tom_pittgoogle

Release v0.1.0

Troy Raen

Nov 06, 2021

TROY RAEN'S DOCS

CHAPTER

ONE

CONNECTING TO TOM TOOLKIT

- TOM Toolkit
- tom_desc
  - ingestmessages.py (ingest SCIMMA)
- tom_fink

ToDo:

- run Django
- run TOM
- run `tom_desc`
- run `tom_fink`
- change `ingestmessages.py` to listen to our stream
- add us as a `tom_toolkit` module

- Following TOM Toolkit Getting Started

```
conda create --name tom python=3.7
conda activate tom

# use mypgb test account
export GOOGLE_CLOUD_PROJECT="pitt-broker-user-project"
export GOOGLE_APPLICATION_CREDENTIALS="/Users/troyraen/Documents/broker/repo/GCP_auth_
↪key-pitt_broker_user_project.json"
# export GOOGLE_APPLICATION_CREDENTIALS=/Users/troyraen/Documents/broker/repo/GCP_auth_
↪key-mypgb-raentroy.json
# export PITTGOOGLE_OAUTH_CLIENT_ID="187635371164-eoeg3i6vp4bcd26p7l8cvjir3ga6nb7a.apps.
↪googleusercontent.com"

export PITTGOOGLE_OAUTH_CLIENT_ID="591409139500-hb4506vjuao7nvq40k509n7lljf3o3oo.apps.
↪googleusercontent.com"
export PITTGOOGLE_OAUTH_CLIENT_SECRET=""
# /Users/troyraen/Documents/broker/repo/GCP_oauth-client_secret.json

# add tom_pittgoogle to path
python -m pip install -e .
# export PYTHONPATH="${PYTHONPATH}:/Users/troyraen/Documents/broker/tom/tom_pittgoogle"
```

(continues on next page)

(continued from previous page)

```
# export DJANGO_SETTINGS_MODULE=tom_pittgoogle.settings

# export PYTHONPATH="${PYTHONPATH}:/Users/troyraen/Documents/broker/tommy/tommy"
export DJANGO_SETTINGS_MODULE=tommy.settings
# export DJANGO_SETTINGS_MODULE="tom_desc.settings"

# pip install requests requests_oauthlib
pip install google-cloud-bigquery
pip install google-cloud-pubsub
pip install fastavro
pip install requests_oauthlib

pip install tomtoolkit
pip install whitenoise
pip install psycopg2
# create a new project
django-admin startproject tommy

cd tommy

# edit settings to add tom_setup. then:
./manage.py tom_setup
./manage.py migrate
./manage.py runserver
# navigate to http://127.0.0.1:8000/

# to make updates
./manage.py makemigrations
./manage.py migrate
./manage.py runserver
```

1.1 Register an app

```
import os
from django.core.wsgi import get_wsgi_application
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'tommy.settings')
application = get_wsgi_application()
```

Add some things from our Broker-Web

Print more helpful errors for `RuntimeError("populate() isn't reentrant")`

- edit `django/apps/registry.py` as described here

1.2 Build in RTD

```
export BUILD_IN_RTD=True
export DJANGO_SETTINGS_MODULE=tom_pittgoogle.settings
export SECRET_KEY='4iq)g7qh+1+0g03$!3kx0@*=v!#2ioi@^-f=-^ix6l(z7c_6d8'
```

Put at top of python modules, if needed:

```
import os
import troy_fncs as tfncs
settings = tfncs.AttributeDict({
    'GOOGLE_CLOUD_PROJECT': os.getenv('GOOGLE_CLOUD_PROJECT'),
    'PITTGOOGLE_OAUTH_CLIENT_ID': '591409139500-hb4506vjuao7nvq40k509n7lljf3o3oo.apps.
↪googleusercontent.com',
    'PITTGOOGLE_OAUTH_CLIENT_SECRET': "<FILL-IN>",
})
```

1.3 Run StreamPython locally

```
clean_params = {
    'subscription_name': 'ztf-loop',
    'classtar_threshold': None,
    'classtar_gt_lt': 'gt',
    'max_results': 100,

}
```

1.4 Message size

```
from python_fncs.pubsub_consumer import Consumer as Consumer

consumer = Consumer('ztf-loop')
msgs = consumer.stream_alerts(parameters={'max_results': 1, 'max_backlog': 1})
msg = msgs[0]
msg.size  # bytes
# result is: 67362

# 1 TiB ~= 1.6e7 alerts = $40
```

CHAPTER

TWO

INDEX FOR (OTHER) DOCS

2.1 Basic Code Workflow

Each implementation does things a bit differently, but they share a basic workflow:

The *Broker* instantiates a *Consumer* and uses it to fetch, unpack, and process alerts.

