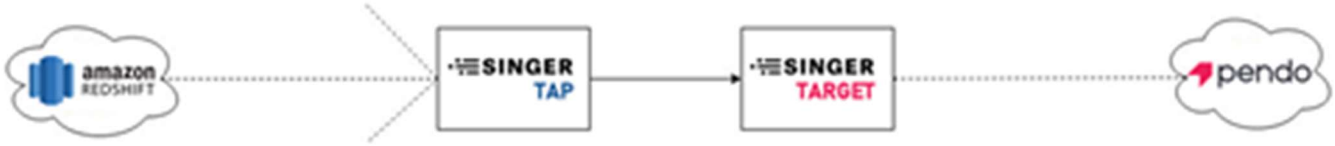
Redshift-Pendo Integration

KELLEN C. BONILLA
https://github.com/kc-bonilla/

REDSHIFT-PENDO INTEGRATION

ABOUT



ETL pipeline to extract data from Redshift data warehouse and batch stream requests to pendo.io restful API to be consumed and joined on Pendo accounts/visitors data, allowing for better insights & user experiences; built upon the singer.io open-source ETL framework (development in Python); leveraged an asynchronous HTTP client to stream up to 2,500 records (~3mb)/second to Pendo; capable of both full-table replication and incremental data loads; deployed via EC2 instance.

REDSHIFT-PENDO INTEGRATION

INITIAL REQUIREMENTS

1. Updates Pendo Visitor/Accounts Custom Fields with any source attribute(s)
2. Performs Full-Replication
3. Performs Incremental Data Loads
4. Maintains State Between Runs
5. Versions States and Can Revert to Prior State
6. Catches Tap Schema Changes
7. Allows for Monitored Job Runs/Job Results
8. Notifies on Failure - Triggers SNS Topic Alert to DAE team Topic in AWS upon failure to prompt a response and troubleshooting
9. Adjusts for Runtime Limits

1. Any attribute that lives inside a Redshift source table may be pushed into the Pendo Account or Visitor objects with a 1:1 primary key match and pre-existing Custom Field Mapping in Pendo

2. Supports Scheduled Runs: Runs on a scheduled basis, detecting which records have been added, deleted, or changed, and inserts, updates, or deletes the corresponding data in the corresponding Pendo tables.

3. Supports Full Replication

1. Extracts all data from the source table each time the tap is invoked without a state file.

2. Supports Incremental Replication: ability to incrementally update custom attributes in Pendo so that synchronization time and risk exposure are minimized with each run.

3. Maintains state by bookmarking the designated replication column (i.e., Last Updated) with each incremental update/invocation; Stored in state.json file.

4. Replication_method and replication_key JSON attributes passed to respective stream's metadata in the catalog.json file after run.

5. Detects Schema Changes in Source Data, Updates Stream Catalog Accordingly

6. Taps detect schema changes in source databases and target connectors alter the destination tables automatically. Based on the schema change type:

7. Idempotent; Not at risk of record corruption, Mistake Tolerant; PUT/POST requests with redundant payload will not insert duplicates.

No Matching Key in Pendo?

Pendo will only push data into matching objects. It will not create new records in Pendo. If no matching record is found the data will not push to Pendo.

How Can I Add Custom Fields for Data to Be Pushed to Pendo?

Data types must match for custom field mappings the account/visitor object

Singer, developed by Stitch, is an open source tool that helps deal with the messy world of custom ETL scripts. Singer is set up to allow users to mix and match inputs and outputs, making their ETL processes much more modular, and therefore easier to run and maintain. It's also designed to use JSON to move all data between sources and destinations, so you won't have to worry about incompatible formats once you've set your ETL tasks up to run. So how does Singer do all this? Basically, it breaks traditional ETL scripts into Taps and Targets, described below:

Taps

Taps are data extraction scripts, designed to pull data from files, databases, and APIs. Taps take a configuration file and an optional state file as input and produces an ordered stream of record, state, & schema messages as output.

