#!/usr/bin/env python

"""

Post request and get response From Bloomberg Transport.

"""

import getpass

import json

import logging

import os

import sys

from sqlalchemy import or\_

from sqlalchemy.exc import DBAPIError

from etl.bbg\_transport.dto import RequestDataItem, RequestItem, RequestOptionItem

from etl.core.util import uri\_get, uri\_post

from etl.repo.fnd\_cfdw.etl\_config import EtlConfigRepo

from etl.repo.pim\_pm.pl\_bbg\_batch import PlBbgBatchRepo

from etl.repo.pim\_pm.pl\_bbg\_batch\_series\_vw import PlBbgBatchSeriesVwRepo

USAGE = ['Pl Automation Agent', ['action', {'help': 'QUEUE or FETCH or LOAD'}]]

class FetcherAgent(object):

"""

This class Does the Fetching operations.

"""

def \_\_init\_\_(self):

self.log = logging.getLogger("{}".format(

os.path.splitext(os.path.basename(\_\_file\_\_))[0]))

self.USERNAME = getpass.getuser()

def \_\_enter\_\_(self):

# make a database connection and return it

return self

def \_\_exit\_\_(self, exc\_type, exc\_value, exc\_traceback):

if exc\_type is None:

# No exception

pass

@classmethod

def get\_request(cls, repo, option):

"""

get\_request(repo) -> Object

Return object with IN\_QUEUE as batch\_status\_code.

:param repo: Object

:param option: Str

:return: Object

"""

model = repo.model

try:

if option == 'req':

logging.info('Getting requests from pl\_bbg\_batch table with IN\_QUEUE status')

data = repo.query.filter(model.batch\_status\_code == 'IN\_QUEUE').all()

elif option == 'resp':

logging.info('Getting requests from pl\_bbg\_batch '

'table with SENT\_TO\_BT or PENDING status')

data = repo.query.filter(

or\_(model.batch\_status\_code == 'SENT\_TO\_BT',

model.bt\_status\_code == 'PENDING')).all()

elif option == 'load':

logging.info('Getting all the requests from pl\_bbg\_batch table which are in SENT\_TO\_BT status')

data = repo.query.filter(model.batch\_status\_code == 'SENT\_TO\_BT').all()

return data

except DBAPIError as err:

logging.error(err)

class RequestAgent(FetcherAgent):

"""

This class posts request to Bloomberg Transport.

"""

def \_\_init\_\_(self):

super(RequestAgent, self).\_\_init\_\_()

logging.info('Fetching the required values from the EtlConfig Repo')

try:

rec = EtlConfigRepo.instance.get\_by\_config\_code('PL\_BT\_ENDPOINT')

self.base\_url = rec.config\_value

rec = EtlConfigRepo.instance.get\_by\_config\_code('PL\_BT\_DESCRIPTION')

self.description = rec.config\_value

rec = EtlConfigRepo.instance.get\_by\_config\_code('PL\_BT\_FORMAT')

self.response\_format = rec.config\_value

rec = EtlConfigRepo.instance.get\_by\_config\_code('PL\_BT\_REQUESTOR\_CODE')

self.requestor\_code = rec.config\_value

except Exception as ex:

self.log.critical(

"Unable to initialize %s: %s", self.\_\_class\_\_.\_\_name\_\_, ex)

raise

def get\_priority\_list(self, result):

"""

get\_priority\_list(result) -> List of Objects

Return prioritized list of Objects

:param result: Object

:return: List of Objects

"""

logging.info('Prioritizing the list as per bbg\_program\_code '

'with GETDATA as high priority')

data\_list = list()

history\_list = list()

for i in result:

if i.bbg\_program\_code == 'GETDATA':

data\_list.append(i)

else:

history\_list.append(i)

data\_list = self.get\_priority\_list\_by\_interface\_code(data\_list)

history\_list = self.get\_priority\_list\_by\_interface\_code(history\_list)

plist = data\_list + history\_list

return plist

@staticmethod

def get\_priority\_list\_by\_interface\_code(result):

"""

get\_priority\_list\_by\_interface\_code(result) -> List of Objects

Return prioritized list of Objects

:param result: Object

:return: List of Objects

"""

logging.info('Re-Prioritizing the list as per bbg\_interface\_code,'

'with SAPI as high priority')

plist = []

for i in result:

if i.bbg\_interface\_code == 'SAPI':

plist.insert(0, i)

else:

plist.append(i)

return plist

def get\_request\_object(self, obj, result\_series):

"""

get\_request\_object(self, obj, result\_series) -> Json

Return Json constructed from the input objects.