The *Consumer* can accept a user filter and return only alerts that pass.

Here is a compact but working example of a *Broker*'s `fetch_alerts` method for the *StreamRest* implementation.

```
def fetch_alerts(self):
    from consumer_stream_rest import ConsumerStreamRest

    subscription_name = "ztf-loop"
    max_messages = 10
    lighten_alerts = True  # flatten the alert dict and drop extra fields. optional.
    # If you pass a callback function, the Consumer will run each alert through it.
    # Optional. Useful for user filters. Here's a basic demo.
    def user_filter(alert_dict):
        passes_filter = True
        if passes_filter:
            return alert_dict
        else:
            return None
    callback = user_filter

    consumer = ConsumerStreamRest(subscription_name)

    response = consumer.oauth2.post(
        f"{consumer.subscription_url}:pull", data={"maxMessages": max_messages},
    )

    alerts = consumer.unpack_and_ack_messages(
        response, lighten_alerts=lighten_alerts, callback=callback,
    )  # List[dict]

    return iter(alerts)
```

2.2 How to integrate with TOM Toolkit

This assumes that you know how to run a TOM server/site using the TOM Toolkit.

1. Clone this repo and put the directory on your path. (`git clone https://github.com/mwvgroup/tom_pittgoogle.git`)
2. Add Pitt-Google to your TOM. Follow the TOM Toolkit instructions in the section Using Our New Alert Broker. Our modules were written following the instructions preceding that section.
   - In your `settings.py` file:
     - Add these to the `TOM_ALERT_CLASSES` list:

```
'tom_pittgoogle.broker_stream_rest.BrokerStreamRest',
'tom_pittgoogle.broker_stream_python.BrokerStreamPython',
'tom_pittgoogle.broker_database_python.BrokerDatabasePython',
```

     - Add these additional settings:

```
# see the Authentication docs for more info
GOOGLE_CLOUD_PROJECT = "pitt-broker-user-project"  # user's project
PITTGOOGLE_OAUTH_CLIENT_ID = os.getenv("PITTGOOGLE_OAUTH_CLIENT_ID")
PITTGOOGLE_OAUTH_CLIENT_SECRET = os.getenv("PITTGOOGLE_OAUTH_CLIENT_SECRET")
```

3. After running `makemigrations`, etc. and authenticating yourself, navigate to the "Alerts" page on your TOM site. You should see three new broker options corresponding to the three Pitt-Google classes you added to the `TOM_ALERT_CLASSES` list.

2.3 Authentication

Users authenticate themselves by following an OAuth 2.0 workflow. Authentication is required to make API calls.

2.3.1 Requirements

1. The user must have a Google account (e.g., Gmail address) that is authorized make API calls through the project that is defined by the `GOOGLE_CLOUD_PROJECT` variable in the Django `settings.py` file. Any project can be used, as long as the user is authorized.
   - We have a test project setup that we are happy to add community members to, for as long as that remains feasible. Send Troy a request, and include your Google account info (Gmail address).
2. Since this is still in dev: Contact Troy to be added to the OAuth's list of authorized test users, and to obtain the `PITTGOOGLE_OAUTH_CLIENT_ID` and `PITTGOOGLE_OAUTH_CLIENT_SECRET`. Include your Google account info (Gmail address).

2.3.2 Authentication Workflow

Note: Currently this is a bit painful because the user must:

- re-authenticate every time a query is run.
- interact via the command line. When running a query from the TOM site's "Query a Broker" page, **the process will hang until the user follows the prompts on the command line and completes the authentication**. The site may temporarily crash until this is completed.

