airflow\dags\spotify_dag.py

```python
# 685.652, Spring 2025 - Group 6 Final Project
# spotify_dag.py

# This DAG gets spotify data
# Cleans and transforms it
# Then loads it into postgres

from airflow import DAG
from airflow.operators.empty import EmptyOperator
from airflow.providers.postgres.operators.postgres import PostgresOperator
from airflow.operators.python_operator import PythonOperator, BranchPythonOperator
from airflow.models import Variable
from airflow.providers.postgres.hooks.postgres import PostgresHook
from datetime import datetime, timedelta
import requests
import base64
from urllib.parse import urlencode
import time
import pandas as pd
from datetime import datetime
import os
import unicodedata
import uuid


default_args = {
    'owner': 'group6',
    'depends_on_past': False,
    'start_date': datetime(2025, 1, 1),
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}



# Retrieve tracks from specific playlists from Spotify API
def get_spot_tracks():

    # These need to be set in variables.json
    client_id = Variable.get("SPOTIFY_CLIENT_ID")
    client_secret = Variable.get("SPOTIFY_CLIENT_SECRET")

    # Get a Spotify access token
    auth_header = base64.b64encode(f"{client_id}:{client_secret}".encode()).decode()
    headers = {
        "Authorization": f"Basic {auth_header}",
```

```
        "Content-Type": "application/x-www-form-urlencoded",
    }
    data = {"grant_type": "client_credentials"}

    print("Requesting an access token from Spotify")
    response = requests.post("https://accounts.spotify.com/api/token", headers=headers, data=data)
    if response.status_code == 200:
        token = response.json().get("access_token")
        token_type = response.json().get("token_type")
        expires_in = response.json().get("expires_in")
        print(f"Received a Spotify {token_type} access token that expires in {expires_in} seconds\n")
    else:
        print("Error:", response.status_code, response.text)


    headers = {"Authorization": f"Bearer {token}"}

    # Harcoding good playlists for our purposes
    # Need to do query to get the playlist ID, Spotify seems to rotate playlist IDs
    playlist_queries = [
        "Billboard Hot 100: All #1 hit songs 1958-2024",
        "Most well known songs ever",
        "100 Greatest Rock Songs",
        "Top 100 hip hop hits of all time",
        "100 Greatest Pop Songs",
        "Top 100 Most Popular Electronic Songs Of All Time",
        "Top 100 Alternative Rock Songs",
        "Top 100 Jazz Songs",
        "The 100 Greatest Heavy Metal Songs of All Time",
        "100 Best Folk Songs",
        "Top hits of the 2000s",
        "90s hits top 100 songs",
        "Top hits of the 2010s",
        "80s hits top 100 songs",
        "70s hits top 100 songs"
    ]

    # Dictionary to store playlist information
    playlist_info = {}

    # Search for each playlist and retrieve its ID
    for query in playlist_queries:
        time.sleep(1)
        print(f"Searching for playlist: \"{query}\"...")

        # Set up search parameters
        search_url = "https://api.spotify.com/v1/search"
        params = {
```

```
            "q": query,
            "type": "playlist",
            "limit": 1
        }

        # Make the search request
        response = requests.get(f"{search_url}?{urlencode(params)}", headers=headers)

        if response.status_code == 200:
            search_data = response.json()

            # Check if any playlists were found
            if search_data['playlists']['items']:
                # Get the first matching playlist
                first_result = search_data['playlists']['items'][0]
                playlist_name = first_result['name']
                playlist_id = first_result['id']

                print(f"Found playlist: \"{playlist_name}\" with ID: {playlist_id}")

                # Store the playlist info
                playlist_info[playlist_name] = playlist_id
            else:
                print(f"No playlists found for query: \"{query}\"")
        else:
            print(f"Error searching for playlist: {response.status_code}")
            print(response.text)

    print(f"Found {len(playlist_info)} playlists\n")


    all_tracks = []

    # Get tracks from the playlist
    for playlist_name, playlist_id in playlist_info.items():
        playlist_tracks_url = f"https://api.spotify.com/v1/playlists/{playlist_id}/tracks"
        page = 0

        print(f"Retrieving tracks from playlist \"{playlist_name}\"...")