:param obj: Object

:param result\_series: Object

:return: Json

"""

logging.info('Preparing the Request Object')

data\_items = [RequestDataItem(tag=i.pl\_series\_code, yellow\_key=i.bbg\_yellow,

ticker=i.bbg\_ticker) for i in result\_series]

headers = self.get\_headers(obj)

fields = self.get\_request\_fields(result\_series)

request\_options = [RequestOptionItem(option\_name=key, option\_value=headers[key])

for key in headers]

request = RequestItem(request\_description=self.description,

requestor\_code=self.requestor\_code,

program\_code=obj.bbg\_program\_code,

interface\_code=obj.bbg\_interface\_code,

response\_format\_code=self.response\_format,

request\_data\_items=data\_items,

request\_options=request\_options,

request\_fields=fields)

logging.info('Converting the Request Object to json format')

payload = json.dumps(request.to\_json())

return payload

@staticmethod

def get\_headers(obj):

"""

Return string of value obtained from Object.

get\_headers(obj) -> str

:param obj: Object

:return: str

"""

headers = dict()

if obj.bbg\_program\_code == "GETDATA":

headers['DATERANGE'] = str(obj.asof\_end\_date\_key) + \

"|" + str(obj.asof\_end\_date\_key)

elif obj.bbg\_program\_code == "GETHISTORY":

headers['DATERANGE'] = str(obj.asof\_start\_date\_key) + \

"|" + str(obj.asof\_end\_date\_key)

return headers

@staticmethod

def get\_request\_fields(result\_series):

"""

Return List obtained from Object.

get\_request\_fields(result\_series) -> List

:param result\_series:

:return:

"""

request\_fields\_list = []

for i in result\_series:

request\_fields\_list.append(i.bbg\_mnemonic)

return list(set(request\_fields\_list))

def post\_to\_bt(self, payload):

"""

post\_to\_bt(payload) -> Json

Return Json, response from BT

:param payload: Json

:return: Json

"""

logging.info('Sending the request to BT')

logging.info('POST: %s, \r\n\t%s', self.base\_url, payload)

try:

response = uri\_post(self.base\_url + 'request\_data', payload)

logging.info('response: %s \r\nresponse:\t%s', self.base\_url, response)

return response

except Exception as ex:

logging.error(ex)

@staticmethod

def update\_request(batch\_id, bt\_request\_id, bt\_status\_code,

request\_obj, batch\_status\_code, repo):

"""

:param batch\_id: Int

:param bt\_request\_id: Int

:param bt\_status\_code: Str

:param request\_obj: Json

:param batch\_status\_code: Str

:param repo: Object

"""

logging.info('Updating the staging the table with the status of the request')

model = repo.model

try:

update\_row = repo.query.filter(model.batch\_status\_code == 'IN\_QUEUE',

model.batch\_id == batch\_id).all()

update\_row[0].batch\_status\_code = batch\_status\_code

update\_row[0].bt\_request\_id = bt\_request\_id

update\_row[0].bt\_status\_code = bt\_status\_code

update\_row[0].bt\_request\_payload = request\_obj

repo.save(update\_row)

except DBAPIError as ex:

logging.error(ex)

def run(self):

"""

Get the response for all the requests and update the table.

"""

result = FetcherAgent.get\_request(PlBbgBatchRepo(), 'req')

priority\_list = self.get\_priority\_list(result)

for i in priority\_list:

logging.info("Fetching the records with batch\_id" + ' ' + str(i.batch\_id))

repo = PlBbgBatchSeriesVwRepo()

model = repo.model

try:

result\_batch = repo.query.filter(model.batch\_id == i.batch\_id).all()

obj = self.get\_request\_object(i, result\_batch)

response = self.post\_to\_bt(obj)

self.update\_request(i.batch\_id, response['request\_id'],

str(response['request\_status']),

str(obj), 'SENT\_TO\_BT', PlBbgBatchRepo())

except Exception as ex:

logging.error(ex)

class ResponseAgent(FetcherAgent):

"""

This class gets the response from the Bloomberg Transport.

"""

def \_\_init\_\_(self):

super(ResponseAgent, self).\_\_init\_\_()

try:

rec = EtlConfigRepo.instance.get\_by\_config\_code('PL\_BT\_ENDPOINT')

self.base\_url = rec.config\_value

except Exception as ex:

self.log.critical(

"Unable to initialize %s: %s", self.\_\_class\_\_.\_\_name\_\_, ex)

def get\_request\_status\_by\_url(self, obj):

"""

Return Json by GET method.