(TODO: integrate the OAuth with Django, and automatically refresh tokens)

Workflow - The user will:

1. Visit a URL, which will be displayed on the command line when the *Consumer* class is initialized (currently, when the *Broker*'s `fetch_alerts` is called).
2. Log in to their Google account. This authenticates their access to make API calls through the project.
3. Authorize this *PittGoogleConsumer* app/module to make API calls on their behalf. This only needs to be done once for each API access "scope" (Pub/Sub, BigQuery, and Logging).
4. Respond to the prompt on the command line by entering the full URL of the webpage they are redirected to after completing the above.

What happens next? - The *Consumer*:

1. Completes the instantiation of an `OAuth2Session`, which is used to either make HTTP requests directly, or instantiate a credentials object for the Python client.
2. Instantiates a `Client` object to make API calls with (Python methods only).
3. Checks that it can successfully connect to the requested resource.

2.4 StreamRest

Note: The Pitt-Google broker uses Pub/Sub to publish live streams, rather than Apache Kafka. See pubsub for a basic overview.

2.4.1 BrokerStreamRest

TOM Toolkit broker to listen to a Pitt-Google Pub/Sub stream via the REST API.

Relies on *ConsumerStreamRest* to manage the connections and work with data.

See especially:

BrokerStreamRest.request_alerts	Pull alerts using a POST request with OAuth2, unpack, apply user filter.
BrokerStreamRest.user_filter	Apply the filter indicated by the form's parameters.

class tom_pittgoogle.broker_stream_rest.**BrokerStreamRest**
Pitt-Google broker class to pull alerts from a stream via the REST API.

Base class: tom_alerts.alerts.GenericBroker

fetch_alerts(*parameters*)
Entry point to pull and filter alerts.

form
alias of *tom_pittgoogle.broker_stream_rest.FilterAlertsForm*

request_alerts(*parameters*)
Pull alerts using a POST request with OAuth2, unpack, apply user filter.

Returns alerts (List[dict])

to_generic_alert(*alert*)
Map the Pitt-Google alert to a TOM *GenericAlert*.

static user_filter(*alert_dict*, *parameters*)
Apply the filter indicated by the form's parameters.

Used as the *callback* to *BrokerStreamRest.unpack_and_ack_messages*.

Parameters

- **alert_dict** – Single alert, ZTF packet data as a dictionary. The schema depends on the value of *lighten_alerts* passed to *BrokerStreamRest.unpack_and_ack_messages*. If *lighten_alerts=False* it is the original ZTF alert schema (https://zwickytransientfacility.github.io/ztf-avro-alert/schema.html). If *lighten_alerts=True* the dict is flattened and extra fields are dropped.
- **parameters** – parameters submitted by the user through the form.

Returns *alert_dict* if it passes the filter, else *None*

class tom_pittgoogle.broker_stream_rest.**FilterAlertsForm**(**args*, ***kwargs*)
Basic form for filtering alerts.

Fields:

subscription_name (CharField)

classtar_threshold (FloatField)

classtar_gt_lt (ChoiceField)

max_results (IntegerField)

property media
Return all media required to render the widgets on this form.

2.4.2 ConsumerStreamRest

Consumer class to manage Pub/Sub connections via REST, and work with message data.

Pub/Sub REST API docs: https://cloud.google.com/pubsub/docs/reference/rest

Used by *BrokerStreamRest*, but can be called independently.

Basic workflow:

```
consumer = ConsumerStreamRest(subscription_name)

response = consumer.oauth2.post(
    f"{consumer.subscription_url}:pull", data={"maxMessages": max_messages},
)

alerts = consumer.unpack_and_ack_messages(
    response, lighten_alerts=True, callback=user_filter,
)  # List[dict]
```

See especially:

ConsumerStreamRest.authenticate	Guide user through authentication; create *OAuth2Session* for HTTP requests.
ConsumerStreamRest.touch_subscription	Make sure the subscription exists and we can connect.
ConsumerStreamRest.unpack_and_ack_messages	Unpack and acknowledge messages in *response*; run *callback* if present.

class tom_pittgoogle.consumer_stream_rest.**ConsumerStreamRest**(*subscription_name*)

Consumer class to manage Pub/Sub connections and work with messages.

