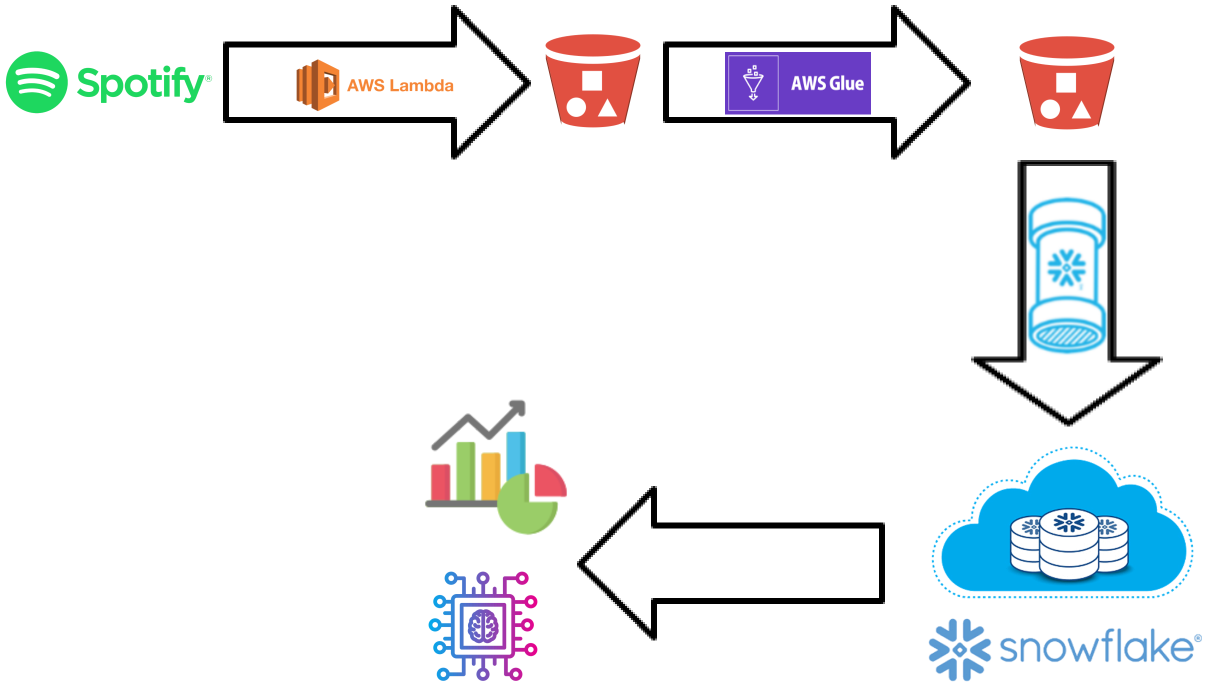
SPOTIFY DATA ENGINEER PROJECT

1. Problem Statement

This project will ingest data from Spotify API into AWS S3 as raw files. Afterward, This dataset will be transformed into readable format and stored in another storage. Finally, this transformed data will be loaded into data warehouse Snowflake for analysis or building machine learning models purposes.

All processes must be automated.

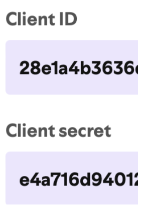
2. Diagram of this project



3. Process

3.1. Ingest data from Spotify API

Firstly, we need to register an account for developer on Spotify website
https://developer.spotify.com/
After creating the app, we need to store Client ID and Client Secret information, it will be used to connect between AWS Lambda and Spotify API to get data sources.

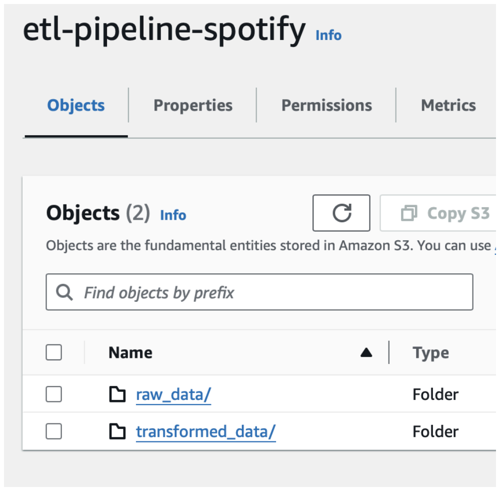

Client ID

28e1a4b3636

Client secret

e4a716d9401

Secondly, we need to register an AWS account and create an AWS Bucket, this bucket will be used to store raw data files.