:param obj: Object

:return: Json

"""

logging.info('Sending the check status request to BT')

logging.info('GET: %s, \r\n\t%s', self.base\_url, str(obj.bt\_request\_id))

try:

response = uri\_get(self.base\_url + 'check\_status' + '/' + str(obj.bt\_request\_id))

logging.info('response: %s \r\nresponse:\t%s', self.base\_url, response)

return response

except Exception as ex:

logging.error(ex)

@staticmethod

def update\_request(batch\_id, data\_file\_path,

bt\_request\_id, bt\_status\_code, repo):

"""

:param batch\_id: Int

:param data\_file\_path: Str

:param bt\_request\_id: Int

:param bt\_status\_code: Str

:param repo: Object

"""

logging.info('Updating the staging the table with the status of the request')

model = repo.model

try:

update\_row = repo.query.filter(model.bt\_request\_id == bt\_request\_id,

model.batch\_id == batch\_id).all()

update\_row[0].bt\_status\_code = bt\_status\_code

if update\_row[0].bt\_status\_code == 'BBGERROR':

update\_row[0].batch\_status\_code = 'BBG\_ERROR'

elif update\_row[0].bt\_status\_code == 'BTERROR':

update\_row[0].batch\_status\_code = 'BT\_ERROR'

update\_row[0].bt\_response\_file\_path = data\_file\_path

repo.save(update\_row)

except DBAPIError as err:

logging.error(err)

def run(self):

"""

Get the response for all the requests and update the table.

"""

result = FetcherAgent.get\_request(PlBbgBatchRepo(), 'resp')

for i in result:

response = self.get\_request\_status\_by\_url(i)

self.update\_request(i.batch\_id, response['data\_file\_path'], response['request\_id'],

response['request\_status'], PlBbgBatchRepo())

class LoaderAgent(object):

"""

This class is for downloading CSV file to FF\_PL

"""

def \_\_init\_\_(self):

logging.info('Fetching the values of config code from EtlConfigRepo')

self.log = logging.getLogger("{}".format(

os.path.splitext(os.path.basename(\_\_file\_\_))[0]))

try:

logging.info('LoaderAgent')

self.USERNAME = getpass.getuser()

rec = EtlConfigRepo.instance.get\_by\_config\_code('PL\_BT\_DATA\_FILE\_PATH')

self.destination = rec.config\_value

except Exception as ex:

self.log.critical(

"Unable to initialize %s: %s", self.\_\_class\_\_.\_\_name\_\_, ex)

raise

def \_\_enter\_\_(self):

# make a database connection and return it

return self

def \_\_exit\_\_(self, exc\_type, exc\_value, exc\_traceback):

if exc\_type is None:

# No exception

pass

@staticmethod

def copy\_file(src, dst, program\_code):

"""

copy\_file(src, dst, program\_code) -> Bool

Take src, dst and program\_code as agrs and copy the file

if successful return True else False.

:param src: str

:param dst: str

:param program\_code: str

:return: Bool

"""

if os.path.isdir(dst):

destination\_file = os.path.basename(src)

if program\_code == 'GETDATA':

ext = '.inc'

else:

ext = '.full'

name\_ext = os.path.splitext(destination\_file)

destination\_file = name\_ext[0]+ext+name\_ext[1]

dst = os.path.join(dst, destination\_file)

copyfile(src, dst)

logging.info("Copying %s to %s" % (src, dst))

return True

else:

logging.info("Can't copy %s to destination: %s" % (src, dst))

return False

@staticmethod

def update\_request(batch\_id, batch\_status\_code, repo):

"""

Update the table with the corresponding values.

:param batch\_id: int

:param batch\_status\_code: str

:param repo: object

"""

logging.info('Updating the staging table with the status of the request')

model = repo.model

try:

update\_row = repo.query.filter(model.batch\_id == batch\_id).all()

update\_row[0].batch\_status\_code = batch\_status\_code

repo.save(update\_row)

except DBAPIError as ex:

logging.error(ex)

def run(self):

"""

Get the data file path for all the requests, copy and update the table.

"""

result = result = FetcherAgent.get\_request(PlBbgBatchRepo(), 'load')

for i in result:

if i.bt\_status\_code == 'SUCCESS':

if self.copy\_file(i.bt\_response\_file\_path.strip(), self.destination, i.bbg\_program\_code):

self.update\_request(i.batch\_id, 'BT\_DONE', PlBbgBatchRepo())

def main():

"""

Delegates all processing to Agent instance.

"""

logger = logging.getLogger("{}".format(

os.path.splitext(os.path.basename(\_\_file\_\_))[0]))

args = parse\_args(\*USAGE)

action = args.action.upper()

if action == 'QUEUE':

try:

logging.info("Request Agent started")

with RequestAgent() as agent:

agent.run()

except Exception as ex:

logger.critical("Request Agent exited with error: %s", ex)

return -1

else:

logger.info("Request Agent completed successfully.")

elif action == 'FETCH'

try:

logging.info("Response Agent started")

with ResponseAgent() as agent:

agent.run()

except Exception as ex:

logger.critical("Response Agent exited with error: %s", ex)

return -1

else:

logger.info("Response Agent completed successfully.")

elif action == 'LOAD'

try:

logging.info("Loader Agent started")

with LoaderAgent() as agent:

agent.run()

except Exception as ex:

logger.critical("Loader Agent exited with error: %s", ex)

return -1

else:

logger.info("Loader Agent completed successfully.")

else:

raise RuntimeError('Unknown action specified: {}'.format(action))

return 0

if \_\_name\_\_ == "\_\_main\_\_":

sys.exit(main())