype
    df[k] = df[k].astype(dtype_convert(v["dtype"]))
```

**Example of what DataFrame would be:**

| id | name | publish_time |
|---|---|---|
| 1 | Foo | 2021-04-19 |
| 2 | Bar | 2021-04-19 |

### ListExtract

**Summary of ListExtract Usage:**

```
data = { "actual_data_rows": [{"id": "row1"}, {"id": "row2"}], "erroneous_data":
↪"FooBar" }
pre = ListExtract(lam=lambda x: x["actual_data_rows"])
```

**Example Usage of ListExtract:**

```
>>> data = {
...     "Foo": "Metadata",
...     "data_array": [
...             {
...                 "data_id": 1,
...                 "name": "name_1"
...             },
...             {
...                 "data_id": 2,
...                 "name": "name_2"
...             }
...         ]
... }
...
... pre = ListExtract(lam=lambda x: x["data_array"])
```

(continues on next page)

(continued from previous page)

```
... pre.lam_wrap(data)
[
    {
        "data_id": 1,
        "name": "name_1"
    },
    {
        "data_id": 2,
        "name": "name_2"
    }
]
```

### 2.2.3 Extracting data-features

#### Summary of Feature Usage:

```
data = [{"id": 1, "name": "Foo", "nest0": {"nest1": {"x": True}, "y": 14.3 } }, ... ]
data_features = [

    Feature(key="uuid", lam=lambda x: x["id"], dtype=int),                  #␣
↪Extracts "id" and rename it to "uuid"

    Feature(key="name", lam=lambda x: x["name"], dtype=str),                # Will␣
↪extract "name" keeping the key as "name"

    Feature(key="x", lam=lambda x: x["nest0"]["nest1"]["x"], dtype=bool),   # Will␣
↪extract "x"

    Feature(key="y", lam=lambda x: x["nest0"]["y"], dtype=bool)             # Will␣
↪extract "y"
]
```

### 2.2.4 Post-processing

#### ColumnAdd

**Summary of ColumnAdd Usage:**

DataFrame `df`

| Foo | Bar |
|---|---|
| 1 | A |
| 2 | B |
| 3 | C |

```
post = ColumnAdd(key="FooBar", lam=lambda: 5, dtype=int)
```

DataFrame `df`

| Foo | Bar | FooBar |
|---|---|---|
| 1 | A | 5 |
| 2 | B | 5 |
| 3 | C | 5 |

**Example Usage of ColumnAdd:**

```
>>> import pandas as pd
... def f():
...     return 5
... df = pd.DataFrame({"Foo": [1, 2, 3], "Bar": ["A", "B", "C"]})    # Setup
...
... post = ColumnAdd(key="timestamp", lam=lambda x: f, dtype=int)
... post.lam_wrap(df)
pd.DataFrame({"Foo": [1, 2, 3], "Bar": ["A", "B", "C"], "FooBar": [5, 5, 5]})
```

### ColumnApply

**Summary of ColumnApply Usage:**

DataFrame `df`

| Foo | Bar |
|---|---|
| 1 | A |
| 2 | B |
| 3 | C |

```
post = ColumnApply(key="Foo", lam=lambda x: x + 1, dtype=int)
```

DataFrame `df`

| Foo | Bar |
|---|---|
| 2 | A |
| 3 | B |
| 4 | C |

**Example Usage of ColumnApply:**

```
>>> import pandas as pd
... df = pd.DataFrame({"Foo": [1, 2, 3], "Bar": ["A", "B", "C"]})    # Setup
...
... post = ColumnApply(key="Foo", lam=lambda x: x + 1, dtype=int)
... post.lam_wrap(df)
pd.DataFrame({"Foo": [2, 3, 4], "Bar": ["A", "B", "C"]})
```

### ColumnsCalculate

---

**Note: ColumnsCalculate can be used to**

1. Replace columns in a DataFrame with calculated values
2. Add new columns to a DataFrame based on calculations from existing columns

---

**Summary of ColumnsCalculate Usage:**

DataFrame `df`

| Foo | Bar |
|---|---|
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

```
def foobar(df):
    df["Foo+Bar"] = df["Foo"] + df["Bar"]
    df["Foo*Bar"] = df["Foo"] * df["Bar"]
    return df[["Foo+Bar", "Foo*Bar"]]

post = ColumnsCalculate(keys=["Foo+Bar", "Foo*Bar"], lam=lambda x: foobar(x),␣
↪dtype=int)
```

DataFrame `df`

| Foo | Bar | Foo+Bar | Foo*Bar |
|---|---|---|---|
| 1 | 2 | 3 | 2 |
| 2 | 4 | 6 | 8 |
| 3 | 8 | 11 | 24 |

**Example Usage of ColumnsCalculate:**

```
>>> import pandas as pd
... df = pd.DataFrame({"Foo": [1, 2, 3], "Bar": [2, 4, 8]})    # Setup
...
... def foobar(d):
...     d["Foo+Bar"] = d["Foo"] + d["Bar"]
...     d["Foo*Bar"] = d["Foo"] * d["Bar"]
...     return d[["Foo+Bar", "Foo*Bar"]]
...
... post = ColumnsCalculate(keys=["Foo+Bar", "Foo*Bar"], lam=lambda x: foobar(x),␣
↪dtype=int)
... post.lam_wrap(df)
pd.DataFrame({"Foo+Bar": [3, 6, 11], "Foo*Bar": [2, 8, 24]})
```

### DateCast

**Summary of DateCast Usage:**

DataFrame `df`

| Foo | Bar |
|---|---|
| 2021-04-29 01:39:00 | False |
| 2021-04-29 01:39:00 | False |
| Bar! | True |

DataFrame `df.dtypes`

| Foo | Bar |
|---|---|
| string | bool |

```
post = DateCast(key="Foo", fmt="%Y-%m-%d %H:%M:%S")
```

DataFrame `df`

| Foo | Bar |
|---|---|
| 2021-04-29 01:39:00 | False |
| 2021-04-29 01:39:00 | False |
| NaT | True |

DataFrame `df.dtypes`

| Foo | Bar |
|---|---|
| datetime64[ns] | bool |

### DropNa

Simply a shortcut class for a common operation.

**Summary of DropNa Usage:**

See pandas Documentation

### MergeStatic

---

**Note:** MergeStatic is used to merge data together. A common use case of this is in situations where a data-vendor provides an API that gives data-points "Foo", "Bar", and "location_id" where "location_id" references a different data-set.

It is common for data-providers to have a file that does not update very frequently, i.e. is mostly static that contains this information.

The typical workflow of a MergeStatic instance is as follows:

1. Create a LocalStream with mode set to *update* or *replace* and a target like *CACHE/my_local_stream.pickle*
2. Set the LocalStream to run periodically (6 hours, 24 hours, 10 days, whatever frequency this data is updated)

3. Add a MergeStatic object to the frequently updating datas post-processors and set the path to the LocalStream storage path.

## 2.3 The future of api2db

### 2.3.1 Future Plans

- Add support for more storage targets
  - Oracle
  - Amazon Aurora
  - MariaDB
  - Microsoft SQL Server
  - Firebase RTDB
- Add support for uploading in chunks and resumable uploads
- Add support for sharding data collection across multiple servers.
  - If one server does not send a heartbeat out after a certain period of time, another server begins collecting data.
- Add 100% test coverage for code-base
- Add library support utilizing the ApiForm to create database migration support
  - Treat a database as a stream, and pull data from it in chunks before migrating to a new target
  - Used to switch databases, and also clean messy database data
- Add ML targets that can be attached directly to streams
  - Allow for streams to feed directly into predictive models
- Add support for an api2db implementation
  - Performs things in a manner opposite of api2db
  - Objects such as an EndPoint object used to create Api Endpoints
  - Take any database, and turn it into an API
  - Include role-based authentication
- Remove BaseLam object
  - Since collectors run in a single process, this needs depreciated. No need to serialize the state
- Add additional pre/post processors
  - Listen to what users want upon release, and implement them
- Remove unnecessary strings
  - Fix implementations using strings to represent object types I.e. ctype in processors
  - Use isinstance, and pandas.api.types.is_x_type
- Add support for GPU processing of data
  - Allow for ingestion to be performed on a GPU for high-volume streams

- Rewrite performance critical areas of application in C
- Create a Store2Local object that can be used to aggregate storage in time intervals
- Add support for messaging
  - Redis Pub/Sub
  - Google Cloud Pub/Sub
  - Kafka Pub/Sub
  - Using Firestore
- Add the ability to generate live insights
  - As data arrives, create the ability to perform rolling averages of data
  - Allow for chaining messaging abilities onto streams

## 2.4 api2db

### 2.4.1 api2db package

#### Subpackages

#### api2db.app package

#### Submodules

#### api2db.app.api2db module

#### Contains the Api2Db class

api2db.app.api2db.**DEV_SHRINK_DATA = 0**
Library developer setting to shrink incoming data to the first DEV_SHRINK_DATA rows

**Type** int

**class** api2db.app.api2db.**Api2Db**(*collector:* api2db.ingest.collector.Collector)
Bases: object

Performs data import, passes data to streams, and schedules data storage

**__init__**(*collector:* api2db.ingest.collector.Collector)
Creates a Api2Db object and attaches the collector

**Parameters** **collector** – The collector object to attach to

**wrap_start**() → multiprocessing.context.Process
Starts the running loop of an Api2Db instance in a spawned process

**Returns** The process spawned with target start

**start**() → None
The target for Api2Db main process running loop

**Returns** None

**schedule**() → None
schedule starts the streams, schedules collector refresh, schedules storage refresh

**Returns** None

**Raises** **NameError or ModuleNotFoundError if streams cannot be created** –

**static collect_wrap**(*import_target: Callable[], Optional[List[dict]]], api_form: Callable[], api2db.ingest.api_form.ApiForm], stream_qs: List[queue.Queue], stream_locks: List[_thread.allocate_lock]*) → Optional[type]
Starts/restarts dead streams, and calls method collect to import data

**Parameters**

- **import_target** – Function that returns data imported from an Api
- **api_form** – Function that instantiates and returns an ApiForm object
- **stream_qs** – A list of queues to pass the incoming data into to be handled by stream targets
- **stream_locks** – A list of locks that become acquirable if their respective stream has died

**Returns** CancelJob if stream has died, restarting the streams, None otherwise

**static collect**(*import_target: Callable[], Optional[List[dict]]], api_form: Callable[], api2db.ingest.api_form.ApiForm], stream_qs: List[queue.Queue]*) → None
Performs a data-import, cleans the data, and sends the data into its stream queues

**Parameters**

- **import_target** – Function that returns data imported from an Api
- **api_form** – Function that instantiates and returns an ApiForm object
- **stream_qs** – A list of queues to pass the incoming data into to be handled by stream targets

**Returns** None

**static store_wrap**(*stores: Callable[], List[api2db.store.store.Store]]*) → None
Checks to ensure that storage jobs are scheduled to run and schedules any jobs that have been unscheduled

**Parameters** **stores** – Function that returns a list of Store subclassed objects

**Returns** None

**Raises** **NameError or ModuleNotFoundError if stores cannot be created** –

**static store**(*store: api2db.store.store.Store*) → None
Performs the data storage operation of a Store subclass

**Parameters** **store** – The Store to perform storage on

**Returns** None

**static import_handle**(*e: Exception*) → Exception
Handles import errors. Informs the user of libraries they need

**Parameters** **e** – The raised Exception

**Returns** ModuleNotFoundError if dependencies missing otherwise the original exception

### api2db.app.auth_manager module

### Contains the auth_manage function

api2db.app.auth_manager.**auth_manage**(*path: str*) → Optional[dict]
Loads authentication credentials from the specified path

**Parameters** **path** – The path where the authentication file resides

**Returns** Authentication credentials if file successfully loaded, None otherwise

### api2db.app.log module

### Contains the get_logger function

api2db.app.log.**get_logger**(*filename: Optional[str] = None*, *q: Optional[multiprocessing.context.BaseContext.Queue] = None*) → logging.Logger
Retrieves the logger for the current process for logging to the log file

If no filename is provided, the logger for the current process is assumed to already have handlers registered, and will be returned.

If a filename is provided an the logger has no handlers, a handler will be created and registered

**Parameters**

- **filename** – The name of the file to log to
- **q** – The queue used to pass messages if the collector is running in debug mode

**Returns** A logger that can be used to log messages

### api2db.app.run module

### Contains the Run class

**class** api2db.app.run.**Run**(*collectors: List[*api2db.ingest.collector.Collector*]*)
Bases: object

Serves as the main entry point for the application

**__init__**(*collectors: List[*api2db.ingest.collector.Collector*]*)
The Run object is the application entry point

**Parameters** **collectors** – A list of collector objects to collect data for

**q**
Used for message passing for collectors with debug mode enabled

**Type** multiprocessing.Queue

**run**()
Starts the application

**Returns** None

**multiprocessing_start**()
Starts each collector in it's own process

**Returns** None

### Module contents

| Original Author | Tristen Harr |
|---|---|
| Creation Date | 04/27/2021 |
| Revisions | None |

## api2db.ingest package

### Subpackages

### api2db.ingest.data_feature package

### Submodules

### api2db.ingest.data_feature.feature module

### Contains the Feature class

### Summary of Feature Usage:

```
data = [{"id": 1, "name": "Foo", "nest0": {"nest1": {"x": True}, "y": 14.3 } }, ... ]
data_features = [

    Feature(key="uuid", lam=lambda x: x["id"], dtype=int),      # Extracts "id" and␣
↪rename it to "uuid"

    Feature(key="name", lam=lambda x: x["name"], dtype=str),    # Will extract "name"␣
↪keeping the key as "name"

    Feature(key="x", lam=lambda x: x["nest0"]["nest1"]["x"], dtype=bool),   # Will␣
↪extract "x"

    Feature(key="y", lam=lambda x: x["nest0"]["y"], dtype=bool)             # Will␣
↪extract "y"
]
```

**class** api2db.ingest.data_feature.feature.**Feature**(*key: str*, *lam: Callable[[dict], Any]*, *dtype: Any*, *nan_int: Optional[int] = None*, *nan_float: Optional[float] = None*, *nan_bool: Optional[bool] = False*, *nan_str: Optional[str] = None*)

Bases: *api2db.ingest.base_lam.BaseLam*

Used to extract a data-feature from incoming data

**__init__**(*key: str*, *lam: Callable[[dict], Any]*, *dtype: Any*, *nan_int: Optional[int] = None*, *nan_float: Optional[float] = None*, *nan_bool: Optional[bool] = False*, *nan_str: Optional[str] = None*)

Creates a Feature object

**Note:** All values default to nulling the data that cannot be type-casted to its expected type. For the majority of instances this is going to be the programmers desired effect. If there is a way to make it so that the data can be cleaned in order to prevent it from being nulled, that should be done using the libraries pre-processing tools. If the data cannot be cleaned in pre-processing and it cannot be type-casted to its expected type, then it is by definition worthless. If it is possible to clean it, it can be cleaned in pre-processing, although it may require the programmer to subclass *Pre*

**Parameters**

- **key** – The name of the column that will be stored in the storage target
- **lam** – Function that takes as parameter a dictionary, and returns where the data the programmer wants **should** be. api2db handles null data and unexpected data types automatically
- **dtype** – The python native type of the data feature
- **nan_int** – If specified and `dtype` is `int` this value will be used to replace null values and values that fail to be casted to type `int`
- **nan_float** – If specified and `dtype` is `float` this value will be used to replace null values and values that fail to be casted to type `float`
- **nan_bool** – If specified and `dtype` is `bool` this value will be used to replace null values and values that fail to be casted to type `bool`
- **nan_str** – If specified and `dtype` is `str` this value will be used to replace null values and values that fail to be casted to type `str`

**lam_wrap**(*data: dict*) → Any

Overrides super class method

Extracts a feature from incoming data

Workflow:

1. Attempt to call `lam` on data to get data-feature
2. Attempt to typecast result to `dtype`
3. If `dtype` is `str` and the result.lower() is "none", "nan", "null", or "nil" replace it with `nan_str`
4. If an exception occurs when attempting any of the above, set the result to None
5. Return the result

**Parameters** **data** – A dictionary of incoming data representing a single row in a DataFrame

**Returns** The extracted data-feature

### Module contents

| Original Author | Tristen Harr |
|---|---|
| Creation Date | 04/29/2021 |
| Revisions | None |

### api2db.ingest.post_process package

### Submodules

### api2db.ingest.post_process.column_add module

### Contains the ColumnAdd class

### Summary of ColumnAdd Usage:

**DataFrame** `df`

| Foo | Bar |
|---|---|
| 1 | A |
| 2 | B |
| 3 | C |

```
post = ColumnAdd(key="FooBar", lam=lambda: 5, dtype=int)
```

**DataFrame** `df`

| Foo | Bar | FooBar |
|---|---|---|
| 1 | A | 5 |
| 2 | B | 5 |
| 3 | C | 5 |

### Example Usage of ColumnAdd:

```
>>> import pandas as pd
... def f():
...     return 5
... df = pd.DataFrame({"Foo": [1, 2, 3], "Bar": ["A", "B", "C"]})    # Setup
...
... post = ColumnAdd(key="timestamp", lam=lambda x: f, dtype=int)
... post.lam_wrap(df)
pd.DataFrame({"Foo": [1, 2, 3], "Bar": ["A", "B", "C"], "FooBar": [5, 5, 5]})
```

**class** api2db.ingest.post_process.column_add.**ColumnAdd**(*key: str*, *lam: Callable[], Any]*, *dtype: Any*)

Bases: *api2db.ingest.post_process.post.Post*

Used to add global values to a DataFrame, primarily for timestamps/ids

**__init__** (*key: str*, *lam: Callable[], Any]*, *dtype: Any*)
Creates a ColumnAdd object

**Parameters**

- **key** – The column name for the DataFrame
- **lam** – A function that returns the value that should be globally placed into the DataFrame in `key` column
- **dtype** – The python native type of the functions return

**ctype**
type of the data processor

**Type** str

**lam_wrap** (*lam_arg: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame
Overrides super class method

Workflow:

1. Assign the `lam` function return to `lam_arg[self.key]`
2. Typecast `lam_arg[self.key]` to `dtype`
3. Return `lam_arg`

**Parameters** **lam_arg** – The DataFrame to add a column to

**Returns** The modified DataFrame

### api2db.ingest.post_process.column_apply module

### Contains the ColumnApply class

### Summary of ColumnApply Usage:

**DataFrame** `df`

| Foo | Bar |
|---|---|
| 1 | A |
| 2 | B |
| 3 | C |

```
post = ColumnApply(key="Foo", lam=lambda x: x + 1, dtype=int)
```

**DataFrame** `df`

| Foo | Bar |
|---|---|
| 2 | A |
| 3 | B |
| 4 | C |

### Example Usage of ColumnApply:

```
>>> import pandas as pd
... df = pd.DataFrame({"Foo": [1, 2, 3], "Bar": ["A", "B", "C"]})    # Setup
...
... post = ColumnApply(key="Foo", lam=lambda x: x + 1, dtype=int)
... post.lam_wrap(df)
pd.DataFrame({"Foo": [2, 3, 4], "Bar": ["A", "B", "C"]})
```

**class** api2db.ingest.post_process.column_apply.**ColumnApply**(*key: str*, *lam: Callable[[Any], Any]*, *dtype: Any*)

Bases: *api2db.ingest.post_process.post.Post*

Used to apply a function across the rows in a column of a DataFrame

**__init__**(*key: str*, *lam: Callable[[Any], Any]*, *dtype: Any*)
Creates a ColumnApply Object

**Parameters**

- **key** – The column to apply the function to
- **lam** – The function to apply
- **dtype** – The python native type of the function output

**ctype**
type of data processor

**Type** str

**lam_wrap**(*lam_arg: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame
Overrides a super class method

Workflow:

1. Apply `lam` to `lam_arg[self.key]`
2. Cast `lam_arg[self.key]` to `dtype`
3. Return `lam_arg`

**Parameters** **lam_arg** – The DataFrame to modify

**Returns** The modified DataFrame

## api2db.ingest.post_process.columns_calculate module

### Contains the ColumnsCalculate class

**Note: ColumnsCalculate can be used to**

1. Replace columns in a DataFrame with calculated values
2. Add new columns to a DataFrame based on calculations from existing columns

### Summary of ColumnsCalculate Usage:

**DataFrame** `df`

| Foo | Bar |
|---|---|
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

```
def foobar(df):
    df["Foo+Bar"] = df["Foo"] + df["Bar"]
    df["Foo*Bar"] = df["Foo"] * df["Bar"]
    return df[["Foo+Bar", "Foo*Bar"]]

post = ColumnsCalculate(keys=["Foo+Bar", "Foo*Bar"], lam=lambda x: foobar(x),␣
↪dtype=int)
```

**DataFrame** `df`

| Foo | Bar | Foo+Bar | Foo*Bar |
|---|---|---|---|
| 1 | 2 | 3 | 2 |
| 2 | 4 | 6 | 8 |
| 3 | 8 | 11 | 24 |

### Example Usage of ColumnsCalculate:

```
>>> import pandas as pd
... df = pd.DataFrame({"Foo": [1, 2, 3], "Bar": [2, 4, 8]})    # Setup
...
... def foobar(d):
...     d["Foo+Bar"] = d["Foo"] + d["Bar"]
...     d["Foo*Bar"] = d["Foo"] * d["Bar"]
...     return d[["Foo+Bar", "Foo*Bar"]]
...
... post = ColumnsCalculate(keys=["Foo+Bar", "Foo*Bar"], lam=lambda x: foobar(x),␣
↪dtype=int)
... post.lam_wrap(df)
pd.DataFrame({"Foo+Bar": [3, 6, 11], "Foo*Bar": [2, 8, 24]})
```

**class** api2db.ingest.post_process.columns_calculate.**ColumnsCalculate**(*keys: List[str], lam: Callable[[pandas.core.frame.DataFrame], pandas.core.frame.DataFrame], dtypes: List[Any]*)

Bases: *api2db.ingest.post_process.post.Post*

Used to calculate new column values to add to the DataFrame

**__init__**(*keys: List[str], lam: Callable[[pandas.core.frame.DataFrame], pandas.core.frame.DataFrame], dtypes: List[Any]*)

Creates a ColumnsCalculate object

**Parameters**

- **keys** – A list of the keys to add/replace in the existing DataFrame
- **lam** – A function that takes as parameter a DataFrame, and returns a DataFrame with column names matching `keys` and the columns having/being castable to `dtypes`
- **dtypes** – A list of python native types that are associated with `keys`

**ctype**
type of data processor

**Type** str

**lam_wrap**(*lam_arg: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame
Overrides super class method

Workflow:

1. Create a temporary DataFrame `tmp_df` by applying `lam` to `lam_arg`
2. For each `key` in `self.keys` set `lam_arg[key] = tmp_df[key]`
3. For each `key` in `self.keys` cast `lam_arg[key]` to the appropriate pandas dtype
4. Return `lam_arg`

**Parameters** **lam_arg** – The DataFrame to modify

**Returns** The modified DataFrame

### api2db.ingest.post_process.date_cast module

### Contains the DateCast class

### Summary of DateCast Usage:

**DataFrame** `df`

| Foo | Bar |
|---|---|
| 2021-04-29 01:39:00 | False |
| 2021-04-29 01:39:00 | False |
| Bar! | True |

**DataFrame** `df.dtypes`

| Foo | Bar |
|---|---|
| string | bool |

