"""

The script basically generates the request\_data\_object and posts to BT.

"""

import json

import os

import shutil

import sys

import copy

import pandas

import csv

import re

from stdnum import isin, cusip

from stdnum.gb import sedol

import getpass

import logging

from etl.core.util import uri\_post

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

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

from sqlalchemy.exc import DBAPIError

from ff\_dais.common.utils import decorator

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

from etl.core.util import uri\_post, uri\_get, sanitize\_cmd\_line

from datetime import datetime

from dateutil.parser import parse

from etl.core import util

from etl.core import da\_config

from etl.core.timed import timed

from etl.core.db import ora\_xxx

\_\_app\_\_ = sys.modules['\_\_main\_\_']

BASE\_URL = 'http://ptp-dev/workshop/service/da/bbg\_transport/'

class QueuerAgent(object):

"""

"""

def \_\_init\_\_(self,logger=None):

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

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

self.USERNAME = getpass.getuser()

try:

self.start\_time = datetime.now()

self.end\_time = None

self.options = copy.deepcopy(

vars(options) if options is not None else dict())

self.default\_config = da\_config.get\_etl\_cfg()

self.config = copy.deepcopy(self.default\_config)

self.config.update(self.options)

level\_name = self.config.get('log\_level')

if not level\_name:

level\_name = self.config.get('dais').get('log\_level', 'INFO')

level = logging.getLevelName(level\_name)

os.environ['PYPIMCO\_LOG\_LEVEL\_OVERRIDE'] = level\_name

from core.log import log\_config

log\_config.init\_logging(None, True)

self.log.setLevel(level)

formatter = logging.Formatter(

'%(asctime)s %(threadName)s:%(thread)d %(name)s %(levelname)s %(message)s')

logger = logging.getLogger('')

for handler in logger.handlers:

handler.setFormatter(formatter)

# Try command line argument first --audit-id

self.etl\_audit\_id = self.options.get('etl\_audit\_id')

self.log.info("ETL\_AUDIT\_ID: %s", self.etl\_audit\_id)

# Try command line argument first --audit-id

self.etl\_source\_code = self.options.get('source\_code')

# Use environment variable param if command line

# for etl source code is not set

if self.etl\_source\_code is None:

# Capture etl source code. Created by etl wrapper script

# and saved to the ETL\_SOURCE\_CODE environment variable

self.etl\_source\_code = os.environ.get('ETL\_SOURCE\_CODE')

self.outfile = self.options.get('outfile')

self.requestor\_code = self.options.get('requestor\_code')

self.vertical = self.options.get('vertical')

self.use\_bt\_output\_file = self.options.get('use\_bt\_output\_file')

self.use\_local\_data = self.options.get('use\_local\_data')

if self.use\_local\_data is None:

self.use\_local\_data = os.environ.get('USE\_LOCAL\_DATA', 0)

self.use\_local\_data = bool(int(self.use\_local\_data))

self.dais\_own = None

self.cfdw\_own = None

self.ctx = util.struct(

use\_local\_data=self.use\_local\_data, \*\*self.options)

self.log.info("Agent started at %s", self.start\_time)

except Exception as e:

self.log.critical(

"Unable to initialize QueuerAgent: %s", e)

raise

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

# make a database connection and return it

self.pm\_own = ora\_xxx('PM\_OWN', 'ORAPIM\_DBP')

self.ctx = util.struct(pm\_own=self.pm\_own,\*\*self.ctx)

return self

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

if exc\_type is None:

# No exception

pass

# make sure the db connection gets closed

# Release resources

try:

if self.pm\_own is not None:

self.pm\_own.release()

finally:

self.pm\_own = None

# Release resources

if self.config is not None:

self.config = None

# Display auditing details

self.end\_time = datetime.now()

elapsed\_time = self.end\_time - self.start\_time

self.log.info("Overall time elapsed: %ss", elapsed\_time)

self.log.info("Agent completed at %s", self.end\_time)

self.log = None

# endregion

def get\_request(self,repo):

"""

:param repo:

:return:

"""

model = repo.model

try:

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

except DBAPIError:

logging.info('An Error has occured while connecting to the database.')

except:

logging.info('An Error has occured while fetching the data.')

return data

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

"""

:param result:

:return:

"""

plist = []

data\_list =[]

history\_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

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

"""

:param result:

:return:

"""