Initialization does the following:

- Authenticate the user. Create an *OAuth2Session* object for the user/broker to make HTTP requests with.
- Make sure the subscription exists and we can connect. Create it, if needed.

authenticate()

Guide user through authentication; create *OAuth2Session* for HTTP requests.

The user will need to visit a URL, authenticate themselves, and authorize *PittGoogleConsumer* to make API calls on their behalf.

The user must have a Google account that is authorized make API calls through the project defined by the *GOOGLE_CLOUD_PROJECT* variable in the Django *settings.py* file. Any project can be used, as long as the user has access.

Additional requirement because this is still in dev: The OAuth is restricted to users registered with Pitt-Google, so contact us.

TODO: Integrate this with Django. For now, the user interacts via command line.

delete_subscription()

Delete the subscription.

This is provided for the user's convenience, but it is not necessary and is not automatically called.

- Storage of unacknowledged Pub/Sub messages does not result in fees.
- Unused subscriptions automatically expire; default is 31 days.

touch_subscription()

Make sure the subscription exists and we can connect.

If the subscription doesn't exist, try to create one (in the user's project) that is attached to a topic of the same name in the Pitt-Google project.

Note that messages published before the subscription is created are not available.

unpack_and_ack_messages(*response*, *lighten_alerts=False*, *callback=None*, ***kwargs*)

Unpack and acknowledge messages in *response*; run *callback* if present.

If *lighten_alerts* is True, drop extra fields and flatten the alert dict.

callback is assumed to be a filter. It should accept an alert dict and return the dict if the alert passes the filter, else return None.

2.5 StreamPython

Note: The Pitt-Google broker uses Pub/Sub to publish live streams, rather than Apache Kafka. See pubsub for a basic overview.

2.5.1 BrokerStreamPython

TOM Toolkit broker to listen to a Pitt-Google Pub/Sub stream via the Python client.

Relies on *ConsumerStreamPython* to manage the connections and work with data.

See especially:

BrokerStreamPython.fetch_alerts	Entry point to pull and filter alerts.
BrokerStreamPython.user_filter	Apply the filter indicated by the form's parameters.

class tom_pittgoogle.broker_stream_python.**BrokerStreamPython**

Pitt-Google broker interface to pull alerts from Pub/Sub via the Python client.

Base class: `tom_alerts.alerts.GenericBroker`

fetch_alerts(*parameters*)

Entry point to pull and filter alerts.

Pull alerts using a Python client, unpack, apply user filter.

This demo assumes that the real use-case is to save alerts to a database rather than view them through a TOM site. Therefore, the *Consumer* currently saves the alerts in real time, and then simply returns a list of alerts after all messages are processed. That list is then coerced into an iterator here. If the user really cares about the iterator, *ConsumerStreamPython.stream_alerts* can be tweaked to yield the alerts in real time.

form

alias of *tom_pittgoogle.broker_stream_python.FilterAlertsForm*

to_generic_alert(*alert_dict*)

Map the Pitt-Google alert to a TOM *GenericAlert*.

to_target(*alert_dict*)

Map the Pitt-Google alert to a TOM *Target*.

static user_filter(*alert_dict*, *parameters*)

Apply the filter indicated by the form's parameters.

Used as the *callback* to *BrokerStreamPython.unpack_and_ack_messages*.

Parameters

- **alert_dict** – Single alert, ZTF packet data as a dictionary. The schema depends on the value of *lighten_alerts* passed to *BrokerStreamPython.unpack_and_ack_messages*. If *lighten_alerts=False* it is the original ZTF alert schema (https://zwickytransientfacility.github.io/ztf-avro-alert/schema.html). If *lighten_alerts=True* the dict is flattened and extra fields are dropped.
- **parameters** – parameters submitted by the user through the form.

Returns *alert_dict* if it passes the filter, else *None*

class tom_pittgoogle.broker_stream_python.**FilterAlertsForm**(**args*, ***kwargs*)

Basic form for filtering alerts.

