"""

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

"""

import copy

import getpass

import logging

import os

import sys

from datetime import datetime

from sqlalchemy.exc import DBAPIError

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

from etl.core import da\_config

from etl.core import util

from etl.core.db import ora\_xxx

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

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

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

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

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

USAGE = [

'QUEUER agent',

# [['-l', '--log-level', '--log\_level'],

# {'help': 'DEBUG, INFO, WARN, ERROR, CRITICAL',

# 'choices': ['DEBUG', 'INFO', 'WARN', 'ERROR', 'CRITICAL']}],

# [['-e', '--etl-audit-id', '--etl\_audit\_id'],

# {'help': 'Etl audit id for etl jobs max-len(10)', 'type': int}],

# # [['-o', '--outfile'],

# # {'help': 'Etl output file name', 'required': True}],

# [['-v', '--vertical'],

# {'help': 'Etl output file format(VERTICAL if specified), '

# 'works ONLY with single level non bulk requests',

# 'action': 'store\_true', 'default': False}],

# # [['-c', '--requestor-code', '--requestor\_code'],

# # {'help': 'User token for bt API calls', 'required': True}],

# [['-s', '--source-code', '--source\_code'],

# {'help': 'Etl source code for etl source', 'required': True}],

# # [['-b', '--use-bt-output-file', '--use\_bt\_output\_file'],

# # {'help': 'Use bt output file, without modification, works ONLY for single'

# # 'level requests', 'action': 'store\_true', 'default': False}]

]

class QueuerAgent(object):

"""

"""

def \_\_init\_\_(self, logger=None, options=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

cfg\_section = self.config.get('pl\_auto')

self.base\_url = cfg\_section.get('base\_url')

self.description = cfg\_section.get('description')

self.response\_format=cfg\_section.get('response\_format')

self.requestor\_code = cfg\_section.get('requestor\_code')

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.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

@staticmethod

def get\_request(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:

"""

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

@staticmethod

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

#description = 'Get data'

#response\_format ='VERTICAL'

data\_items = map(lambda i: RequestDataItem(tag=i.pl\_series\_code), result\_series)

headers = self.get\_headers(objdata)

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

request\_options = map(

lambda key: RequestOptionItem(

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

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

requestor\_code=self.requestor\_code,

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

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

response\_format\_code=response\_format,

request\_data\_items=data\_items,

request\_options=request\_options,

request\_fields=fields

)

payload = json.dumps(request)

return payload

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

request\_fields\_list = []

for i in result\_series:

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

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

def get\_headers(self, objdata):

headers = dict()

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

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

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

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

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

return headers

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)

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")

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:

"""

ctx = self.ctx

ctx = util.struct(\*\*ctx)

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=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:

cmd\_line = sanitize\_cmd\_line(copy.copy(sys.argv))

logging.info(cmd\_line)

args = util.parse\_args(\*USAGE)

args.source\_code = ''

args.etl\_audit\_id = ''

logging.info("Agent started")

with QueuerAgent(logger=logger, options=args) 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())