Targets

Targets are data loading scripts, designed to take the data extracted by Taps and move it along to their target destination

Record Message

JSON-encoded data of any kind.

State Message

Used to persist information between invocations of a Tap.

Schema Message

Describes the datatypes of the records in the stream.

Pendo is a software platform that helps product teams and application owners improve users' experience with their software (whether those users are customers or their fellow employees).

Pendo provides product-led organizations with the tools they need to understand what their customers and/or employees are doing in (and how they feel about) the product, guide users to success with in-app messages, and collect feedback in order to drive continuous improvement and efficiency. This allows organizations to understand whether the products and features they're building are being used and if so, by whom.

Pendo combines product analytics and in-app messaging belong together in one tool.. This ability to experiment and drive behavior change within your product is truly what can improve your business and create better outcomes for your customers. Customer feedback spurs innovation, helps teams identify and understand friction, and provides visibility into where additional opportunity may lie. Collecting feedback at scale, though, can be incredibly challenging and time consuming. Pendo makes it easier to capture and prioritize customer feature requests, so that customers remain at the center of your planning process.

Pendo Entities

- Visitors: General information on all visitors Pendo has ever seen.
- Accounts: General information on all accounts Pendo has ever seen.
- Events: Each interaction with your application by a visitor (click, page load, metadata, guide events)

REDSHIFT TAP

Extracts data from Redshift data warehouse and writes it to a standard stream in a JSON-based format, which can be piped to-and consumed by-any target.

CONFIGURATION FILE

* description: the fields required for the Redshift connection config file are specified here.

* filetype: JSON

* arg options: (-c, --config)

```
{
    "host": "aws.something.or.other",
    "port": 5439,
    "dbname": "my_analytics",
    "user_password": "myuser",
    "start_date": "1234",
    "schema": "mytapname"
}
```

CATALOG FILE

Contains a list of stream objects that correspond to each available table in the Redshift schema designated in your config file.

- filetype: JSON
- arg options: (--catalog)

DEPENDENCIES

- Connection to Redshift
- Python 3.6+

DISCOVERY MODE

~/Github/data-and-analytics/singer/tap-redshift/tap-redshift/bin/tap-redshift --config tap_rs_config.json -d

PENDO TARGET

Consumes JSON-based standard stream data from the Redshift Tap (or any other tap) and communicates with the Pendo API to set or update pendo attributes in accordance with this incoming data.

Pendo Target: API Features

METHOD

POST

URI

/api/v1/metadata/{kind}/{group}/value

PARAMETERS:

- kind - "visitor" or "account"
- group - type of metadata field: agent or custom

DATA

JSON array (see example format) Request Headers
content-type: application/json
x-pendo-integration-key: <PENDO_INTEGRATION_KEY>

STATUS CODES

- 200: The bulk update completed.
- 400: The format is unacceptable due to malformed JSON or missing field mappings.
- 408: The call took too long and timed out.

ATTRIBUTES

- Total number = Sum of updated + failed.
- Failed number = Sum of length(missing) + length(errors).

RATE LIMITS & SERVICE PROTECTION API LIMITS

Pendo API allows for any number of records to be submitted for update, but the call is limited to five (5) minutes. Any calls that take longer will violate this service protection API Limit and result in a 408 Request Timeout.

COMPILATION FOLDER STRUCTURE

target-pendo/

├── target_pendo/ <-- Python package with source code

└── target_pendo.py

└── client.py

└── request.py

└── errors.py

└── target_pendo.py

└── setup.py

*format: (yyyy-mm-ddthh:mm:ssz)
**optional

DEPENDENCIES

- Pendo Admin access credentials may create custom fields via Data Mappings page in Pendo.

