AZURE DATABRICKS TO LOAD SHAREPOINT LIST TO SYNAPSE

Retrieve SharePoint List data using Databricks to load data into Synapse

ABSTRACT

This document explains how to fetch SharePoint list, with drop down column and data using Azure Databricks and load it to Synapse

Ashish Sinha

Senior Data Engineer

Contents

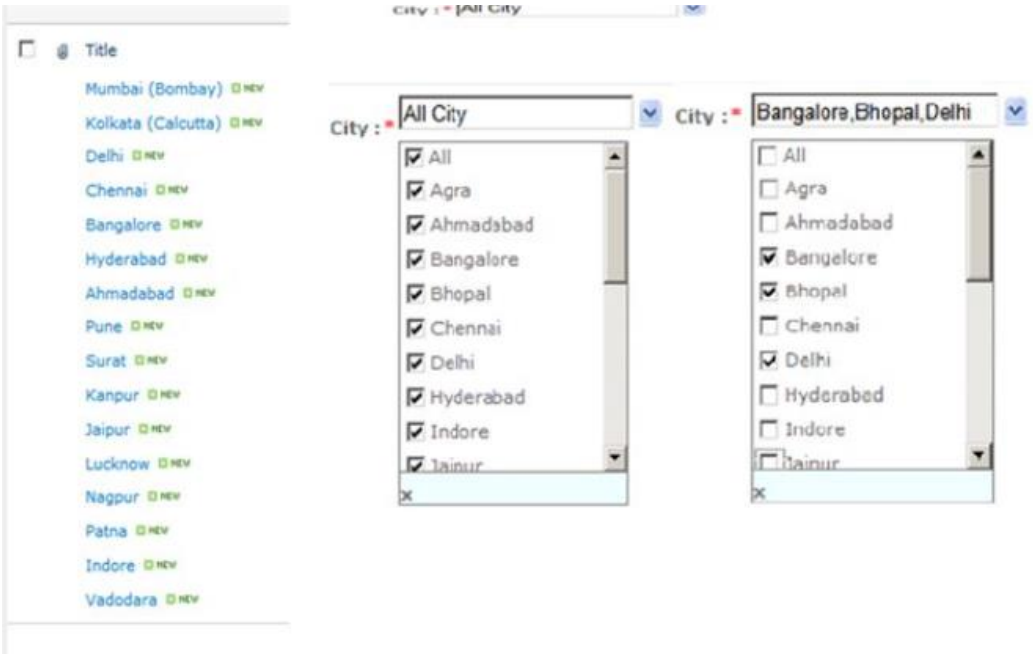
1. Introduction

To fetch the SharePoint list, firstly we connect to SharePoint API with credentials and use GET API call to read all items. This will get us data as a response stored in each items for example Item ID, Title and other metadata associated with it.

But sometimes the list contains set of pre-defined values which are selected using drop-down for a column. This information is stored in items metadata '**FieldValueAsText'**. To retrieve them we hit another GET API for that particular item id.

Example: In the below example, we can see that we have multiple options to select City.

Each row would have different city name or a list of cities which would be stored in the item metadata. This document will explain how could we fetch these text values and store in dataframe to be loaded into Azure Synapse.



2. Authentication and Response

Step1: Authentication

The first step that we would do is to authenticate the SharePoint connection using username and password. We would import all the required libraries for this notebook and connect to SharePoint URL. To work with SharePoint, make sure you have **sharepy** module. If not use the command ***pip install sharepy***. We would also need json library to parse the SharePoint API response.

```
# Import Required Libraries
import json
import sharepy
import pandas as pd
```

```
import time
import datetime
from pyspark.sql.functions import lit,unix_timestamp

# Connect to Sharepoint using username and password
s = sharepy.connect("https://your.sharepoint.com", username="username",
password="password")

# Ge the list items for the LIst Request Tracker
r =
s.get("https://your.sharepoint.com/sites/sitename/_api/web/lists/GetByTitle
('List Name')/items")
```

Step2: Response Processing

We load the response **'r'** into json format and create a dictionary with key as the column names and values as list of data for each of the items fetched.

```
# load the response as json format
json_data = json.loads(r.text)

# create a dictioanlry to be loaded as dataframe
df_dict = {"Report": [], "Request": [], "Date Requested": [], "Requestor":
[], "Date Due": [], "Status": [],"Assigned To": [], "Category": [],
"Request Type": [],"Comments": [], "Effort": [], " Effort Unit": [],
"Purpose": [], "Area": [],"Indication": [], "Name": [], "Created": [],
     "Modified": [], "Created By": [], "Modified By": [], "Cancel": []}
```

3. Data Processing

Now we loop through the json response we got from above steps and fill the values in the dictionary based on the columns.