        # Get all pages
        while playlist_tracks_url:
            page += 1
            if page > 50:  # Just a failsafe to prevent excessive API calls
                break

            # To comply with Spotify API rate limits
            time.sleep(1)

            response = requests.get(playlist_tracks_url, headers=headers)
```

```
            if response.status_code == 200:
                data = response.json()
                track_items = data.get("items", [])

                for track_info in track_items:
                    track = track_info.get("track", {})
                    all_tracks.append(track)
                print(f"Getting page {page} of tracks from playlist \"{playlist_name}\"...")
                playlist_tracks_url = data.get("next")

            else:
                print("Error fetching tracks:", response.status_code)
                break

    print(f"Retrieved all tracks from Spotify playlist(s): ")
    print(f"Total Tracks Retrieved: {len(all_tracks)}\n")

    return all_tracks


# Takes columns we want to keep, and returns a dataframe
def parse_spotify_tracks(spot_tracks):
    print(f"Parsing Spotify data to keep relevant columns...\n")
    all_track_details = []

    # Only keep columns with data we find interesting for tables
    for track in spot_tracks:
        track_details = {}

        # Keep top artist and list of all artists separately
        artists = track.get("artists", None)
        if artists:
            top_artist = artists[0]["name"]
            track_details["top_artist"] = top_artist
            artist_names = [artist.get("name", "") for artist in artists] # Empty is OK here
            track_details["artists"] = ", ".join(artist_names)
        else:
            track_details["top_artist"] = None
            track_details["artists"] = None

        track_details["song_name"] = track.get("name", None)
        track_details["duration"] = track.get("duration_ms", None)
        track_details["popularity"] = track.get("popularity", None)
        track_details["spotify_id"] = track.get("id", None)


        album = track.get("album", None)
        if album:
            album_name = album.get("name", None)
            track_details["album_name"] = album_name
```

```
            track_details["album_id"] = album.get("id", None)
            track_details["album_release_date"] = album.get("release_date", None)
            track_details['album_release_date_precision'] = album.get("release_date_precisi-
on", None)
            images = album.get("images", [])
            if len(images) > 1:
                track_details["album_image"] = images[1].get("url", None)
            else:
                track_details["album_image"] = None
        else:
            track_details["album_name"] = None
            track_details["album_id"] = None
            track_details["album_release_date"] = None
            track_details['album_release_date_precision'] = None
            track_details["album_image"] = None

        track_details["explicit_lyrics"] = track.get("explicit", None)
        track_details["isrc"] = track.get("external_ids", {}).get("isrc", None)
        track_details["spotify_url"] = track.get("external_urls", {}).get("spotify", None)
        track_details["available_markets"] = ", ".join(track.get("available_markets", None))

        all_track_details.append(track_details)

    df = pd.DataFrame(all_track_details)

    # Make a timestamped csv for record
    # Saves to data folder
    data_dir = os.path.join(os.getcwd(), 'data')
    os.makedirs(data_dir, exist_ok=True)
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    spot_tracks_file_name = f"spotify_tracks_{timestamp}.csv"
    spot_tracks_file_path = os.path.join(data_dir, spot_tracks_file_name)
    df.to_csv(spot_tracks_file_path, index=False, encoding='utf-8-sig')
    print(f"Successfully wrote to {spot_tracks_file_name}\n")

    return df

# A few cases where only year is listed
# Need to prepend day and month
# Since to_datetime doesn't work with just year
def convert_date(date_str):
    if pd.isna(date_str):
        return None
    date_str = str(date_str).strip()
    if len(date_str) == 4 and date_str.isdigit():
        return pd.to_datetime(f"01/01/{date_str}", errors='coerce')
    return pd.to_datetime(date_str, errors='coerce')


# Function to replace accented characters
```

```
246 def replace_accented_characters(s):
247     if isinstance(s, str):  # Check if the value is a string
248         normalized_string = unicodedata.normalize('NFD', s)
249         return ''.join(c for c in normalized_string if unicodedata.category(c) != 'Mn')
250     return s
251 
252 
253 # Further clean the spotify tracks dataframe
254 def clean_spotify_tracks(df):
255     print(f"Cleaning Spotify data...\n")
256 
257     spot_df = df.copy()
258     spot_df.replace('', None, inplace=True)
259 
260     print(f"Current number of tracks: {len(spot_df)}")
261     print(f"Dropping duplicates where playlists overlap...")