EXAMPLE: INITIAL FULL REPLICATION

```
sudo ~/Github/data-and-analytics/singer/tap-redshift/tap-redshift/bin/tap-redshift --config tap_rs_config.json --catalog catalog_full_rep.json | /Users/kbonilla/.virtualenvs/target-pendo/src/target-pendo/target_pendo --config target_config.json > state.json
```

The tap can be invoked in discovery mode to get the available tables and columns in the database

$ tap-redshift --config config.json -d

A full catalog tap is written to stdout, with a JSON-schema description of each table.
Each source table directly corresponds to a Singer stream.

Redirect output from tap's discovery mode to a file to be modified when the tap is next invoked in sync mode.
To run tap in discovery mode and copy output into a catalog.json file:

$ tap-redshift -c config.json -d > catalog.json

STEP 1 SELECT THE TABLES YOU WANT TO SYNC

In sync mode, tap-redshift requires a catalog file to be supplied, where the user must have selected which streams (tables) should be transferred. Streams are not selected by default.

For each stream in the catalog, find the metadata section. That is the section you will modify to select the stream and, optionally, individual properties too.

The stream itself is represented by an empty “breadcrumb” object.

You can select it by adding "selected": true to its metadata.

The tap can then be invoked in sync mode with the properties catalog argument:

tap-redshift -c config.json --catalog catalog.json | target-pendo -c config-dw.json

STEP 2 SYNC YOUR DATA

FULL_TABLE replication is used by default.

EXAMPLE

```
{       "metadata": [
        {
                "breadcrumb": [],
                "metadata": {
                  "selected": true,
                  "selected-by-default": false,
                  "replication-method": "INCREMENTAL",
                  "replication-key": "updated_at",
                  ...
                }
          }
        ]
```

Can now Invoke the tap again in sync mode. This time the output will have STATE message that contains a replication_key_value and bookmark for data that was extracted.

Redirect the output to a state.json file. Normally, the target will echo the last STATE after it has finished processing data.

EXAMPLE

```
tap-redshift -c config.json --catalog catalog.json | \
        target-pendo -c config-dw.json > state.json
```

The state.json file should look like:

```
{
  "currently_syncing": null,
  "bookmarks": {
        "sample-dbname.public.sample-name": {
          "replication_key": "updated_at",
          "version": 1516304171710,
          "replication_key_value": "2013-10-29T09:38:41.341Z"
        }
  }
}
```

For subsequent runs, can invoke the incremental replication by passing the latest state in order to limit data only to what has been modified since the last execution.

EXAMPLE: INCREMENTAL REPLICATION

```
tail -1 state.json > latest-state.json; tap-redshift -c config-redshift.json –catalog \
      catalog.json -s latest-state.json | target-pendo -c config.json > state.json
```

EXAMPLE: INCREMENTAL REPLICATION

```
tail -1 state.json > latest-state.json;
tap-redshift -c –config redshift.json –catalog catalog.json -s \
     latest-state.json | target-pendo -c config.json > state.json
```

EXAMPLE: INCREMENTAL REPLICATION

```
tail -1 state.json > latest-state.json;
tap-redshift -c –config redshift.json –catalog catalog.json -s \
     latest-state.json | target-pendo -c config.json > state.json
```

discover:

tap-redshift -c config-redshift.json -d > catalog.json

sync:

**tail -1 state.json > latest-state.json; **
**tap-redshift **
**-c config-redshift.json **
**--catalog catalog.json **
**-s latest-state.json | **
target-pendo -c config-dw.json > state.json

Retry Operations

Service protection API limit errors will return a Retry-After Duration value indicating the duration before any new requests from the user can be processed.