Note: Each list has a specific GUID generated automatically when we create the list and all the items in that list are referenced by this GUID. Thus, to get any metadata associated with that particular item we use GUID along with item number.

```
# loop through json data and fetch field values
for data in json_data['d']['results']:
    # get Item id. Used fetch Field values
    item_id = data["Id"]

    # item field url is to find field value as text. The guid would remain
same for all the items in the list
    item_field_url =
"https://your.sharepoint.com/sites/sitename/_api/Web/Lists(" \
                    "guid'number')/Items(" + str(item_id) +
")/FieldValuesAsText"
    item_data = s.get(item_field_url)
    item_json_data = json.loads(item_data.text)

# Load the values of the dictionary by appending the values extracted from
JSON
```

```
    df_dict['Report'].append(None if data['Final_x0020_Report'] is None
    else data['Final_x0020_Report']['Url'])
    df_dict['Request'].append(data['Title'])
    df_dict['Date Requested'].append(data['DateRequested'])
    df_dict['Date Due'].append(data['DateDue'])
    df_dict['Status'].append(data['Status'])
    df_dict['Comments'].append(data['Comments'])
    df_dict['Effort'].append(data['RIEffort'])
    df_dict['Effort Unit'].append(data['RIEffortUnit'])
    df_dict['Purpose'].append(data['Purpose'])
    df_dict['Created'].append(data['Created'])
    df_dict['Modified'].append(data['Modified'])
    df_dict['Cancel'].append(data['Cancel'])

    # field values extacted from item json data
    df_dict['Requestor'].append(item_json_data['d']['Requestor'])
    df_dict['Assigned To'].append(item_json_data['d']['AssignedTo'])
    df_dict['Category'].append(item_json_data['d']['Category'])
    df_dict['Request
Type'].append(item_json_data['d']['Request_x005f_x0020_x005f_Type'])
    df_dict['Area'].append(item_json_data['d'][ x005f_x0020_x005f_Area'])
    df_dict['Indication'].append(item_json_data['d']['Indication'])
    df_dict['Name'].append(item_json_data['d']['Name'])
    df_dict['Created By'].append(item_json_data['d']['Author'])
    df_dict['Modified By'].append(item_json_data['d']['Editor'])

    # Load dataframe from dictionary
    df = pd.DataFrame(df_dict)
    df.fillna("", inplace=True)

    # Number of records to be processed
    print("Data count in data frame is : {}".format(df['Request'].count()))
```

To explain the code in nutshell, what we are trying to do is to loop through each of the items, fetch all the required columns. If there are columns whose data has drop-down values, we get the json response in the variable '***item_field_url'*** and '***item_json_data'.*** We then create the dataframe from that dictionary to be loaded to synapse.

4. Data Cleaning

We clean the data by replacing Null or None data. Also rename columns as required or format certain columns to certain data type for instance converting string date type data into date time format before loading it to synapse workspace.

```
def cleaned_df(df):
  for index, rows in df.iterrows():
    row = []
    for i in df.columns:
      row.append(rows[i])
      ##### create a set to get only unique values
    row_set = set(row)
    if len(row_set) == 1:
      df.drop(index, inplace = True)
  return df
```

```
# call the function
df = cleaned_df(df)

# replace unwanted characters from the columns and rename them
cols =
[i.replace(".","").replace("(","").replace(")","").replace("/","").replace(
" ","_") for i in df.columns]
df.columns = cols

# convert column to string. This is so that it is easy to convert the df to
sparkDF
for i in df.columns:
  if df[i].dtype =='object':
    df[i] = df[i].astype('str')
    df[i] = df[i].replace(",","")
#print(df.tail())

# convert some columns to datetime
df['Date_Requested'] = pd.to_datetime(df['Date_Requested']).dt.date
df['Date_Due'] = pd.to_datetime(df['Date_Due']).dt.date

# convert pandas to sparkdf
spark.conf.set("spark.sql.execution.arrow.enabled","true")
sparkDF=spark.createDataFrame(df)
print("Data in spark data frame is :{}".format(sparkDF.count()))
```

5. Loading Data to Azure Synapse

Step 1: Creating Key Vault and Secret Scope

Please follow below links:

1. For Azure Key Vault click here
2. For Secret Scope in Azure Databricks click here

Step 2: Azure Synapse load

1. Connect to Azure Synapse using JDBC connection
2. Convert the pandas dataframe to spark dataframe
3. Load the spark dataframe to synapse

```
blob_account_name = ""
blob_container_name = ""
blob_folder_name = ""
blob_account_key = ""
#connection details for synapse analytics
dwServer = ""
dwPort = "1433"
dwDBName = ""

#Retrieve sql username and passwrod from key vault scope
dwUserName = dbutils.secrets.get(scope = "scope", key = "sqluser")
dwPassword = dbutils.secrets.get(scope = "scope", key = "sqlpassword")

#Create link to connect with the Synapse via JDBC url
tempDir = "wasbs://" + blob_container_name + "@" + blob_account_name +
".blob.core.windows.net/" + blob_folder_name
dwUrl =
"jdbc:sqlserver://"+dwServer+":"+dwPort+";database="+dwDBName+";user="+dwUs
```

```
erName+";password="+dwPassword+";encrypt=true;trustServerCertificate=true;h
ostNameInCertificate=*.sql.azuresynapse.net;loginTimeout=30;"

#connection string for blob
connection_string =
"DefaultEndpointsProtocol=https;AccountName=%s;AccountKey=%s;EndpointSuffix
=core.windows.net" % (blob_account_name,blob_account_key)

# Set up the Blob storage account access key in the notebook session conf.
spark.conf.set(
  "fs.azure.account.key.storagename.blob.core.windows.net",
  blob_account_key)

timestamp = datetime.datetime.fromtimestamp(time.time()).strftime('%Y-%m-%d
%H:%M:%S')
sparkDF =
sparkDF.withColumn('LoadDate',unix_timestamp(lit(timestamp),'yyyy-MM-dd
HH:mm:ss').cast("timestamp"))

sparkDF.write \
  .format("com.databricks.spark.sqldw") \
  .option("url", dwUrl) \
  .option("forwardSparkAzureStorageCredentials", "true") \
  .option("dbTable", "tblName") \
  .option("tempDir", tempDir) \
  .option("truncate","true") \
  .option("maxStrLength", "4000" ) \
  .mode("overwrite") \
  .save()
```