Fields:

subscription_name (CharField)

classtar_threshold (FloatField)

classtar_gt_lt (ChoiceField)

max_results (IntegerField)

timeout (IntegerField)

max_backlog (IntegerField)

property media

Return all media required to render the widgets on this form.

2.5.2 ConsumerStreamPython

Consumer class to pull Pub/Sub messages via a Python client, and work with data.

Pub/Sub Python Client docs: https://googleapis.dev/python/pubsub/latest/index.html

Used by *BrokerStreamPython*, but can be called independently.

Use-case: Save alerts to a database

The demo for this implementation assumes that the real use-case is to save alerts to a database rather than view them through a TOM site. Therefore, the *Consumer* currently saves the alerts in real time, and then simply returns a list of alerts after all messages are processed. That list is then coerced into an iterator by the *Broker*. If the user really cares about the *Broker*'s iterator, *stream_alerts* can be tweaked to yield the alerts in real time.

Basic workflow:

```
consumer = ConsumerStreamPython(subscription_name)

alert_dicts_list = consumer.stream_alerts(
    user_filter=user_filter,
    **kwargs,
)
# alerts are processed and saved in real time. the list is returned for convenience.
```

See especially:

ConsumerStreamPython.touch_subscription	Make sure the subscription exists and we can connect.
ConsumerStreamPython.stream_alerts	Execute a streaming pull and process alerts through the *callback*.
ConsumerStreamPython.callback	Process a single alert; run user filter; save alert; acknowledge Pub/Sub msg.
ConsumerStreamPython.save_alert	Save the alert to a database.

class tom_pittgoogle.consumer_stream_python.**ConsumerStreamPython**(*subscription_name*, *ztf_fields=None*)

Consumer class to manage Pub/Sub connections and work with messages.

Initialization does the following:

- Authenticate the user via OAuth 2.0.
- Create a *google.cloud.pubsub_v1.SubscriberClient* object.
- Create a *queue.Queue* object to communicate with the background thread running the streaming pull.
- Make sure the subscription exists and we can connect. Create it, if needed.

To view logs, visit: https://console.cloud.google.com/logs

- Make sure you are logged in, and your project is selected in the dropdown at the top.
- Click the "Log name" dropdown and select the subscription name you instantiate this consumer with.

TODO: Give the user a standard logger.

authenticate_with_oauth()

Guide user through authentication; create *OAuth2Session* for credentials.

The user will need to visit a URL, authenticate themselves, and authorize *PittGoogleConsumer* to make API calls on their behalf.

The user must have a Google account that is authorized make API calls through the project defined by the *GOOGLE_CLOUD_PROJECT* variable in the Django *settings.py* file. Any project can be used, as long as the user has access.

Additional requirement because this is still in dev: The OAuth is restricted to users registered with PittGoogle, so contact us.

TODO: Integrate this with Django. For now, the user interacts via command line.

callback(*message*)

Process a single alert; run user filter; save alert; acknowledge Pub/Sub msg.

Used as the callback for the streaming pull.

delete_subscription()

Delete the subscription.

This is provided for the user's convenience, but it is not necessary and is not automatically called.

- Storage of unacknowledged Pub/Sub messages does not result in fees.
- Unused subscriptions automatically expire; default is 31 days.

get_credentials(*user_project*)

Create user credentials object from service account credentials or an OAuth.

Try service account credentials first. Fall back to OAuth.

save_alert(*alert*)
Save the alert to a database.

stream_alerts(*user_filter=None*, *user_callback=None*, ***kwargs*)
Execute a streaming pull and process alerts through the *callback*.

The streaming pull happens in a background thread. A *queue.Queue* is used to communicate between threads and enforce the stopping condition(s).

Parameters

- **user_filter** (*Callable*) – Used by *callback* to filter alerts before saving. It should accept a single alert as a dictionary (flat dict with fields determined by *ztf_fields* attribute). It should return the alert dict if it passes the filter, else None.
- **user_callback** (*Callable*) – Used by *callback* to process alerts. It should accept a single alert as a dictionary (flat dict with fields determined by *ztf_fields* attribute). It should return True if the processing was successful; else False.
- **kwargs** (*dict*) – User's parameters. Should include the parameters defined in *BrokerStreamPython*'s *FilterAlertsForm*. There must be at least one stopping condition (*max_results* or *timeout*), else the streaming pull will run forever.

touch_subscription()
Make sure the subscription exists and we can connect.