262 
263     # isrc is great duplicate key for this spotify data
264     duplicate_count = spot_df.duplicated(subset='isrc', keep=False).sum()
265     print(f"Number of 'isrc' duplicates: {duplicate_count}")
266     spot_df = spot_df.drop_duplicates(subset='isrc', keep='first')
267     print(f"Removed {duplicate_count} duplicates")
268     print(f"Remaining number of tracks: {len(spot_df)}\n")
269 
270     # Clean a few specific columns
271     cols_to_clean = ['top_artist', 'artists', 'song_name', 'album_name']
272     for col in cols_to_clean:
273         if col in spot_df.columns:
274             spot_df[col] = spot_df[col].str.lower().str.strip()
275             spot_df[col] = spot_df[col].apply(replace_accented_characters)
276 
277     # Make sure integer columns are ints
278     int_columns = ['duration', 'popularity']
279     for col in int_columns:
280         spot_df[col] = pd.to_numeric(spot_df[col], errors='coerce')
281     spot_df[int_columns] = spot_df[int_columns].where(pd.notnull(spot_df[int_columns]), None)
282 
283     # Force int columns to Int64 (needed for one column that was being read as float)
284     for col in int_columns:
285         spot_df[col] = spot_df[col].astype('Int64')
286 
287     # Replace & with and - helps with matching
288     spot_df['top_artist'] = spot_df['top_artist'].str.replace('&', 'and')
289     spot_df['artists'] = spot_df['artists'].str.replace('&', 'and')
290 
291     # Helps with matching - removes " (feat. *)"
292     spot_df['song_name'] = spot_df['song_name'].str.replace(r' \(feat\..*?\)', '',
regex=True)
293 
294     # Delete if no artist
```

```
    spot_df = spot_df[spot_df['artists'].notna() & (spot_df['artists'] != '')] # Delete if no artist

    spot_df['album_release_date'] = spot_df['album_release_date'].apply(convert_date)

    # Checking for any artists, song_name duplicates
    print("Checking for any duplicates by top_artist and song_name...")
    duplicate_count = spot_df.duplicated(subset=['top_artist', 'song_name']).sum()
    print(f"Number of duplicates in 'top_artist' and 'song_name': {duplicate_count}")
    spot_df = spot_df.drop_duplicates(subset=['top_artist', 'song_name'])
    print(f"Removed {duplicate_count} duplicates")
    print(f"Remaining number of tracks: {len(spot_df)}\n")

    # Sort by top artist and song name
    # Not needed for storage, but helps with debugging
    spot_df.sort_values(by=['top_artist', 'song_name'], inplace=True)

    # Check for similar entries (adjacent rows after sorting)
    print("Checking for similar songs based on character matching...")