INFO METRIC:

{"type": "counter",
"metric": "record_count", "value": 331207,

"tags": {"database": null, "table": "public.pendo_integration_account"}}

{"type": "ACTIVATE_VERSION",
"stream": "pendo_integration_account", "version": 1614185051899}

{"type": "STATE",
"value": {"currently_syncing": "dev.public.pendo_integration_account", "bookmarks":
{"dev.public.pendo_integration_account": {"version": null}}}}

INFO METRIC:

{"type": "timer",
"metric": "job_duration", "value": 27.544612169265747,
"tags": {"job_type": "sync_table", "database": null, "table": "public.pendo_integration_account", "status":
"succeeded"}}

{"type": "STATE",
"value": {"currently_syncing": null,
"bookmarks": {"dev.public.pendo_integration_account": {"version": null}}}}

CHANGELOG

An idiosyncratic feature of Foreground's Tap-Redshift-Target-Pendo Singer Integration is a POST request (althgouh resembling a GET request) made on the Tap-Redshift side (in sync.py) that utilizes the Pendo Aggregation API that allows us to query all Foreground Pendo Accounts/Visitors for those having UUID-formatted IDs. This indicates that the Account/Visitor has been active since a shift toward UUIDs was enacted in the Pendo Snippet and drastically reduces the Accounts/Visitor IDs to be queried from Redshift before updating that IDs associated attributes in Pendo.

EXAMPLE: INITIAL FULL REPLICATION

sudo ~/Github/data-and-analytics/singer/tap-redshift/tap-redshift/bin/tap-redshift --config tap_rs_config.json --

catalog catalog.json | /Users/kbonilla/.virtualenvs/target-pendo/src/target-pendo/target_pendo --config

target_config.json > state.json

Redirect output from tap's discovery mode to a file to be modified when the tap is next invoked in sync mode.
To run tap in discovery mode and copy output into a catalog.json file:

$ tap-redshift -c config.json -d > catalog.json

Select the tables you want to sync:

tap-redshift -c tap_config.json --catalog catalog.json | target-pendo -c target_config.json > state.json

You can now Invoke the tap again in sync mode. This time the output will have STATE message that contains a replication_key_value and bookmark for data that was extracted.

Redirect the output to a state.json file. Normally, the target will echo the last STATE after it has finished processing data.

EXAMPLE

The state.json file should look like:

```
{
  "currently_syncing": null,
  "bookmarks": {
    "sample-dbname.public.sample-name": {
      "replication_key": "updated_at",
      "version": 1516304171710,
      "replication_key_value": "2013-10-29T09:38:41.341Z"
    }
  }
}
```

For subsequent runs, can invoke the incremental replication by passing the latest state in order to limit data only to what has been modified since the last execution.

```
tail -1 state.json > latest-state.json;
```

tap-redshift -c tap_config.json --catalog catalog.json -s latest-state.json | target-pendo -c target_config.json > state.json

1. open terminal
2. ssh dae > passphrase
3. open terminal
4. cd Github/data-and-analytics/singer/tap-redshift/tap-redshift
5. sudo ~/Github/data-and-analytics/singer/tap-redshift/tap-redshift/bin/tap-redshift -c tap_config.json --catalog catalog.json

sudo ~/Github/data-and-analytics/singer/tap-redshift/tap-redshift/bin/tap-redshift -c tap_config.json --catalog catalog.json | ~/.virtualenvs/target-pendo/bin/target-pendo -c target_config.json

sudo ~/Github/data-and-analytics/singer/tap-redshift/tap-redshift/bin/tap-redshift -c tap_config.json --catalog catalog3.json | ~/.virtualenvs/target-pendo/bin/target-pendo -c target_config.json

rate-limits, but not a prob unless concurrency, multithread pooling for Pendo ID update

STEP 1 CREATE A CONFIGURATION FILE

When you install tap-redshift, you need to create a config.json file for the database connection.