```
post = DateCast(key="Foo", fmt="%Y-%m-%d %H:%M:%S")
```

**DataFrame** `df`

| Foo | Bar |
|---|---|
| 2021-04-29 01:39:00 | False |
| 2021-04-29 01:39:00 | False |
| NaT | True |

**DataFrame** `df.dtypes`

| Foo | Bar |
|---|---|
| datetime64[ns] | bool |

**class** api2db.ingest.post_process.date_cast.**DateCast**(*key: str*, *fmt: str*)
Bases: *api2db.ingest.post_process.post.Post*

Used to cast columns containing dates in string format to pandas DateTimes

**__init__**(*key: str*, *fmt: str*)
Creates a DateCast object

**Parameters**

- **key** – The name of the column containing strings that should be cast to datetimes
- **fmt** – A string formatter that specifies the datetime format of the strings in the column named `key`

**ctype**
type of data processor

**Type** str

**lam_wrap**(*lam_arg: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame
Overrides super class method

Workflow:

1. Attempt to cast `lam_arg[self.key]` from strings to datetimes
2. Return the modified `lam_arg`

**Parameters** **lam_arg** – The DataFrame to modify

**Returns** The modified DataFrame

### api2db.ingest.post_process.drop_na module

#### Contains the DropNa class

Simply a shortcut class for a common operation.

#### Summary of DropNa Usage:

See pandas Documentation

**class** api2db.ingest.post_process.drop_na.**DropNa**(*keys: List[str]*)
Bases: *api2db.ingest.post_process.post.Post*

Used to drop columns with null values on specified keys

**__init__**(*keys: List[str]*)
Creates a DropNa object

**Parameters** **keys** – The subset of keys to drop if the keys are null

**ctype**
type of data processor

**Type** str

**lam_wrap**(*lam_arg: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame
Overrides super class method

Shortcut used to drop null values. Performs `pd.DataFrame.drop_na(subset=self.keys)`

**Parameters** **lam_arg** – The DataFrame to modify

**Returns** The modified DataFrame

### api2db.ingest.post_process.merge_static module

### Contains the MergeStatic class

---

**Note:** MergeStatic is used to merge data together. A common use case of this is in situations where a data-vendor provides an API that gives data-points "Foo", "Bar", and "location_id" where "location_id" references a different data-set.

It is common for data-providers to have a file that does not update very frequently, i.e. is mostly static that contains this information.

The typical workflow of a MergeStatic instance is as follows:

1. Create a LocalStream with mode set to *update* or *replace* and a target like *CACHE/my_local_stream.pickle*
2. Set the LocalStream to run periodically (6 hours, 24 hours, 10 days, whatever frequency this data is updated)
3. Add a MergeStatic object to the frequently updating datas post-processors and set the path to the LocalStream storage path.

---

**class** api2db.ingest.post_process.merge_static.**MergeStatic**(*key: str*, *path: str*)
Bases: *api2db.ingest.post_process.post.Post*

Merges incoming data with a locally stored DataFrame

**__init__**(*key: str*, *path: str*)
Creates a MergeStatic object

**Parameters**

- **key** – The key that the DataFrames should be merged on
- **path** – The path to the locally stored file containing the pickled DataFrame to merge with

**ctype**
type of data processor

**Type** str

**lam_wrap**(*lam_arg: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame
Overrides super class method

Workflow:

1. Load DataFrame `df` from file specified at `self.path`
2. Use `lam_arg` to perform left-merge on `self.key` merging with `df`
3. Return the modified DataFrame

**Parameters** **lam_arg** – The DataFrame to modify

**Returns** The modified DataFrame

### api2db.ingest.post_process.post module

### Contains the Post class

**class** api2db.ingest.post_process.post.**Post**
Bases: *api2db.ingest.base_lam.BaseLam*

Used as a BaseClass for all PostProcessors

**static typecast**(*dtype: Any*) → str
Yields a string that can be used for typecasting to pandas dtype.

**Parameters** **dtype** – A python native type

**Returns** A string that can be used in conjunction with a pandas DataFrame/Series for typecasting

### Module contents

| Original Author | Tristen Harr |
|---|---|
| Creation Date | 04/29/2021 |
| Revisions | None |

### api2db.ingest.pre_process package

### Submodules

### api2db.ingest.pre_process.bad_row_swap module

### Contains the BadRowSwap class

---

**Note:** BadRowSwap should not be used until **AFTER** ListExtract has been performed on the data, unless performing a list extract is not necessary on the data.

When using BadRowSwap, the following conditions must be met:

1. The value contained at location `key_1` must be able to be identified as valid, or in need of being swapped without any reference to the value at location `key_2`. (Typically using regex or performing type-checking)
2. `key_1` and `key_2` must be unique within their respective row of data. `data = {"key_1": {"key_1": 1, "key_2": 2}}` would be invalid.

BadRowSwap **will potentially drop rows of data**. Rows meeting the following conditions will be dropped:

- Any row that is missing `key_1` as a key will be dropped.
- Any row that evaluates as needing to be swapped based on `key_1` that is missing `key_2` will be dropped.

BadRowSwap will keep rows that meet the following conditions:

- Any row that evaluates as not needing to be swapped based on `key_1` will be kept, regardless of if `key_2` exists or not.

- Any row that evaluates as needing to be swapped based on `key_1` that also contains `key_2` will swap the values at the locations of the `key_1` and `key_2` and the row will be kept.

Performing BadRowSwap can be computationally expensive, since it walks all nested data until it finds the desired keys. So here are a few tips to help you determine if you should be using it or not.

Usage Tips for using BadRowSwap:

- If both `key_1` and `key_2` are unimportant fields, I.e. Nullable fields and keeping them does not add significant value to the data consider just allowing the collector to Null them if they do not match the types or consider allowing them to simply have the wrong values if they have the same data-types. Otherwise you risk both slowing down data-collection, and dropping rows that have good data other than those swapped rows.
- Always attempt to place the key at location `key_1` as the more important value to retain. If you need to swap data like a "uuid" and a "description", use the "uuid" as `key_1`
- If you cannot place the key at location `key_1` as the more important key, consider if the risk of losing data with a valid value for the more important key is worth it in instances where the less important key is missing
- Consider the frequency that BadRowSwap would need to be run. If 1 out of 1,000,000 data-points contains values with swapped keys, is it worth running the computation on all 1,000,000 rows to save just that 1 row?
- Analyze the data by hand. Pull it into a pandas DataFrame, and check it.
  - How often are is a row incorrect?
  - Are the erroneous rows ALWAYS the same key?
  - How often is one of the keys for the row missing when the rows have bad data?

---

### Summary of BadRowSwap usage:

```
data = [
    {
        "id": "17.0",
        "size": "Foo",
    },
    {
        "id": "Bar",
        "size": "10.0"
    }
]

pre = BadRowSwap(key_1="id",
                 key_2="size",
                 lam=lambda x: re.match("[0-9][0-9]\.[0-9]+", x["id"]) is not None
                 )
```

### Example Usage of BadRowSwap:

Occasionally when dealing with an API, the data is not always where it is supposed to be. Oftentimes this results in the rows containing the misplaced data being dropped altogether. In the instance that for some unknown reason the incoming data has keys that tend to occasionally have their values swapped so long as it is possible to check to see if the data has been swapped due to what the data *should* be, use BadRowSwap.

This example assumes that the API occasionally swaps the values for "id" and "latitude". BadRowSwap can handle any level of nested data in these instances, so long as the keys for the values that are occasionally swapped are **unique within a single row**

```
>>> import re
... data = [
...     {
...         "id": "16.53",                                    # NEEDS SWAP = True
...          "place": {
...              "coords": {
...                  "latitude": "ID_1",
...                  "longitude": "-20.43"
...              },
...              "name": "place_1"
...          },
...         "details": "Some details... etc"
...      },
...
...     {
...         "id": "ID_2",                                     # NEEDS SWAP = False
...          "place": {
...              "coords": {
...                  "latitude": "15.43",
...                  "longitude": "-20.43"
...              },
...              "name": "place_2"
...          },
...         "details": "Some details... etc"
...      },
...
...     {
...         "id": "10.21",                                    # NEEDS SWAP = True
...         "place": {
...             "coords": {
...                                                           # Missing "latitude" key,␣
↪results in row being skipped
...                 "longitude": "-20.43"
...             },
...             "name": "place_2"
...         },
...         "details": "Some details... etc"
...     },
...
...     {
...                                                           # Missing "id" key,␣
↪results in row being skipped
...         "place": {
...             "coords": {
...                 "latitude": "ID_4",
...                 "longitude": "-20.43"
```

(continues on next page)

(continued from previous page)

```
...             },
...             "name": "place_2"
...         },
...         "details": "Some details... etc"
...     },
...
...     {
...         "id": "ID_5",                                       # NEEDS SWAP = False
...         "place": {
...             "coords": {
...                                                             # Missing "latitude" row␣
↪is kept, because no row swap needed
...                 "longitude": "-20.43"
...             },
...             "name": "place_2"
...         },
...         "details": "Some details... etc"
...     }
... ]
...
... pre = BadRowSwap(key_1="id",
...                  key_2="latitude",
...                  lam=lambda x: re.match("[0-9][0-9]\.[0-9]+", x["id"]) is not None
...                  )
...
... pre.lam_wrap(data)
[
    {
        "id": "ID_1",  # "id" and "latitude" have been swapped
        "place": {
            "coords": {
                "latitude": "16.53",
                "longitude": "-20.43"
            },
            "name": "place_1"
        },
        "details": "Some details... etc"
    },
    {
        "id": "ID_2",  # No changes required with this row
        "place": {
            "coords": {
                "latitude": "15.43",
                "longitude": "-20.43"
            },
            "name": "place_2"
        },
        "details": "Some details... etc"
    },
    # Row 3, and Row 4 have been dropped because they were missing key_1 or they␣
↪required a swap and were missing key_2
    {
        "id": "ID_5",  # No changes required with this row
        "place": {
            "coords": {
                # The latitude is still missing but that can be handled later, it may␣
↪be nullable, so it should be kept
```

(continues on next page)

(continued from previous page)

```
                "longitude": "-20.43"
            },
            "name": "place_2"
        },
        "details": "Some details... etc"
    }
]
```

**class** api2db.ingest.pre_process.bad_row_swap.**BadRowSwap**(*key_1: str*, *key_2: str*, *lam: Callable[[dict], bool]*)

Bases: *api2db.ingest.pre_process.pre.Pre*

Used to swap rows arriving from the API that have the values for the given key swapped occasionally

**__init__**(*key_1: str*, *key_2: str*, *lam: Callable[[dict], bool]*)

Creates a BadRowSwap object

**Parameters**

- **key_1** – The key of a field that occasionally has its value swapped with the data from key_2
- **key_2** – The key of a field that occasionally has its value swapped with the data from key_1
- **lam** – A function (anonymous, or not) that when given the value located under key_1 returns True if the keys need their values swapped otherwise returns False

**lam_wrap**(*lam_arg: List[dict]*) → List[dict]

Overrides super class method

**Parameters** **lam_arg** – A list of dictionaries with each dictionary containing what will become a row in a DataFrame

**Returns** A modified list of dictionaries with bad rows being either swapped or dropped.

### api2db.ingest.pre_process.feature_flatten module

### Contains the FeatureFlatten class

**Note:** FeatureFlatten should not be used until **AFTER** ListExtract has been performed on the data, unless performing a list extract is not necessary on the data.

### Summary of FeatureFlatten usage:

```
data = [
    {
        "data_id": 1,
        "data_features": [
                            {
                                "x": 5,
                                "y": 10
                            },
```

(continues on next page)

(continued from previous page)

```
                                {
                                    "x": 7,
                                    "y": 15
                                },
                                .
                                .
                                .
                            ]
    }
]
pre = FeatureFlatten(key="data_features")
```

### Example Usage of FeatureFlatten:

```
>>> data = [
...     {
...         "data_id": 1,
...         "data_features": {
...                             "Foo": 5,
...                             "Bar": 10
...                          }
...     },
...
...     {
...         "data_id": 2,
...         "data_features": [
...                             {
...                                 "Foo": 5,
...                                 "Bar": 10
...                             },
...                             {
...                                 "Foo": 7,
...                                 "Bar": 15
...                             }
...                          ]
...     }
... ]
... pre = FeatureFlatten(key="data_features")
... pre.lam_wrap(data)
[
    {
        "data_id": 1,
        "data_features": {
                            "Foo": 5,
                            "Bar": 10
                        }
    },
    {
        "data_id": 2,
        "data_features": {
                            "Foo": 5,
                            "Bar": 10
                        }
    },
```

(continues on next page)

(continued from previous page)