    indices_to_drop = []
    similar_count = 0

    # Efficient way to check adjacent rows for duplicates
    # Keep the more popular one
    for i in range(len(spot_df) - 1):  # Iterate through all rows except the last one
        current_row = spot_df.iloc[i]
        next_row = spot_df.iloc[i + 1]

        # Get the values to compare
        current_artist = str(current_row['top_artist'])
        next_artist = str(next_row['top_artist'])
        current_song = str(current_row['song_name'])
        next_song = str(next_row['song_name'])

        # Compare full length or first 10 characters - 10 is arbitrary
        artist_compare_len = min(len(current_artist), len(next_artist), 10)
        song_compare_len = min(len(current_song), len(next_song), 10)
        if song_compare_len == 0:  # Skip if either song is empty
            continue

        # Check if both beginning parts match
        if (current_artist[:artist_compare_len] == next_artist[:artist_compare_len] and
            current_song[:song_compare_len] == next_song[:song_compare_len]):

            # Found a potential duplicate, decide which one to keep based on popularity
            current_popularity = current_row.get('popularity', 0)
            next_popularity = next_row.get('popularity', 0)

            # If next row has higher popularity, drop current row
            if next_popularity > current_popularity:
```

```
                indices_to_drop.append(spot_df.index[i])
                similar_count += 1
                print(f"Similar songs found: '{current_song}' and '{next_song}' by '{current_artist}'")
                print(f"  Keeping '{next_song}' (popularity: {next_popularity})")
                print(f"  Dropping '{current_song}' (popularity: {current_popularity})")
            # If current row has higher or equal popularity, drop next row
            else:
                indices_to_drop.append(spot_df.index[i + 1])
                similar_count += 1
                print(f"Similar songs found: '{current_song}' and '{next_song}' by '{current_artist}'")
                print(f"  Keeping '{current_song}' (popularity: {current_popularity})")
                print(f"  Dropping '{next_song}' (popularity: {next_popularity})")

    # Drop the identified duplicates
    if similar_count > 0:
        spot_df = spot_df.drop(indices_to_drop)
        print(f"Removed {similar_count} similar songs based on character matching")
    else:
        print("No similar songs found")

    print(f"Remaining number of Spotify tracks: {len(spot_df)}\n")


    data_dir = os.path.join(os.getcwd(), 'data')
    os.makedirs(data_dir, exist_ok=True)
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    spot_clean_file_name = f"spotify_tracks_cleaned_{timestamp}.csv"
    spot_clean_file_path = os.path.join(data_dir, spot_clean_file_name)
    spot_df.to_csv(spot_clean_file_path, index=False, encoding='utf-8-sig')
    print(f"Successfully processed Spotify tracks...")
    print(f"Successfully wrote record to {spot_clean_file_name}\n")

    return spot_df



# Calls other functions to get and clean spotify data
# Then loads it into postgres
def load_spotify_data():

    raw_spot_tracks = get_spot_tracks()
    df = parse_spotify_tracks(raw_spot_tracks)
    clean_spot_df = clean_spotify_tracks(df)
    num_tracks = len(clean_spot_df)

    # Get postgres connection
    pg_hook = PostgresHook(postgres_conn_id='pg_group6')
    with pg_hook.get_conn() as conn:
```

```
        with conn.cursor() as cur:
            for index, row in clean_spot_df.iterrows():
                group6_id = uuid.uuid5(uuid.NAMESPACE_DNS, str(row['song_name'].strip() + row['top_artist'].strip()))

                cur.execute("""INSERT INTO spotify_tracks (group6_id, top_artist, artists, song_name, duration,
                            popularity, spotify_id, album_name, album_id, album_release_date, album_release_date_precision,
                            album_image, explicit_lyrics, isrc, spotify_url, available_markets)
                            VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)""",
                            (group6_id,
                            row['top_artist'],
                            row['artists'].split(','),
                            row['song_name'],
                            row['duration'],
                            row['popularity'],
                            row['spotify_id'],
                            row['album_name'],
                            row['album_id'],
                            row['album_release_date'],
                            row['album_release_date_precision'],
                            row['album_image'],
                            row['explicit_lyrics'],
                            row['isrc'],
                            row['spotify_url'],
                            row['available_markets']))
                conn.commit()

            print(f"Saved {num_tracks} Spotify tracks to database")


with DAG(
        'spotify_dag',
        default_args=default_args,
        description='Dag for spotify data',
        schedule_interval=None,
        catchup=False
    ) as spotify_dag:

    start_task = EmptyOperator(task_id='start')

    create_tables = PostgresOperator(
        task_id='create_tables',
        postgres_conn_id='pg_group6',
        sql='sql/spotify_create.sql'
    )

    load_spotify_data = PythonOperator(
```

```
438         task_id='load_spotify_data',
439         python_callable=load_spotify_data
440     )
441 
442     end_task = EmptyOperator(task_id='end')
443 
444     start_task >> create_tables >> load_spotify_data >> end_task
445 
446 
```