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,objdata, result\_series,\*\*ctx):

etl\_bbg\_src = ctx.get('etl\_bbg\_source')

instruments = ctx.get('etl\_source\_in\_instruments\_l1')

options = ctx.get('etl\_source\_in\_options\_l1')

mnemonics = ctx.get('etl\_source\_in\_bbg\_mnemonics\_l1')

fields = map(lambda m: m.bbg\_mnemonic\_code, mnemonics)

items = map(lambda i: dict(

bbg\_query=i.bbq\_query, tag=i.requestor\_tag), instruments)

headers = self.build\_headers(etl\_bbg\_src, options)

use\_bt\_output\_file = ctx.get('use\_bt\_output\_file', False)

payload = self.get\_data(

headers=headers, fields=fields, items=items,

response\_format='VERTICAL' if self.vertical else 'HORIZONTAL'

)

return payload

def get\_data(

self, headers, fields, items,

description='Get data', program\_code='GETDATA',

interface\_code='DL', response\_format='HORIZONTAL'):

data\_items = map(lambda i: RequestDataItem(

bbg\_query=i.get('bbg\_query'), tag=i.get('tag')), items)

request\_options = map(

lambda key: RequestOptionItem(

option\_name=key, option\_value=headers[key]), headers.keys())

request = RequestItem(request\_description=description,

requestor\_code=self.requestor\_code,

program\_code=program\_code,

interface\_code=interface\_code,

response\_format\_code=response\_format,

request\_data\_items=data\_items,

request\_options=request\_options,

request\_fields=fields

)

payload = request.to\_json()

return payload

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

"""

:param payload:

:return:

"""

url = "{}{}".format(self.base\_url, 'request\_data')

self.log.info('POST: %s, \r\n\t%s', url, payload)

response = uri\_post(url, payload)

self.log.info('response: %s \r\nresponse:\t%s', url, response)

if response is None:

raise Exception("Response is empty")

elif response and isinstance(response, dict):

request\_status = response['request\_status']

else:

request\_status = response

raise Exception(

"Service call:HTTP GET - {} failed with status: {}".format(

url, request\_status))

if request\_status in self.bt\_complete\_status:

if request\_status != 'SUCCESS':

raise Exception(

"Service call:HTTP GET - {} failed!".format(url))

if request\_status in self.bt\_error\_status:

error\_type, errors = self.aggregate\_errors(response)

raise Exception(

"Bloomberg{} Returned an Error: {}".format(

error\_type, errors))

status\_uri = response['progression\_url']

response = self.wait\_for\_response(

status\_uri, interval=self.interval, timeout=self.timeout)

if not response:

raise Exception("Timed out while checking for status")

else:

status = response['request\_status']

if status in self.bt\_error\_status:

msg = "Service call:HTTP GET - {} failed!\r\n".format(url)

msg = "{}With Response: {}".format(msg, response)

raise Exception(msg)

if self.use\_bt\_output\_file:

return response

status\_uri = response['progression\_url']

response = self.wait\_for\_response(

status\_uri, interval=self.interval, timeout=self.timeout)

if not response:

raise Exception("Timed out while checking for status")

# data = response['data']

return response

def update\_request(self,batch\_id, bt\_request\_id, progression\_url,

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

"""

:param batch\_id:

:param bt\_request\_id:

:param progression\_url:

:param bt\_status\_code:

:param request\_obj:

:param batch\_status\_code:

:param repo:

:return:

"""

model = repo.model

try:

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

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

except DBAPIError:

logging.info('An Error has occured while connecting to the database.')

except:

logging.info('An Error has occured while fetching the data.')

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

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

repo.save(update\_row)

def run(self):

"""

:return:

"""

result = self.get\_request(PlBbgBatchRepo())

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

for i in priority\_list:

repo = PlBbgBatchSeriesVwRepo()

model = repo.model

try:

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

except DBAPIError:

logging.info('An Error has occured while connecting to the database.')

except:

logging.info('An Error has occured while fetching the data.')

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

print obj

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

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

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

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

# noinspection PyBroadException

def main():

"""

Delegates all processing to Agent instance.

"""

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

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

try:

logging.info("Agent started")

with QueuerAgent(logger=logger) as agent:

agent.run()

except Exception as ex:

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

return -1

else:

logger.info("Agent completed successfully.")

return 0

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

sys.exit(main())