The json file requires the following attributes:

- host
- port
- dbname
- user
- password
- start_date (Notation: yyyy-mm-ddThh:mm:ssZ)

And an optional attribute:

- schema

EXAMPLE

```
{
   "host": "REDSHIFT_HOST",
   "port": "REDSHIFT_PORT",
   "dbname": "REDSHIFT_DBNAME",
   "user": "REDSHIFT_USER",
   "password": "REDSHIFT_PASSWORD",
   "start_date": "REDSHIFT_START_DATE",
   "schema": "REDSHIFT_SCHEMA"
}
```

STEP 2 DISCOVER WHAT CAN BE EXTRACTED FROM REDSHIFT

The tap can be invoked in discovery mode to get the available tables and columns in the database. It points to the config file created to connect to redshift:

```
$ tap-redshift --config config.json -d
```

A full catalog tap is written to stdout, with a JSON-schema description of each table. A source table directly corresponds to a Singer stream.

Redirect output from the tap's discovery mode to a file so that it can be modified when the tap is to be invoked in sync mode.

```
$ tap-redshift -c config.json -d > catalog.json
```

This runs the tap in discovery mode and copies the output into a catalog.json file.
A catalog contains a list of stream objects, one for each table available in your Redshift schema.

EXAMPLE


```
{
    "streams": [
      {
          "tap_stream_id": "sample-dbname.public.sample-name",
          "stream": "sample-stream",
          "database_name": "sample-dbname",
          "table_name": "public.sample-name"
          "schema": {
             "properties": {
                 "id": {
                    "minimum": -2147483648,
                    "inclusion": "automatic",
                    "maximum": 2147483647,
                    "type": [
                        "null",
                        "integer"
                    ]
                 },
                 "name": {
                    "maxLength": 255,
                    "inclusion": "available",
                    "type": [
                        "null",
                        "string"
                    ]
                 },
                 "updated_at": {
                    "inclusion": "available",
                    "type": [
                        "string"
                    ],
                    "format": "date-time"
                 },
             },
             "type": "object"
          },
          "metadata": [
             {
                 "metadata": {
                    "selected-by-default": false,
                    "selected": true,
                    "is-view": false,
                    "table-key-properties": ["id"],
                    "schema-name": "sample-stream",
                    "valid-replication-keys": [
                        "updated_at"
                    ]
                 },
                 "breadcrumb": [],
             },
             {
                 "metadata": {
                    "selected-by-default": true,
                    "sql-datatype": "int2",
                    "inclusion": "automatic"
                 },
                 "breadcrumb": [
                    "properties",
                    "id"
                 ]
             },
             {
                 "metadata": {
                    "selected-by-default": true,
                    "sql-datatype": "varchar",
                    "inclusion": "available"
                 },
                 "breadcrumb": [
                    "properties",
                    "name"
                 ]
             },
             {
                 "metadata": {
                    "selected-by-default": true,
                    "sql-datatype": "datetime",
                    "inclusion": "available",
                 },
                 "breadcrumb": [
                    "properties",
                    "updated_at"
                 ]
             }
          ]
      }
    ]
}
```

STEP 3

SELECT THE TABLES YOU WANT TO SYNC

In sync mode, tap-redshift requires a catalog file to be supplied, where the user must have selected which streams (tables) should be transferred. Streams are not selected by default.

For each stream in the catalog, find the metadata section. That is the section you will modify to select the stream and, optionally, individual properties too.

The stream itself is represented by an empty breadcrumb.

EXAMPLE

```
"metadata": [
  {
        "breadcrumb": [],
        "metadata": {
          "selected-by-default": false,
          ...
        }
  }
]
```

You can select it by adding "selected": true to its metadata.

EXAMPLE

```
"metadata": [
  {
        "breadcrumb": [],
        "metadata": {
          "selected": true,
          "selected-by-default": false,
          ...
        }
  }
]
```

The tap can then be invoked in sync mode with the properties catalog argument:

```
tap-redshift -c config.json --catalog catalog.json | target-datadotworld -c
config-dw.json
```

STEP 4

SYNC YOUR DATA

There are two ways to replicate a given table. FULL_TABLE and INCREMENTAL. FULL_TABLE replication is used by default.

FULL TABLE

Full-table replication extracts all data from the source table each time the tap is invoked without a state file.