```
    {
        "data_id": 2,
        "data_features": {
                              "Foo": 7,
                              "Bar": 15
                          }
    }
]
```

**class** api2db.ingest.pre_process.feature_flatten.**FeatureFlatten**(*key: str*)

Bases: *api2db.ingest.pre_process.pre.Pre*

Used to flatten features containing arrays causing them to be incompatible for storage in a table-based schema

**__init__**(*key: str*)

Creates a FeatureFlatten object

**Parameters** **key** – The key containing nested data that each needs to have its own row in the final DataFrame

**ctype**

type of data processor

**Type** str

**lam_wrap**(*lam_arg: Optional[List[dict]]*) → List[dict]

Overrides super class method

Workflow:

- Create an array of `rows`
- For each dictionary `d` in the array of data-points
  - If the type of `self.key` is in `d.keys()` and `type(d[self.key]) == list`
    - For each item in list
      - Create a new row containing all data-features and the item by itself and add it to `rows`
  - If the type of `self.key` is in `d.keys()` and `type(d[self.key]) == dict`
    - Keep the row as it is, and add it to `rows`

**Parameters** **lam_arg** – A list of dictionaries that each represent a row in the final DataFrame (Optional to safeguard against if previous pre-processors could not parse data, i.e. No data-points existed)

**Returns** An array of dictionaries that each represent a row, with nested data extracted to their own rows

### api2db.ingest.pre_process.global_extract module

### Contains the GlobalExtract class

### Summary of GlobalExtract usage:

```
data = {"date": "2021-04-19", "data_array": [{"id": 1, "name": "Foo"}, {"id": 2, "name
↪": "Bar"}]}
pre = GlobalExtract(key="publish_time",
                    lam=lambda x: x["date"],
                    dtype=str
                    )
```

**Final DataFrame**

| id | name | publish_time |
|---|---|---|
| 1 | Foo | 2021-04-19 |
| 2 | Bar | 2021-04-19 |

### Example Usage of GlobalExtract:

```
>>> # pre-processing operators
... pres = []
... # Dictionary that contains all globally extracted data
... pre_2_post_dict = {}
... # Incoming Data
... data = {"date": "2021-04-19", "data_array": [{"id": 1, "name": "Foo"}, {"id": 2,
↪"name": "Bar"}]}
... # GlobalExtract instance for extracting the "date" from data, but replacing its␣
↪key with "publish_time"
... pre = GlobalExtract(key="publish_time",
...                     lam=lambda x: x["date"],
...                     dtype=str
...                     )
... # The preprocessor gets added to the list of preprocessors
... pres.append(pre)
... # Each preprocesser gets applied sequentially
... for p in pres:
...     if p.ctype == "global_extract":
...         pre_2_post_dict[p.key] = p.lam_wrap(data)
...     else:
...         pass # See other pre-processors
... pre_2_post_dict
{"publish_time": {"value": "2021-04-19", "dtype": str}}
```

**Later after the data has been extracted to a DataFrame df**

```
# Assume df = DataFrame containing extracted data
# Assume dtype_convert is a function that maps a python native type to a pandas dtype

# For each globally extracted item
for k, v in pre_2_post_dict.items():
    # Add the item to the DataFrame -> These are GLOBAL values shared amongst ALL rows
```

(continues on next page)

(continued from previous page)

```
    df[k] = v["value"]
    # Typecast the value to ensure it is the correct dtype
    df[k] = df[k].astype(dtype_convert(v["dtype"]))
```

**Example of what DataFrame would be:**

| id | name | publish_time |
|---|---|---|
| 1 | Foo | 2021-04-19 |
| 2 | Bar | 2021-04-19 |

**class** api2db.ingest.pre_process.global_extract.**GlobalExtract**(*key: str*, *lam: Callable[[dict], Any]*, *dtype*)

Bases: *api2db.ingest.pre_process.pre.Pre*

Used to extract a global feature from incoming data

**__init__**(*key: str*, *lam: Callable[[dict], Any]*, *dtype*)

Creates a GlobalExtract object

**Parameters**

- **key** – The desired key of the feature for the storage target
- **lam** – Anonymous function that specifies where the location of the feature that should be extracted is
- **dtype** – The python native datatype the feature is expected to be

**ctype**

type of the data processor

**Type** str

**lam_wrap**(*lam_arg: dict*) → dict

Overrides super class method

Workflow:

1. Attempt to perform the `lam` operation on the incoming data
2. Attempt to cast the result of the `lam` operation to the `dtype`
   - If an exception occurs, returns {"value": None, "dtype": `dtype`}
3. Return {"value": `result`, "dtype": `dtype`}

**Parameters** **lam_arg** – A dictionary containing the feature that should be extracted

**Returns** `result or None`, "dtype": `dtype`}

**Return type** A dictionary containing {"value"

### api2db.ingest.pre_process.list_extract module

### Contains the ListExtract class

### Summary of ListExtract Usage:

```
data = { "actual_data_rows": [{"id": "row1"}, {"id": "row2"}], "erroneous_data":
↪"FooBar" }
pre = ListExtract(lam=lambda x: x["actual_data_rows"])
```

### Example Usage of ListExtract:

```
>>> data = {
...     "Foo": "Metadata",
...     "data_array": [
...             {
...                 "data_id": 1,
...                 "name": "name_1"
...             },
...             {
...                 "data_id": 2,
...                 "name": "name_2"
...             }
...         ]
... }
...
... pre = ListExtract(lam=lambda x: x["data_array"])
... pre.lam_wrap(data)
[
    {
        "data_id": 1,
        "name": "name_1"
    },
    {
        "data_id": 2,
        "name": "name_2"
    }
]
```

**class** api2db.ingest.pre_process.list_extract.**ListExtract**(*lam: Callable[[dict], list]*)

Bases: *api2db.ingest.pre_process.pre.Pre*

Used to extract a list of dictionaries that will each represent a single row in a database

**__init__**(*lam: Callable[[dict], list]*)

Creates a ListExtract object

**Parameters** **lam** – Anonymous function that attempts to extract a list of data that will become rows in a DataFrame

**ctype**

type of data processor

**Type** str

**dtype**
the datatype performing *lam* should yield

**Type** type(list)

**lam_wrap**(*lam_arg: dict*) → Optional[List[dict]]
Overrides super class method

Workflow:

1. Attempt to perform the `lam` operation on the incoming data
2. Attempt to cast the result `lam` operation to a list
   - If an exception occurs, return None
3. Return the list of data

**Parameters** **lam_arg** – A dictionary containing a list of dictionaries that will become the rows of a DataFrame

**Returns** A list of dictionaries that will become the rows of a DataFrame if successful otherwise None

### api2db.ingest.pre_process.pre module

### Contains the Pre class

**class** api2db.ingest.pre_process.pre.**Pre**
Bases: *api2db.ingest.base_lam.BaseLam*

Direct subclass of BaseLam with no overriders, members, or methods. Exists solely for organizational purposes

### Module contents

| Original Author | Tristen Harr |
|---|---|
| Creation Date | 04/29/2021 |
| Revisions | None |

### Submodules

### api2db.ingest.api2pandas module

### Contains the Api2Pandas class

**class** api2db.ingest.api2pandas.**Api2Pandas**(*api_form: Callable[], api2db.ingest.api_form.ApiForm]*)
Bases: object

Used to extract incoming data from an API into a pandas DataFrame

**__init__**(*api_form: Callable[], api2db.ingest.api_form.ApiForm]*)
Creates a Api2Pandas object and loads its ApiForm

**Parameters** **api_form** – The function that generates the ApiForm for the associated collector

**dependencies_satisfied**() → bool
Checks to ensure any data-linking dependency files exist

This feature currently only exists for *api2db.ingest.post_process.merge_static.MergeStatic*

**Returns** True if all dependencies are satisfied, otherwise False

**extract**(*data: dict*) → Optional[pandas.core.frame.DataFrame]
Performs data-extraction from data arriving from an API.

Workflow:

1. Perform all pre-processing on data
2. Perform all data-feature extraction
3. Perform all post-processing on data
4. Return a DataFrame containing the cleaned data.

**Parameters** **data** – The data arriving from an API to perform data extraction on.

**Returns** The cleaned data if it is possible to clean the data otherwise None

### api2db.ingest.api_form module

### Contains the ApiForm class

**class** api2db.ingest.api_form.**ApiForm**(*name: str, pre_process: Optional[List[*api2db.ingest.pre_process.pre.Pre*]] = None, data_features: Optional[List[*api2db.ingest.data_feature.feature.Feature*]] = None, post_process: Optional[List[*api2db.ingest.post_process.post.Post*]] = None*)
Bases: object

Used to clean and process incoming data arriving from an Api

**__init__**(*name: str, pre_process: Optional[List[*api2db.ingest.pre_process.pre.Pre*]] = None, data_features: Optional[List[*api2db.ingest.data_feature.feature.Feature*]] = None, post_process: Optional[List[*api2db.ingest.post_process.post.Post*]] = None*)
Creates an ApiForm

---

**Note:** The ApiForm is used by api2db to do the processing and cleaning of data. Incoming data goes through 3 phases.

1. Pre-Processing
   - Extract global data-features
   - Extract a list of data-points that will serve as the rows in a database
   - Flatten nested arrays of data
   - Swap extraneous rows returned from poorly implemented APIs
2. Feature Extraction
   - Extracts the data features for each row that will be stored in a database

3. Post-Processing
    - Add new columns of data that will be the same globally for the arriving data. I.e. arrival timestamps
    - Apply functions across data columns, replacing the data with the calculated value. I.e. Reformat strings, strip whitespace, etc.
    - Add new columns of data that are derived from performing calculations on existing columns. I.e. Use a *latitude* and *longitude* column to calculate a new column called *country*
    - Cast columns that contain datetime data from strings to date times.
    - Drop columns that should not contain null values.
    - Perform merging of incoming data with locally stored reference tables. I.e. Incoming data has column *location_id* field, a reference table contains location info with the *location_id* field being a link between the two. This allows for data to be merged on column *location_id* in order to contain all data in a single table.

---

**Parameters**

- **name** – The name of the collector the ApiForm is associated with
- **pre_process** – An array pre-processing objects to be applied sequentially on incoming data
- **data_features** – An array of data features to be extracted from the incoming data. The programmer can choose which data features they require, and keep only those.
- **post_process** – An array of post-processing objects to be applied sequentially on the data after data has been cleaned and extracted to a *pandas.DataFrame*

**add_pre** (*pre:* api2db.ingest.pre_process.pre.Pre) → None
Allows the programmer to manually add a item to the pre-processing array.

**Parameters** **pre** – The pre-processing object to add

**Returns** None

**add_feature** (*feat:* api2db.ingest.data_feature.feature.Feature) → None
Allows the programmer to manually add a item to the data-features array.

**Parameters** **feat** – The feature object to add

**Returns** None

**add_post** (*post:* api2db.ingest.post_process.post.Post) → None
Allows the programmer to manually add a item to the post-processing array.

**Parameters** **post** – The post-processing object to add

**Returns** None

**pandas_typecast** () → dict
Performs typecasting from python native types to their pandas counterparts. Currently supported types are:

- int
- float
- bool
- str

Since API data is inconsistent, all typecasting makes the values nullable inside the DataFrame. Null values can be removed during post-processing.

**Returns** A dictionary that can be used to cast a DataFrames types using DataFrame.astype()

**static typecast** (*dtype: Any*) → str
Yields a string containing the pandas dtype when given a python native type.

**Parameters** **dtype** – The python native type

**Returns** The string representing the type that the native type converts to when put into a DataFrame

**experiment** (*CACHE*, *import_target*) → bool
Tool used to build an ApiForm

---

**Note:** The laboratory is an experimental feature and does not currently support the StaticMerge post-processor.

---

**Parameters**

- **CACHE** – If the data imports should be cached. I.e. Only call the API once
- **import_target** – The target function that performs an API import

**Returns** True if experiment is ready for export otherwise False

### api2db.ingest.base_lam module

### Contains the BaseLam class

**class** api2db.ingest.base_lam.**BaseLam**
Bases: object

Used as a Base object for pre-process subclasses, post-process subclasses, and data-features.

**__call__** (*lam_arg: Any*) → Any
Makes the class callable, with target of class method *lam_wrap* This is used to allow for anonymous functions to be passed to the class, and to enhance ease of use for library developers.

**Parameters** **lam_arg** – The argument to be passed to the *lam_wrap* class method.

**Returns** The response of the *lam_wrap* class method

**__getstate__** () → dict
Allows for lambda operations to be serialized in order to allow for instance to be passed between processes

**Returns** Customized self.__dict__ items with values serialized using the *dill* library

**__setstate__** (*state: dict*) → None
Allows for lambda operations to be deserialized using the *dill* library in order to allow for instance to be passed between processes

**Parameters** **state** – Incoming state

**Returns** None

**lam_wrap** (*lam_arg: Any*) → None
Method that performs class lambda method on lam_arg This method will **ALWAYS** be overriden.

**Parameters** **lam_arg** – The incoming data to perform the lambda operation on.

**Returns** None if attempting to call `BaseLam.lam_wrap`, return is dictated by subclasses.

### api2db.ingest.collector module

### Contains the Collector class

**class** api2db.ingest.collector.**Collector**(*name: str*, *seconds: int*, *import_target: Callable[], Optional[List[dict]]]*, *api_form: Callable[], api2db.ingest.api_form.ApiForm]*, *streams: Callable[], List[api2db.stream.stream.Stream]]*, *stores: Callable[], List[api2db.store.store.Store]]*, *debug: bool = True*)

Bases: `object`

Used for creating a data-collection pipeline from API to storage medium

**__init__**(*name: str*, *seconds: int*, *import_target: Callable[], Optional[List[dict]]]*, *api_form: Callable[], api2db.ingest.api_form.ApiForm]*, *streams: Callable[], List[api2db.stream.stream.Stream]]*, *stores: Callable[], List[api2db.store.store.Store]]*, *debug: bool = True*)

Creates a Collector object

---

**Note:** Project collectors are disabled by default, this allows the project to run immediately after `pmake` is run without any code being written. To enable a collector, you must change its `seconds` parameter to a number greater than zero. This represents the periodic interval that the collectors `import_target` is run. I.e. The collector will request data from its configured API every `seconds` seconds.

A perceptive user may notice that `import_target`, `api_form`, `streams` and `stores` appear to be written in seemingly extraneous functions. Why not just pass in the actual data directly to the Collector object? This occurs due to the extensive use of anonymous functions which is what allows the library to be so expressive. Python's native serialization does not support serializing lambdas. When using the multiprocessing module and spawning a new process the parameters of the process are serialized before being piped into a new python interpreter instance. It is for this reason that functions are used as parameters rather than their returns, since it is possible to pass a function which *will* instantiate an anonymous function upon call, but not to pass an existing anonymous function to a separate process. Feel free to write a supporting package to make it so this is not the case.

---

**Parameters**

- **name** – The name of the collector, this name will be set when using `pmake` or `cadd` and **should not be changed**. Changing this may result in unintended functionality of the api2db library, as this name is used when determining where to store incoming data, what to name database tables, and the location of the dtypes file which gets stored in the projects *CACHE/* directory. If you wish to change the name of a collector, you can run `cadd` to add a new collector with the desired name, and then move the code from the old collector into the new collector.
- **seconds** – This specifies the periodic interval that data should be imported at. I.e. `seconds=30` will request data from the collector api every 30 seconds. This is set to *0* by default, and when set to *0* the collector is disabled and will not be registered with the main program. This allows for all neccesary collectors to be added to a project, and then for each collector to be enabled as its code is written.

- **import_target** – The import_target is the function that the programmer using the library writes that performs the initial data import. In most cases this will utilize a library like *urllib* in order to perform the requests. The return of this function should be a list of dictionary objects.
  - When dealing with XML data use a library like *xmltodict* to convert the data to a python dictionary
  - When dealing with JSON data use a library like the built-in *json* library to convert the data to a python dictionary.

  The implementation of this method is left to the programmer. This method could also be written to collect data from a serial stream, or a web-scraper if desired. Design and implementation of things such as that are left to the users of the library.

  The `import_target` **MUST return a list of dictionaries, or None**. Exceptions that may occur within the function must be handled. The purpose of this implementation is to allow for logic to be written to perform multiple API requests and treat the data as a single incoming request. Most APIs will return a single response, and if the implementation of the `import_target` does not make multiple API calls then simply wrap that data in a list when returning it from the function.
- **api_form** – This is a function that returns an API form.
- **streams** – This is a function that returns a list of Stream object subclasses.
- **stores** – This is a function that returns a list of Store object subclasses.
- **debug** – When set to True logs will be printed to the console. Set to False for production.

**q**
A queue used for message passing if collector is running in debug mode

**Type** Optional[multiprocessing.Queue]

**set_q**(*q: multiprocessing.context.BaseContext.Queue*) → None
Sets the q class member used for collectors running in debug mode

**Parameters** **q** – The queue used for message passing

**Returns** None

## Module contents

| Original Author | Tristen Harr |
|---|---|
| Creation Date | 04/28/2021 |
| Revisions | None |

### api2db.install package

### Submodules

### api2db.install.clear_lab module

api2db.install.clear_lab.**clab**()
This shell command is used to clear a lab.