If the subscription doesn't exist, try to create one (in the user's project) that is attached to a topic of the same name in the Pitt-Google project.

Note that messages published before the subscription is created are not available.

2.6 DatabasePython

2.6.1 BrokerDatabasePython

TOM Toolkit broker to query a BigQuery table via the Python API.

Relies on *ConsumerDatabasePython* to manage the connections and work with data.

See especially:

BrokerDatabasePython.request_alerts	Query alerts using the user filter and unpack.

class tom_pittgoogle.broker_database_python.**BrokerDatabasePython**
Pitt-Google broker to query alerts from the database via the Python client.

Base class: tom_alerts.alerts.GenericBroker

fetch_alerts(*parameters*)
Entry point to query and filter alerts.

form

alias of *tom_pittgoogle.broker_database_python.FilterAlertsForm*

request_alerts(*parameters*)
Query alerts using the user filter and unpack.

The SQL statement returned by the *Consumer* implements the current user filter.

Returns alerts (List[dict])

to_generic_alert(*alert*)
Map the Pitt-Google alert to a TOM *GenericAlert*.

class tom_pittgoogle.broker_database_python.**FilterAlertsForm**(**args*, ***kwargs*)
Basic form for filtering alerts; currently implemented in the SQL statement.

Fields: objectId (`CharField`)

candid (`IntegerField`)

max_results (`IntegerField`)

property media
Return all media required to render the widgets on this form.

2.6.2 ConsumerDatabasePython

Consumer class to manage BigQuery connections via Python client, and work with data.

BigQuery Python Client docs: https://googleapis.dev/python/bigquery/latest/index.html

Used by *BrokerDatabasePython*, but can be called independently.

Basic workflow:

```
consumer = ConsumerDatabasePython(table_name)

sql_stmnt, job_config = consumer.create_sql_stmnt(parameters)
query_job = consumer.client.query(sql_stmnt, job_config=job_config)

alerts = consumer.unpack_query(query_job)  # List[dict]
```

See especially:

ConsumerDatabasePython.authenticate	Guide user through authentication; create *OAuth2Session* for credentials.
ConsumerDatabasePython.create_sql_stmnt	Create the SQL statement and a job config with the user's *parameters*.
ConsumerDatabasePython.unpack_query	Unpack alerts from *query_job*; run *callback* if present.

class tom_pittgoogle.consumer_database_python.**ConsumerDatabasePython**(*table_name*)
Consumer class to query alerts from BigQuery, and manipulate them.

Initialization does the following:

- Authenticate the user via OAuth 2.0.
- Create a *google.cloud.bigquery.Client* object for the user/broker to query database with.
- Check that the table exists and we can connect.

To view logs, visit: https://console.cloud.google.com/logs

- Make sure you are logged in, and your project is selected in the dropdown at the top.
- Click the "Log name" dropdown and select the table name you instantiate this consumer with.

TODO: Give the user a standard logger.

authenticate()
Guide user through authentication; create *OAuth2Session* for credentials.

The user will need to visit a URL, authenticate themselves, and authorize *PittGoogleConsumer* to make API calls on their behalf.

The user must have a Google account that is authorized make API calls through the project defined by the *GOOGLE_CLOUD_PROJECT* variable in the Django *settings.py* file. Any project can be used, as long as the user has access.

Additional requirement because this is still in dev: The OAuth is restricted to users registered with PittGoogle, so contact us.

TODO: Integrate this with Django. For now, the user interacts via command line.

create_sql_stmnt(*parameters*)
Create the SQL statement and a job config with the user's *parameters*.

unpack_query(*query_job*, *callback=None*, ***kwargs*)
Unpack alerts from *query_job*; run *callback* if present.

A basic filter is implemented directly in the SQL statement produced by *create_sql_stmnt*. More complex filters could be implemented here via a *callback* function.

PYTHON MODULE INDEX

t

INDEX

S

T

U