INCREMENTAL

Incremental replication works in conjunction with a state file to only extract new records each time the tap is invoked i.e continue from the last synced data.

To use incremental replication, we need to add the replication_method and replication_key to the streams (tables) metadata in the catalog.json file.

EXAMPLE

```
"metadata": [
  {
        "breadcrumb": [],
        "metadata": {
          "selected": true,
          "selected-by-default": false,
          "replication-method": "INCREMENTAL",
          "replication-key": "updated_at",
          ...
        }
  }
]
```

We can then invoke the tap again in sync mode. This time the output will have STATE messages that contains a replication_key_value and bookmark for data that were extracted.

Redirect the output to a state.json file. Normally, the target will echo the last STATE after it has finished processing data.

Run the code below to pass the state into a state.json file.

EXAMPLE

```
tap-redshift -c config.json --catalog catalog.json | \
  target-datadotworld -c config-dw.json > state.json
```

The state.json file should look like:

```
{
  "currently_syncing": null,
  "bookmarks": {
      "sample-dbname.public.sample-name": {
        "replication_key": "updated_at",
        "version": 1516304171710,
        "replication_key_value": "2013-10-29T09:38:41.341Z"
      }
  }
}
```

For subsequent runs, you can then invoke the incremental replication passing the latest state in order to limit data only to what has been modified since the last execution.

```
tail -1 state.json > latest-state.json; \
tap-redshift \
  -c config-redshift.json \
  --catalog catalog.json \
      -s latest-state.json | \
        target-datadotworld -c config-dw.json > state.json
```

All steps in one Makefile

For your convenience, all the steps mentioned above are captured in the Makefile below. This example uses target-datadotworld but can be modified to use any other Singer target.

REQUIRES PYTHON 3.6+

install:

pip3 install tap-redshift; pip3 install target-datadotworld

```
CREATE TABLE users
  (
     userid INT PRIMARY KEY,
     NAME   TEXT NOT NULL
  );

CREATE TABLE movies
  (
     movieid INT PRIMARY KEY,
     NAME    TEXT NOT NULL
  );

CREATE TABLE taginfo
  (
     tagid   INT PRIMARY KEY,
     content TEXT UNIQUE
  );

CREATE TABLE genres
  (
     genreid INT PRIMARY KEY,
     NAME    TEXT UNIQUE
  );

CREATE TABLE ratings
  (
     userid    INT REFERENCES users(userid),
     movieid   INT REFERENCES movies(movieid),
     rating    NUMERIC CHECK (0 <= rating AND rating <= 5),
     timestamp BIGINT,
     PRIMARY KEY(userid, movieid)
  );

CREATE TABLE tags
  (
     userid    INT REFERENCES users(userid),
     movieid   INT REFERENCES movies(movieid),
     tagid     INT REFERENCES taginfo(tagid),
     timestamp BIGINT,
     PRIMARY KEY(userid, movieid, tagid)
  );

CREATE TABLE hasagenre
  (
     movieid INT REFERENCES movies(movieid),
     genreid INT REFERENCES genres(genreid),
     PRIMARY KEY(movieid, genreid)
  );
```

CATALOG DISCOVERY

discover:

tap-redshift -c config-redshift.json -d > catalog.json

```
tap-redshift -c config-redshift.json --catalog catalog.json |
target-datadotworld -c config-dw.json > state.json
```

```
    // Prints the N-Queens solution to the console

    for(scanf("%d",&s);*a-s;v=a[j*=v]-
a[i],k=i<s,j+=(v=j<s&&(!k&&!!printf(2+"\n\n%c"-(!l<<!j),"
#Q"[l^v?(l^j)&1:2])&&++l||a[i]<s&&v&&v-i+j&&v+i-
j))&&!(l%=s),v||(i==j?a[i+=k]=0:++a[i])>=s*k&&++a[--i]);
```