```
project_dir-----/
                |
                apis-------/
                |           |- __init__.py
                |           |- FooCollector.py
                |           |- BarCollector.py
                |
                AUTH-------/
                |           |- bigquery_auth_template.json
                |           |- omnisci_auth_template.json
                |           |- sql_auth_template.json
                |
                CACHE/
                |
                STORE/
                |
                laboratory-/
                |           |- lab.py    EDIT THIS FILE!
                |
                helpers.py
                |
                main.py
```

**Shell Command:** `path/to/project_dir> clab`

```
project_dir-----/
              |
              apis-----/
              |         |- __init__.py
              |         |- FooCollector.py
              |         |- BarCollector.py
              |
              AUTH-----/
              |         |- bigquery_auth_template.json
              |         |- omnisci_auth_template.json
              |         |- sql_auth_template.json
              |
              CACHE/
              |
              STORE/
              |
              helpers.py
              |
              main.py
```

### api2db.install.collector_add module

### Contains the cadd function

api2db.install.collector_add.**cadd**(*ad: str*) → None
This shell command is used to add a collector to an existing api2db project

Given the following project structure

```
project_dir-----/
                |
                apis-----/
                |        |- __init__.py
                |        |- FooCollector.py
                |
                AUTH-----/
                |        |- bigquery_auth_template.json
                |        |- omnisci_auth_template.json
                |        |- sql_auth_template.json
                |
                CACHE/
                |
                STORE/
                |
                helpers.py
                |
                main.py
```

**Shell Command:** `path/to/procect_dir> cadd BarCollector`

```
project_dir-----/
                |
                apis-----/
                |        |- __init__.py
                |        |- FooCollector.py
                |        |- BarCollector.py
                |
                AUTH-----/
                |        |- bigquery_auth_template.json
                |        |- omnisci_auth_template.json
                |        |- sql_auth_template.json
                |
                CACHE/
                |
                STORE/
                |
                helpers.py
                |
                main.py
```

**Parameters** **ad** – The name of the collector to add.

**Returns** None

### api2db.install.collector_list module

### Contains the clist function

api2db.install.collector_list.**clist**() → None
This shell command is used to show a list of collectors registered with an existing api2db project

Given the following project

```
project_dir-----/
                |
                apis-----/
                |        |- __init__.py
                |        |- FooCollector.py
                |        |- BarCollector.py
                |
                AUTH-----/
                |        |- bigquery_auth_template.json
                |        |- omnisci_auth_template.json
                |        |- sql_auth_template.json
                |
                CACHE/
                |
                STORE/
                |
                helpers.py
                |
                main.py
```

**Shell Command:** `path/to/procect_dir> clist`

**Out:** `["FooCollector", "BarCollector"]`

**Returns** None

### api2db.install.collector_remove module

### Contains the crem function

api2db.install.collector_remove.**crem**(*rem: str*) → None
This shell command is used to remove a collector registered with an existing api2db project

Given the following project

```
project_dir-----/
                |
                apis-----/
                |        |- __init__.py
                |        |- FooCollector.py
                |        |- BarCollector.py
                |
                AUTH-----/
                |        |- bigquery_auth_template.json
                |        |- omnisci_auth_template.json
                |        |- sql_auth_template.json
                |
```

(continues on next page)

(continued from previous page)

```
                CACHE/
                |
                STORE/
                |
                helpers.py
                |
                main.py
```

**Shell Command:** `path/to/project_dir> crem BarCollector`

```
project_dir-----/
                |
                apis-----/
                |        |- __init__.py
                |        |- FooCollector.py
                |
                AUTH-----/
                |        |- bigquery_auth_template.json
                |        |- omnisci_auth_template.json
                |        |- sql_auth_template.json
                |
                CACHE/
                |
                STORE/
                |
                helpers.py
                |
                main.py
```

**Parameters** **rem** – The name of the collector to remove

**Returns** None

### api2db.install.make_lab module

api2db.install.make_lab.**mlab**()

This shell command is used for creation of a lab. Labs offer an easier way to design an ApiForm.

Given a project directory

```
project_dir-----/
                |
                apis-----/
                |        |- __init__.py
                |        |- FooCollector.py
                |        |- BarCollector.py
                |
                AUTH-----/
                |        |- bigquery_auth_template.json
                |        |- omnisci_auth_template.json
                |        |- sql_auth_template.json
                |
                CACHE/
                |
                STORE/
```

(continues on next page)

(continued from previous page)

```
                |
                helpers.py
                |
                main.py
```

**Shell Command:** `path/to/project_dir> mlab`

```
project_dir-----/
                |
                apis-------/
                |           |- __init__.py
                |           |- FooCollector.py
                |           |- BarCollector.py
                |
                AUTH-------/
                |           |- bigquery_auth_template.json
                |           |- omnisci_auth_template.json
                |           |- sql_auth_template.json
                |
                CACHE/
                |
                STORE/
                |
                laboratory-/
                |           |- lab.py    EDIT THIS FILE!
                |
                helpers.py
                |
                main.py
```

**Returns** None

### api2db.install.project_clear module

### Contains the pclear function

**Warning:** Usage of this will completely clear out the project directory. This includes all collectors, all code, and all files. This is a nuclear delete option for when your foo doesn't want to bar and so you need to start over. **Use with caution.**

api2db.install.project_clear.**pclear**() → None

This shell command is used to clear a project and should **ONLY** be used if a complete restart is required.

Given the following project

```
project_dir-----/
                |
                apis-----/
                |        |- __init__.py
                |        |- FooCollector.py
                |        |- BarCollector.py
                |
```

(continues on next page)

(continued from previous page)

```
                AUTH-----/
                |        |- bigquery_auth_template.json
                |        |- omnisci_auth_template.json
                |        |- sql_auth_template.json
                |
                CACHE/
                |
                STORE/
                |
                helpers.py
                |
                main.py
```

**Shell Command:** `path/to/project_dir> pclear`

```
project_dir-----/
```

**Returns** None

### api2db.install.project_make module

### Contains the pmake function

api2db.install.project_make.**pmake**(*apis: Optional[List[str]] = None*) → None

This shell command is used for initial creation of the project structure.

Given a blank project directory

```
project_dir-----/
```

**Shell Command:** `path/to/project_dir> pmake FooCollector BarCollector`

```
project_dir-----/
                |
                apis-----/
                |        |- __init__.py
                |        |- FooCollector.py
                |        |- BarCollector.py
                |
                AUTH-----/
                |        |- bigquery_auth_template.json
                |        |- omnisci_auth_template.json
                |        |- sql_auth_template.json
                |
                CACHE/
                |
                STORE/
                |
                helpers.py
                |
                main.py
```

**Note:** This command can also be used without any collector arguments, and collectors can be added using the

`cadd` shell command.

**Parameters** **apis** – The collector names that should be created for the project APIs

**Returns** None

### api2db.install.run_lab module

`api2db.install.run_lab.`**rlab**()
This shell command is used to run a lab.

### Module contents

| Original Author | Tristen Harr |
|---|---|
| Creation Date | 04/28/2021 |
| Revisions | None |

### api2db.store package

### Submodules

### api2db.store.store module

### Contains the Store class

**class** api2db.store.store.**Store**(*name: str*, *seconds: int*, *path: Optional[str] = None*, *fmt: str = 'parquet'*, *drop_duplicate_exclude: Optional[List[str]] = None*, *move_shards_path: Optional[str] = None*, *move_composed_path: Optional[str] = None*, *chunk_size: int = 0*)
Bases: *api2db.stream.stream.Stream*

Used for storing data into a local or external source periodically

**__init__**(*name: str*, *seconds: int*, *path: Optional[str] = None*, *fmt: str = 'parquet'*, *drop_duplicate_exclude: Optional[List[str]] = None*, *move_shards_path: Optional[str] = None*, *move_composed_path: Optional[str] = None*, *chunk_size: int = 0*)
Creates a Store object and attempts to build its dtypes.

**Parameters**

- **name** – The name of the collector the store is associated with
- **seconds** – The number of seconds between storage cycles
- **path** – The path to the directory that will contain sharded files that should be recomposed for storage
- **fmt** – The file format of the sharded files
  - *fmt="parquet"* (recommended) stores the DataFrame using parquet format
  - *fmt="json"* stores the DataFrame using JSON format

- *fmt="pickle"* stores the DataFrame using pickle format
- *fmt="csv"* stores the DataFrame using csv format

- **drop_duplicate_exclude** –
  - *drop_duplicate_exclude=None*

    DataFrame.drop_duplicates() performed before storage
  - *drop_duplicate_exclude=["request_millis"]*

    .drop_duplicates(subset=df.columns.difference(drop_duplicate_exclude)) performed before storage.

    *Primarily used for arrival timestamps. I.e. API sends the same data on sequential requests but in most applications the programmer will want to timestamp the arrival time of data, which would lead to duplicate data with the only difference being arrival timestamps*
- **move_shards_path** – *Documentation and Examples found here*
- **move_composed_path** – *Documentation and Examples found here*
- **chunk_size** – CURRENTLY NOT SUPPORTED

**store_str**
A string used for logging

**Type** Optional[str]

**build_dependencies**() → None
Builds the dependencies for the storage object. I.e. Makes the directories for the `move_shards_path` and the `move_composed_path`

**Returns** None

**store**() → None
Composed a DataFrame from the files in the stores path, and stores the data to the storage target.

**Returns** None

**start**()
Store objects subclass Stream but do not contain a start method. Stores should NEVER use start

**Raises** **AttributeError** – 'Store' object has no attribute 'start'

**stream_start**()
Store objects subclass Stream but do not contain a stream_start method. Stores should NEVER use stream_start

**Raises** **AttributeError** – 'Store' object has no attribute 'stream_start'

### api2db.store.store2bigquery module

### Contains the Store2Bigquery class

**class** api2db.store.store2bigquery.**Store2Bigquery**(*name: str*, *seconds: int*, *auth_path: str*, *pid: str*, *did: str*, *tid: str*, *path: Optional[str] = None*, *fmt: str = 'parquet'*, *drop_duplicate_exclude: Optional[List[str]] = None*, *move_shards_path: Optional[str] = None*, *move_composed_path: Optional[str] = None*, *location: str = 'US'*, *if_exists: str = 'append'*, *chunk_size: int = 0*)

Bases: *api2db.store.store.Store*

Used for storing data to bigquery periodically

**__init__**(*name: str*, *seconds: int*, *auth_path: str*, *pid: str*, *did: str*, *tid: str*, *path: Optional[str] = None*, *fmt: str = 'parquet'*, *drop_duplicate_exclude: Optional[List[str]] = None*, *move_shards_path: Optional[str] = None*, *move_composed_path: Optional[str] = None*, *location: str = 'US'*, *if_exists: str = 'append'*, *chunk_size: int = 0*)

Creates a Store2Bigquery object and attempts to build its dtypes.

**Parameters**

- **name** – The name of the collector the store is associated with
- **seconds** – The number of seconds between storage cycles
- **auth_path** – The path to the Google provided authentication file. I.e. AUTH/google_auth_file.json
- **pid** – Google project ID
- **did** – Google dataset ID
- **tid** – Google table ID
- **path** – The path to the directory that will contain sharded files that should be recomposed for storage
- **fmt** – The file format of the sharded files
  - *fmt="parquet"* (recommended) loads the sharded files using parquet format
  - *fmt="json"* loads the sharded files using JSON format
  - *fmt="pickle"* loads the sharded files using pickle format
  - *fmt="csv"* loads the sharded files using csv format
- **drop_duplicate_exclude** –
  - *drop_duplicate_exclude=None*

    DataFrame.drop_duplicates() performed before storage
  - *drop_duplicate_exclude=["request_millis"]*

    .drop_duplicates(subset=df.columns.difference(drop_duplicate_exclude)) performed before storage.

    *Primarily used for arrival timestamps. I.e. API sends the same data on sequential requests but in most applications the programmer will want to timestamp the arrival*

*time of data, which would lead to duplicate data with the only difference being arrival timestamps*

- **move_shards_path** – *Documentation and Examples found here*
- **move_composed_path** – *Documentation and Examples found here*
- **location** – Location of the Bigquery project
- **if_exists** –
  - *if_exists="append"* Adds the data to the table
  - *if_exists="replace"* Replaces the table with the new data
  - *if_exists="fail"* Fails to upload the new data if the table exists
- **chunk_size** – CURRENTLY NOT SUPPORTED

### api2db.store.store2omnisci module

### Contains the Store2Omnisci class

**class** api2db.store.store2omnisci.**Store2Omnisci**(*name: str*, *seconds: int*, *db_name: str*, *username: Optional[str] = None*, *password: Optional[str] = None*, *host: Optional[str] = None*, *auth_path: Optional[str] = None*, *path: Optional[str] = None*, *fmt: str = 'parquet'*, *drop_duplicate_exclude: Optional[List[str]] = None*, *move_shards_path: Optional[str] = None*, *move_composed_path: Optional[str] = None*, *protocol: str = 'binary'*, *chunk_size: int = 0*)

Bases: *api2db.store.store.Store*

Used for storing data to omnisci periodically

**__init__**(*name: str*, *seconds: int*, *db_name: str*, *username: Optional[str] = None*, *password: Optional[str] = None*, *host: Optional[str] = None*, *auth_path: Optional[str] = None*, *path: Optional[str] = None*, *fmt: str = 'parquet'*, *drop_duplicate_exclude: Optional[List[str]] = None*, *move_shards_path: Optional[str] = None*, *move_composed_path: Optional[str] = None*, *protocol: str = 'binary'*, *chunk_size: int = 0*)

Creates a Store2Omnisci object and attempts to build its dtypes.

**Note:** *See documentation for Stream2Omnisci*

**Parameters**

- **name** – The name of the collector the store is associated with
- **seconds** – The number of seconds between storage cycles
- **db_name** – The name of the database to connect to
- **username** – The username to authenticate with the database
- **password** – The password to authenticate with the database

- **host** – The host of the database
- **auth_path** – The path to the authentication credentials.
- **path** – The path to the directory that will contain sharded files that should be recomposed for storage
- **fmt** – The file format of the sharded files
  - *fmt="parquet"* (recommended) loads the sharded files using parquet format
  - *fmt="json"* loads the sharded files using JSON format
  - *fmt="pickle"* loads the sharded files using pickle format
  - *fmt="csv"* loads the sharded files using csv format
- **drop_duplicate_exclude** –
  - *drop_duplicate_exclude=None*

    DataFrame.drop_duplicates() performed before storage
  - *drop_duplicate_exclude=["request_millis"]*

    .drop_duplicates(subset=df.columns.difference(drop_duplicate_exclude)) performed before storage.

    *Primarily used for arrival timestamps. I.e. API sends the same data on sequential requests but in most applications the programmer will want to timestamp the arrival time of data, which would lead to duplicate data with the only difference being arrival timestamps*
- **move_shards_path** – *Documentation and Examples found here*
- **move_composed_path** – *Documentation and Examples found here*
- **protocol** – The protocol to use when connecting to the database
- **chunk_size** – CURRENTLY NOT SUPPORTED

### api2db.store.store2sql module

### Contains the Store2Sql class

**class** api2db.store.store2sql.**Store2Sql**(*name: str*, *seconds: int*, *db_name: str*, *dialect: str*, *username: Optional[str] = None*, *password: Optional[str] = None*, *host: Optional[str] = None*, *auth_path: Optional[str] = None*, *port: str = ''*, *path: Optional[str] = None*, *fmt: str = 'parquet'*, *drop_duplicate_exclude: Optional[List[str]] = None*, *move_shards_path: Optional[str] = None*, *move_composed_path: Optional[str] = None*, *if_exists: str = 'append'*, *chunk_size: int = 0*)

Bases: *api2db.store.store.Store*

Used for storing data to an SQL database periodically

**__init__**(*name: str*, *seconds: int*, *db_name: str*, *dialect: str*, *username: Optional[str] = None*, *password: Optional[str] = None*, *host: Optional[str] = None*, *auth_path: Optional[str] = None*, *port: str = ''*, *path: Optional[str] = None*, *fmt: str = 'parquet'*, *drop_duplicate_exclude: Optional[List[str]] = None*, *move_shards_path: Optional[str] = None*, *move_composed_path: Optional[str] = None*, *if_exists: str = 'append'*, *chunk_size: int = 0*)

Creates a Store2Sql object and attempts to build its dtypes.

---

**Note:** *See documentation for Stream2Sql*

---

**Parameters**

- **name** – The name of the collector the store is associated with
- **seconds** – The number of seconds between storage cycles
- **db_name** – The name of the database to connect to
- **dialect** –
  - *dialect="mysql"* -> Use this to connect to a mysql database
  - *dialect="mariadb"* -> Use this to connect to a mariadb database
  - *dialect="postgresql"* -> Use this to connect to a postgresql database
  - *dialect="amazon_aurora"* -> COMING SOON
  - *dialect="oracle"* -> COMING SOON
  - *dialect="microsoft_sql"* -> COMING SOON
  - *dialect="Something else?"* -> Submit a feature request... or even better build it!
- **username** – The username to authenticate with the database
- **password** – The password to authenticate with the database
- **host** – The host of the database
- **auth_path** – The path to the authentication credentials.
- **port** – The port to connect to the database with
- **path** – The path to the directory that will contain sharded files that should be recomposed for storage
- **fmt** – The file format of the sharded files
  - *fmt="parquet"* (recommended) loads the sharded files using parquet format
  - *fmt="json"* loads the sharded files using JSON format
  - *fmt="pickle"* loads the sharded files using pickle format
  - *fmt="csv"* loads the sharded files using csv format
- **drop_duplicate_exclude** –
  - *drop_duplicate_exclude=None*

    DataFrame.drop_duplicates() performed before storage
  - *drop_duplicate_exclude=["request_millis"]*

    .drop_duplicates(subset=df.columns.difference(drop_duplicate_exclude)) performed before storage.

*Primarily used for arrival timestamps. I.e. API sends the same data on sequential requests but in most applications the programmer will want to timestamp the arrival time of data, which would lead to duplicate data with the only difference being arrival timestamps*

- **move_shards_path** – *Documentation and Examples found here*
- **move_composed_path** – *Documentation and Examples found here*
- **chunk_size** – CURRENTLY NOT SUPPORTED

### Module contents

| Original Author | Tristen Harr |
|---|---|
| Creation Date | 04/28/2021 |
| Revisions | None |

### api2db.stream package

### Submodules

### api2db.stream.file_converter module

### Contains the FileConverter class

**class** api2db.stream.file_converter.**FileConverter**(*name: Optional[str] = None*, *dtypes: Optional[dict] = None*, *path: Optional[str] = None*, *fmt: Optional[str] = None*)

Bases: object

Serves as a base-class for all Streams/Stores and is used to store/load pandas DataFrames to different formats

**__init__**(*name: Optional[str] = None*, *dtypes: Optional[dict] = None*, *path: Optional[str] = None*, *fmt: Optional[str] = None*)

Creates a FileConverter object and attempts to build its dtypes

**Parameters**

- **name** – The name of the collector associated with the FileConverter
- **dtypes** – The dictionary of dtypes for the collector associated with the FileConverter
- **path** – Either a path to a file, or a path to a directory, dictated by super class
- **fmt** –
  - *fmt="parquet"* (recommended) sets the FileConverter format to use parquet format
  - *fmt="json"* sets the FileConverter format to use JSON format
  - *fmt="pickle"* sets the FileConverter format to use pickle format
  - *fmt="csv"* sets the FileConverter format to use CSV format

**build_dtypes**() → Optional[dict]

Attempts to build the dtypes so that a loaded pandas DataFrame can be type-casted

**Returns** dtypes that can be used with pandas.DataFrame.astype(dtypes)

**static static_compose_df_from_dir**(*path: str*, *fmt: str*, *move_shards_path: Optional[str] = None*, *move_composed_path: Optional[str] = None*, *force: bool = True*) → Optional[pandas.core.frame.DataFrame]

Attempts to build a single DataFrame from all files in a directory.

**Parameters**

- **path** – The directory path to compose files from
- **fmt** –
  - *fmt="parquet"* (recommended) stores the DataFrame using parquet format
  - *fmt="json"* stores the DataFrame using JSON format
  - *fmt="pickle"* stores the DataFrame using pickle format
  - *fmt="csv"* stores the DataFrame using csv format
- **move_shards_path** – The path to move the file shards to after composing the DataFrame
- **move_composed_path** – The path to move the composed shards to, with naming schema of filename1_filenameN.fmt
- **force** – Forces creation of the directories to move files to if they do not exist.

**Returns** The DataFrame composed from the sharded files if successful, else None

### Example

**Original Directory Structure**