FULL SYNC

fullsync:
  tap-redshift \
    -c config-redshift.json \
    --catalog catalog.json | \
      target-datadotworld -c config-dw.json > state.json

```
tap-redshift -c config-redshift.json --catalog catalog.json | target-
datadotworld -c config-dw.json > state.json
```

INCREMENTAL SYNC

sync:
  **tail -1 state.json > latest-state.json; **
  **tap-redshift **
   **-c config-redshift.json **
   **--catalog catalog.json **
   **-s latest-state.json | **
    target-datadotworld -c config-dw.json > state.json

UPDATING DIRECTORIES IN EC2 INSTANCE

* Tap-redshift and target-pendo directories/virtualenvs in EC2 instance are not up to date with the attached code. To update them, ssh into the EC2 instance with ssh@[instance_public_dns] and execute the command below from a separate, local terminal:

scp -r [local source path] [ec2-username]@[instance_public_ipv4]:[instance destination path]

EXAMPLE:

**scp -r ~/.virtualenvs/target-pendo/src /target_pendo/__init__.py **

kcbonilla@3.16.70.231:~/target-pendo/lib/python3.8/site-packages/target_pendo

If adding a src directory in the virtualenvs (like in updated code), repeat for the src directory.

RUNNING THE PIPELINE (TAP | TARGET) ON EC2 INSTANCE

- After you ssh into the instance, you can execute the following compound command to run the pipe command for sync.

- Attached is an example showing this before the most recent updates/mods. The singer messages render much slower in the EC2 instance than locally, but requests move quickly. After replacing the current instance code with the updated code and config files, the integration will be syncing FULL_TABLE replications for both accounts and visitors streams, one after another. The visitors stream usually kicks off toward the end of the accounts stream's execution, and you will see this in a disruption in the stdout/logging, but it does not prevent the accounts stream from finishing requests.

EXAMPLE (LOCAL, WITH EXECUTABLES IN BIN)

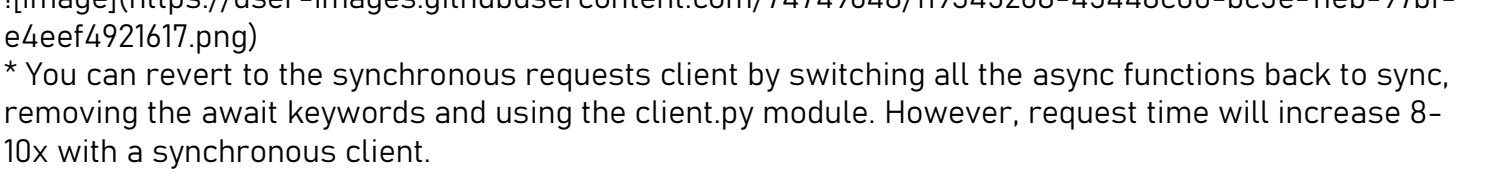
```
python3 ~/.virtualenvs/tap-redshift/bin/tap-redshift -c tap_config.json --catalog

catalog.json | ~/.virtualenvs/target-pendo/bin/target-pendo -c target_config.json \

--batch_records 500 --request_delay 0.05 -—attempts 3 --verbose > state.json
```

EXAMPLE (EC2, CALL PACKAGE MODULES)

```
source tap-redshift/bin/activate; python3.8 \

    ~/tap-redshift/lib/python3.8/site-packages/tap_redshift/__init__.py -c tap_config.json \

        --catalog catalog.json | python3.8 \

            ~/target-pendo/lib/python3.8/site-packages/target_pendo/__init__.py \

                -c target_config.json --batch_records 500 --verbose > state.json
```

NOTE ON HTTP CLIENT(S)

* Can swap HTTPX async client with aiohttp async client (most common async client) but requests will not go through without providing something similar to the following for SSL certification:



* You can revert to the synchronous requests client by switching all the async functions back to sync, removing the await keywords and using the client.py module. However, request time will increase 8-10x with a synchronous client.