```
store/
|    |- file1.parquet
|    |- file2.parquet
|    |- file3.parquet
|
sharded/
|
composed/
|
main.py
```

The following files contain pandas DataFrames stored using parquet format.

**file1.parquet**

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |

**file2.parquet**

| A | B | C |
|---|---|---|
| 4 | 5 | 6 |

**file3.parquet**

| A | B | C |
|---|---|---|
| 7 | 8 | 9 |

```
>>> FileConverter.static_compose_df_from_dir(path="store/", fmt="parquet")
```

**Returns (pandas.DataFrame):**

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

**By default, files will be deleted when the DataFrame is returned**

```
FileConverter.static_compose_df_from_dir(path="store/",
                                         fmt="parquet",
                                         move_shards_path=None,
                                         move_composed_path=None)
```

```
store/
|
sharded/
|
composed/
|
main.py
```

`move_shards_path` **specifies the path the sharded files should be moved to**

```
FileConverter.static_compose_df_from_dir(path="store/",
                                         fmt="parquet",
                                         move_shards_path="sharded/",
                                         move_composed_path=None)
```

```
store/
|
sharded/
|       |- file1.parquet
|       |- file2.parquet
|       |- file3.parquet
|
composed/
|
main.py
```

`move_composed_path` **speficies the path that the recomposed files should be moved to**

```
FileConverter.static_compose_df_from_dir(path="store/",
                                         fmt="parquet",
                                         move_shards_path=None,
                                         move_composed_path="composed/")
```

```
store/
|
```

(continues on next page)

(continued from previous page)

```
sharded/
|
composed/
|       |- file1_file3.parquet
|
main.py
```

**static static_store_df**(*df: pandas.core.frame.DataFrame*, *path: str*, *fmt: str*) → bool
Stores a DataFrame to a file

**Parameters**

- **df** – The DataFrame to store to a file
- **path** – The path to the file the DataFrame should be stored in
- **fmt** –
  - *fmt="parquet"* (recommended) stores the DataFrame using parquet format
  - *fmt="json"* stores the DataFrame using JSON format
  - *fmt="pickle"* stores the DataFrame using pickle format
  - *fmt="csv"* stores the DataFrame using csv format

**Returns** True if successful, else False

**static pickle_store**(*path: str*, *df: pandas.core.frame.DataFrame*, *force: bool = True*) → bool
Stores a DataFrame as a .pickle file

**Parameters**

- **path** – The path to store the DataFrame to
- **df** – The DataFrame to store
- **force** – If the directories in the path do not exist, forces them to be created

**Returns** True if successful, otherwise False

**static json_store**(*path: str*, *df: pandas.core.frame.DataFrame*, *force: bool = True*) → bool
Stores a DataFrame as a .json file

**Parameters**

- **path** – The path to store the DataFrame to
- **df** – The DataFrame to store
- **force** – If the directories in the path do not exist, forces them to be created

**Returns** True if successful, otherwise False

**static csv_store**(*path: str*, *df: pandas.core.frame.DataFrame*, *force: bool = True*) → bool
Stores a DataFrame as a .csv file

**Parameters**

- **path** – The path to store the DataFrame to
- **df** – The DataFrame to store
- **force** – If the directories in the path do not exist, forces them to be created

**Returns** True if successful, otherwise False

**static parquet_store**(*path: str*, *df: pandas.core.frame.DataFrame*, *force: bool = True*) → bool
Stores a DataFrame as a .parquet file

**Parameters**

- **path** – The path to store the DataFrame to
- **df** – The DataFrame to store
- **force** – If the directories in the path do not exist, forces them to be created

**Returns** True if successful, otherwise False

**static store_valid**(*path: str*, *force: bool*) → bool
Determines if a provided path for storage is valid. I.e. The directory structure exists

**Parameters**

- **path** – The path to check
- **force** – When True, will attempt to create necessary directories if they do not exist

**Returns** True if path is valid, otherwise False

**static static_load_df**(*path: str*, *fmt: str*, *dtypes: Optional[dict] = None*) → Optional[pandas.core.frame.DataFrame]
Loads a DataFrame from a file

**Parameters**

- **path** – The path to the file the DataFrame should be loaded from
- **fmt** –
  - *fmt="parquet"* (recommended) loads the DataFrame using parquet format
  - *fmt="json"* loads the DataFrame using JSON format
  - *fmt="pickle"* loads the DataFrame using pickle format
  - *fmt="csv"* loads the DataFrame using csv format
- **dtypes** – The dtypes to cast the DataFrame to before returning it. I.e. DataFrame.astype(dtypes)

**Returns** Loaded DataFrame if successful, otherwise None

**static pickle_load**(*path: str*) → Optional[pandas.core.frame.DataFrame]
Loads a DataFrame from a .pickle file

**Parameters** **path** – The path to load the DataFrame from

**Returns** The loaded DataFrame if successful, otherwise None

**static json_load**(*path: str*) → Optional[pandas.core.frame.DataFrame]
Loads a DataFrame from a .json file

**Parameters** **path** – The path to load the DataFrame from

**Returns** The loaded DataFrame if successful, otherwise None

**static csv_load**(*path: str*) → Optional[pandas.core.frame.DataFrame]
Loads a DataFrame from a .csv file

**Parameters** **path** – The path to load the DataFrame from

**Returns** The loaded DataFrame if successful, otherwise None

**static parquet_load**(*path: str*) → Optional[pandas.core.frame.DataFrame]
Loads a DataFrame from a .parquet file

**Parameters** **path** – The path to load the DataFrame from

**Returns** The loaded DataFrame if successful, otherwise None

### api2db.stream.stream module

### Contains the Stream class

**class** api2db.stream.stream.**Stream**(*name: str*, *path: Optional[str] = None*, *dtypes: Optional[dict] = None*, *fmt: Optional[str] = None*, *chunk_size: int = 0*, *stream_type: str = 'stream'*, *store: bool = False*)
Bases: *api2db.stream.file_converter.FileConverter*

Used for streaming data into a local or external source

**__init__**(*name: str*, *path: Optional[str] = None*, *dtypes: Optional[dict] = None*, *fmt: Optional[str] = None*, *chunk_size: int = 0*, *stream_type: str = 'stream'*, *store: bool = False*)
Creates a Stream object and attempts to build its dtypes. If store flag is false, spawns a thread that polls the Stream queue for incoming data

**Parameters**

- **name** – The name of the collector the stream is associated with
- **path** – The directory path the stream should store to (Usage dictated by super classes)
- **dtypes** – A dictionary containing the dtypes that the stream data DataFrame has
- **fmt** – The file format that the stream data should be stored as
  - *fmt="parquet"* (recommended) stores the DataFrame using parquet format
  - *fmt="json"* stores the DataFrame using JSON format
  - *fmt="pickle"* stores the DataFrame using pickle format
  - *fmt="csv"* stores the DataFrame using csv format
- **chunk_size** – The size of chunks to send to the stream target. I.e. Insert data in chunks of chunk_size rows
- **stream_type** – The type of the stream (Primarily used for logging)
- **store** – This flag indicates whether or not the stream is being called by a Store object

**Raises** **NotImplementedError** – chunk_storage is not yet implemented.

**is_store_instance**
True if the super-class has base-class Store otherwise False

**Type** bool

**lock**
Stream Lock used to signal if the stream has died

**Type** threading.Lock

**q**
Stream queue used to pass data into

**Type** queue.Queue

**start**() → None
Starts the stream running loop in a new thread

**Returns** None

**check_failures**() → None
Checks to see if previous uploads have failed and if so, loads the previous upload data and attempts to upload it again.

This method searches the directory path

STORE/upload_failed/**collector_name/stream_type/**

This path is the target location for failed uploads. If an upload fails 5 times in a row, it is stored in this location with the filename being the timestamp it is stored.

**Returns** None

**stream_start**() → None
Starts the stream listener that polls the stream queue for incoming data.

**Returns** None

**stream**(*data: pandas.core.frame.DataFrame*) → AttributeError
Overridden by supers, a Stream object is NEVER directly used to stream data. It is ALWAYS inherited from

**Parameters** **data** – The data to stream

**Raises** **AttributeError** – *Stream* does not have the ability to stream data. It must be subclassed.

### api2db.stream.stream2bigquery module

### Contains the Stream2Bigquery class

**class** api2db.stream.stream2bigquery.**Stream2Bigquery**(*name: str*, *auth_path: str*, *pid: str*, *did: str*, *tid: str*, *location: str = 'US'*, *if_exists: str = 'append'*, *chunk_size: int = 0*, *store: bool = False*)
Bases: *api2db.stream.stream.Stream*

Streams data from the associated collector directly to Bigquery in real-time

**__init__**(*name: str*, *auth_path: str*, *pid: str*, *did: str*, *tid: str*, *location: str = 'US'*, *if_exists: str = 'append'*, *chunk_size: int = 0*, *store: bool = False*)
Creates a Stream2Bigquery object and attempts to build its dtypes.

If dtypes can successfully be created I.e. Data arrives from the API for the first time the following occurs:

- Auto-generates the table schema
- Creates the dataset if it does not exist within the project
- Creates the table if it does not exist within the project

**Parameters**

- **name** – The name of the collector associated with the stream
- **auth_path** – The path to the Google provided authentication file. I.e. AUTH/google_auth_file.json

- **pid** – Google project ID
- **did** – Google dataset ID
- **tid** – Google table ID
- **location** – Location of the Bigquery project
- **if_exists** –
  - *if_exists="append"* Adds the data to the table
  - *if_exists="replace"* Replaces the table with the new data
  - *if_exists="fail"* Fails to upload the new data if the table exists
- **chunk_size** – CURRENTLY NOT SUPPORTED
- **store** – True if the super class is a Store object, otherwise False

**schema**
contains schema if dtypes exist else None

**Type** Optional[List[google.cloud.bigquery.SchemaField]]

**bq_schema**
contains bq_schema if dtypes exist else None

**Type** Optional[List[dict]]

**cred**
contains the credentials used to authenticate with bigquery

**Type** google.oauth2.service_account.Credentials

**client**
The bigquery client

**Type** google.cloud.bigquery.Client

**dataset**
The dataset associated with the collector

**Type** google.cloud.bigquery.Dataset

**table**
The table associated with the collector

**Type** google.cloud.bigquery.Table

**connected**
True if a connection has been established I.e. credentials have been authenticated, otherwise False

**Type** bool

**connect**() → bool
Attempts to authenticate with provided credentials.

**Workflow**

1. Load the credentials from the service account file
2. Instantiate the bigquery Client
3. Attempt to create the dataset and if a Conflict exception is thrown then load the dataset
4. Attempt to load the table and if a NotFound exception is thrown then create the table

**Returns** True if the table is successfully loaded/created otherwise False

**stream**(*data: pandas.core.frame.DataFrame*, *retry_depth: int = 5*) → None
Attempts to store the incoming data into bigquery

**Workflow**

1. If authentication has not been performed, call *self.connect()*
2. Attempt to store the DataFrame to bigquery
   - If successful, check to see if any previous uploads have failed and attempt to store those as well
3. If the DataFrame cannot be successfully stored set the connection to False
4. If the retry_depth is not 0 perform a recursive call attempting to store the data again
5. If the retry_depth has reached zero, log an exception and store the DataFrame locally

Failed uploads will be stored in

- STORE/upload_failed/**collector_name**/bigquery/**timestamp_ns**.parquet

**Parameters**

- **data** – The DataFrame that should be stored to bigquery
- **retry_depth** – Used for a recursive call counter should the DataFrame fail to be stored

**Returns** None

**build_schema**() → Optional[List[google.cloud.bigquery.schema.SchemaField]]
Attempts to build the schema that will be used for table creation

Iterates through the dtypes items and generate the appropriate SchemaFields

**Returns** The schema generated if successful otherwise None

**build_bq_schema**() → Optional[List[dict]]
Attempts to build the schema that will be used to upload data to bigquery via DataFrame.to_gbq()

Iterates through the dtypes items and generate the appropriate schema dictionary

**Returns** The schema generated if successful otherwise None

### api2db.stream.stream2local module

### Contains the Stream2Local class

**class** api2db.stream.stream2local.**Stream2Local**(*name: str*, *path: Optional[str] = None*, *mode: str = 'shard'*, *fmt: str = 'parquet'*, *drop_duplicate_keys: Optional[List[str]] = None*)
Bases: *api2db.stream.stream.Stream*

Streams data from the associated collector directly to a local file in real-time

**__init__**(*name: str*, *path: Optional[str] = None*, *mode: str = 'shard'*, *fmt: str = 'parquet'*, *drop_duplicate_keys: Optional[List[str]] = None*)
Creates a Stream2Local object and attempts to build its dtypes

**Parameters**

- **name** – The name of the collector associated with the stream

- **path** – The path to either a single file or a file directory dictated by the *mode* parameter
- **mode** –
  - *mode="shard"* (default) will store each incoming file independently in the specified *path* In shard mode the file will be named **timestamp_ns**.fmt
  - *mode="update"* will update the file located at the specified *path* with the new data
  - *mode="replace"* will replace the file located at the specified *path* with the new data
- **fmt** –
  - *fmt="parquet"* (default/recommended) stores the files using parquet format
  - *fmt="json"* stores the files using JSON format
  - *fmt="pickle"* stores the files using pickle format
  - *fmt="csv"* stores the files using csv format
- **drop_duplicate_keys** –
  - *drop_duplicate_keys=None* -> DataFrame.drop_duplicates() performed before storage
  - *drop_duplicate_keys=["uuid"]* -> DataFrame.drop_duplicates(subset=drop_duplicate_keys) performed before storage

**stream**(*data: pandas.core.frame.DataFrame*) → None
Stores the incoming data into its stream target using the specified *mode*

**Parameters** **data** – The data to be stored

**Returns** None

**stream_shard**(*data: pandas.core.frame.DataFrame*) → None
Stores the incoming data to the specified directory path using the file naming schema **timestamp_ns**.fmt

**Parameters** **data** – The data to store to the file

**Returns** None

**stream_update**(*data: pandas.core.frame.DataFrame*) → None
Updates the existing data at the specified file path and adds the incoming data

**Parameters** **data** – The data to add to the file

**Returns** None

**stream_replace**(*data: pandas.core.frame.DataFrame*) → None
Replaces the existing data at the specified file path with the incoming data

**Parameters** **data** – The data to replace the file with

**Returns** None

### api2db.stream.stream2omnisci module

### Contains the Stream2Omnisci class

**Warning:** Due to dependency conflicts and issues with the current published branch of the pymapd library the following steps must be taken to support streaming/storing data to Omnisci

```
> pip install pymapd==0.25.0

> pip install pandas --upgrade

> pip install pyarrow --upgrade
```

This occurs because of issues with the dependencies of the pymapd library being locked in place. I've opened an issue on this, and they appear to be working on it. The most recent publish seemed to break other things. Until this gets fixed this is a simple work-around. This will allow api2db to work with Omnisci, however there may be issues with attempts to utilize features of the pymapd library outside of what api2db uses, so use with caution. –Tristen

**class** api2db.stream.stream2omnisci.**Stream2Omnisci**(*name: str*, *db_name*, *username: Optional[str] = None*, *password: Optional[str] = None*, *host: Optional[str] = None*, *auth_path: Optional[str] = None*, *protocol: str = 'binary'*, *chunk_size: int = 0*, *store: bool = False*)

Bases: *api2db.stream.stream.Stream*

Streams data from the associated collector directly to Omnisci in real-time

**__init__**(*name: str*, *db_name*, *username: Optional[str] = None*, *password: Optional[str] = None*, *host: Optional[str] = None*, *auth_path: Optional[str] = None*, *protocol: str = 'binary'*, *chunk_size: int = 0*, *store: bool = False*)

Creates a Stream2Omnisci object and attempts to build its dtypes

If dtypes can successfully be created I.e. Data arrives from the API for the first time the following occurs:

- Auto-generates the table schema
- Casts all *string* columns to *categories* as required by Omnisci
- Creates a Omnisci table with the name `collector_name`_stream

---

**Note:** Data-attribute fields will have a `_t` appended to them due to naming conflicts encountered.

Example:

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |

Will become the following in the Omnisci database. (This is only applied to data in the database)

| A_t | B_t | C_t |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |

**Authentication Methods:**

- Supply `auth_path` with a path to an authentication file. Templates for these files can be found in your projects *AUTH/* directory

**OR**

- Supply the `username`, `host`, and `password`

**Parameters**

- **name** – The name of the collector associated with the stream
- **db_name** – The name of the database to connect to
- **username** – The username to authenticate with the database
- **password** – The password to authenticate with the database
- **host** – The host of the database
- **auth_path** – The path to the authentication credentials.
- **protocol** – The protocol to use when connecting to the database
- **chunk_size** – CURRENTLY NOT SUPPORTED
- **store** – True if the super class is a Store object, otherwise False

**Raises**

- **ValueError** – If `auth_path` is provided but is invalid or has incorrect values
- **ValueError** – If `auth_path` is not provided and `username`, `password` or `host` is missing

**con**
The connection to the database

**Type** Optional[pymapd.Connection]

**connected**
returns True if connection is established else False

**Type** Callable[Optional[pymapd.Connection], bool]

**log_str**
A string used for logging

**Type** str

**connect**() → Optional[<Mock name='mock.Connection' id='140699503473936'>]
Attempts to establish a connection to a omnisci database

**Returns** A connection object if a connection can be established, else None

**static cast_categorical**(*data: pandas.core.frame.DataFrame*, *dtypes: pandas.core.series.Series*) → pandas.core.frame.DataFrame
Casts all columns with type `str` to type `category` as required by omnisci and appends a `_t` to column names

**Parameters**

- **data** – The DataFrame that will be stored into the omnisci database
- **dtypes** – The dtypes of the DataFrame

**Returns** Modified DataFrame

**stream**(*data*, *retry_depth=5*)

Attempts to store the incoming data into omnisci

**Workflow**

1. If authentication has not been performed, call *self.connect()*
2. Attempt to store the DataFrame to omnisci
   - If successful, check to see if any previous uploads have failed and attempt to store those as well
3. If the DataFrame cannot be successfully stored set the con to None
4. If the retry_depth is not 0 perform a recursive call attempting to store the data again
5. If the retry_depth has reached zero, log an exception and store the DataFrame locally

Failed uploads will be stored in

- STORE/upload_failed/**collector_name**/omnisci/**timestamp_ns**.parquet

**Parameters**

- **data** – The DataFrame that should be stored to omnisci
- **retry_depth** – Used for a recursive call counter should the DataFrame fail to be stored

**Returns** None

### api2db.stream.stream2sql module

### Contains the Stream2Sql class

**class** api2db.stream.stream2sql.**Stream2Sql**(*name: str*, *db_name: str*, *dialect: str*, *username: Optional[str] = None*, *password: Optional[str] = None*, *host: Optional[str] = None*, *auth_path: Optional[str] = None*, *port: str = ''*, *if_exists: str = 'append'*, *chunk_size: int = 0*, *store: bool = False*)

Bases: *api2db.stream.stream.Stream*

Streams data from the associated collector directly to an SQL database target in real-time

**__init__**(*name: str*, *db_name: str*, *dialect: str*, *username: Optional[str] = None*, *password: Optional[str] = None*, *host: Optional[str] = None*, *auth_path: Optional[str] = None*, *port: str = ''*, *if_exists: str = 'append'*, *chunk_size: int = 0*, *store: bool = False*)

Creates a Stream2Sql object and attempts to build its dtypes

If dtypes can successfully be created I.e. Data arrives from the API for the first time the following occurs:

- Auto-generates the table schema
- Creates the table in the database if it does not exist

**Authentication Methods:**

- Supply `auth_path` with a path to an authentication file. Templates for these files can be found in your projects *AUTH/* directory

**OR**

- Supply the `username`, `host`, and `password`

**Parameters**

- **name** – The name of the collector associated with the stream
- **db_name** – The name of the database to connect to
- **dialect** –
  - *dialect="mysql"* -> Use this to connect to a mysql database
  - *dialect="mariadb"* -> Use this to connect to a mariadb database
  - *dialect="postgresql"* -> Use this to connect to a postgresql database
  - *dialect="amazon_aurora"* -> COMING SOON
  - *dialect="oracle"* -> COMING SOON
  - *dialect="microsoft_sql"* -> COMING SOON
  - *dialect="Something else?"* -> Submit a feature request... or even better build it!
- **username** – The username to authenticate with the database
- **password** – The password to authenticate with the database
- **host** – The host of the database
- **auth_path** – The path to the authentication credentials.
- **port** – The port used when establishing a connection to the database
- **if_exists** –
  - *if_exists="append"* Adds the data to the table
  - *if_exists="replace"* Replaces the table with the new data
  - *if_exists="fail"* Fails to upload the new data if the table exists
- **chunk_size** – CURRENTLY NOT SUPPORTED
- **store** – True if the super class is a Store object, otherwise False

**Raises**

- **ValueError** – If `auth_path` is provided but is invalid or has incorrect values
- **ValueError** – If `auth_path` is not provided and `username`, `password` or `host` is missing

**driver**
The driver to use when connecting with SQLAlchemy

**Type** str

**engine_str**
The full string that is used with sqlalchemy.create_engine

**Type** str

**log_str**
A string used for logging

**Type** str

**con**
The connection to the database

**Type** sqlalchemy.engine.Engine

**connected**
True if connection is established otherwise False

**Type** bool

**load**() → None
Loads the driver and creates the engine string and the log string

**Raises**

- **NotImplementedError** – Support for amazon_aurora has not been implemented in api2pandas yet
- **NotImplementedError** – Support for oracle has not been implemented in api2db yet
- **NotImplementedError** – Support for microsoft_sql has not been implemented in api2db yet

**Returns** None

**connect**() → bool
Attempts to establish a connection to the database

**Returns** True if the connection is established otherwise False

**stream**(*data*, *retry_depth=5*)
Attempts to store the incoming data into the SQL database

**Workflow**

1. If authentication has not been performed, call *self.connect()*
2. Attempt to store the DataFrame to the database
   - If successful, check to see if any previous uploads have failed and attempt to store those as well
3. If the DataFrame cannot be successfully stored set the connected to False
4. If the retry_depth is not 0 perform a recursive call attempting to store the data again
5. If the retry_depth has reached zero, log an exception and store the DataFrame locally

Failed uploads will be stored in

- STORE/upload_failed/**collector_name**/sql.**dialect**/**timestamp_ns**.parquet

**Parameters**

- **data** – The DataFrame that should be stored to the database
- **retry_depth** – Used for a recursive call counter should the DataFrame fail to be stored

**Returns** None

### Module contents

| | |
|---|---|
| Original Author | Tristen Harr |
| Creation Date | 04/27/2021 |
| Revisions | None |

### Module contents

| | |
|---|---|
| Original Author | Tristen Harr |
| Creation Date | 04/27/2021 |
| Revisions | None |

**The creator of api2db is currently searching for a job. He graduates with a bachelors in CS May 15th**

**Contact him by emailing tristenharr@gmail.com**

CHAPTER

THREE

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## a

# INDEX

## Symbols



## A

## B



## C

## D



## E



## F



## G



## I



## J

## L



## M



## P

## Q



## R



## S



## T



## W