etl-pipeline-spotify Info

Objects (2) Info

Objects are the fundamental entities stored in Amazon S3. You can use

Name	Type
raw_data/	Folder
transformed_data/	Folder

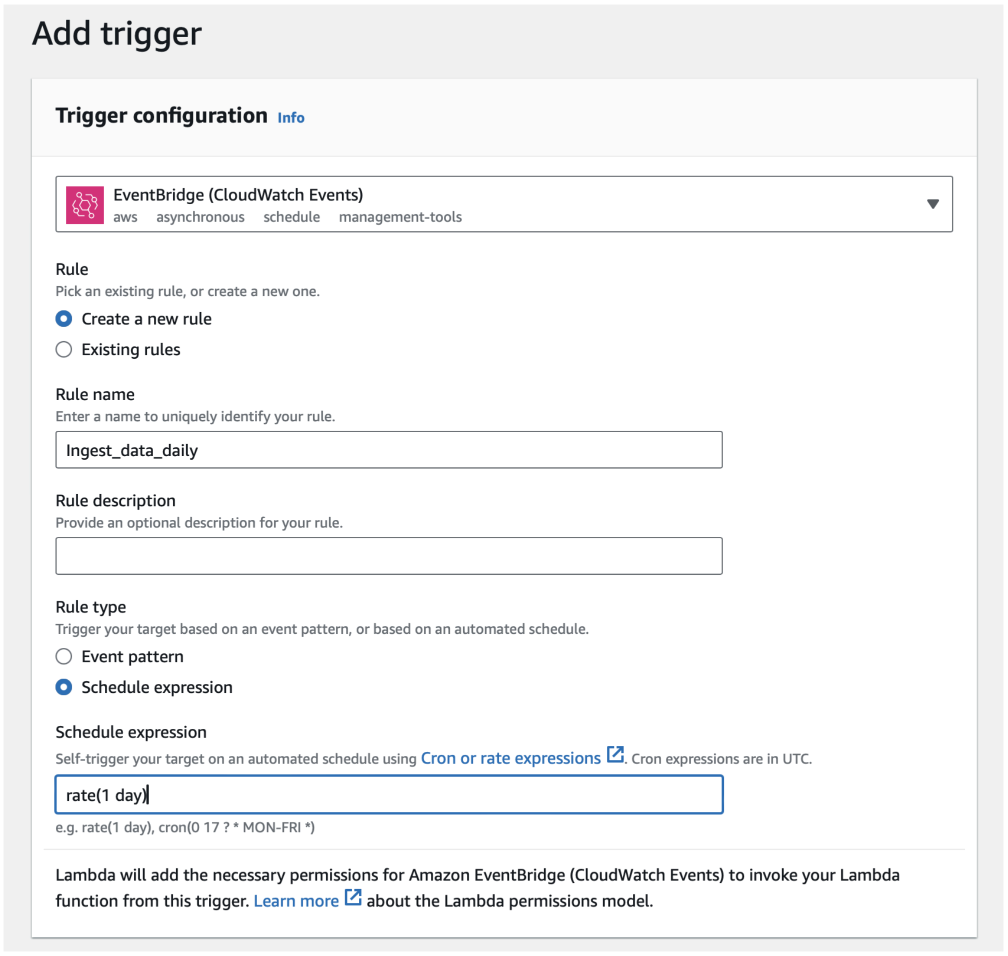
Thirdly, we prepare the AWS Lambda function to connect to the Spotify API and get data sources. The **client_id** and **client_secret** must be defined in **Environment Variables**.

```python
import json
import os
import spotipy
from spotipy.oauth2 import SpotifyClientCredentials
from datetime import datetime
import boto3

def lambda_handler(event, context):
    client_id = os.environ.get('client_id')
    client_secret = os.environ.get('client_secret')
    client_credentials_manager = SpotifyClientCredentials(client_id=client_id, client_secret=client_secret)
    sp = spotipy.Spotify(client_credentials_manager=client_credentials_manager)

    link = 'https://open.spotify.com/playlist/37i9dQZF1DWZ7eJRBxKzdO'
    playlist_URI = link.split('/')[-1]
    data = sp.playlist_tracks(playlist_URI)

    filename = 'raw_spotify_' + str(datetime.now()) + '.json'

    client = boto3.client('s3')
    client.put_object(
        Bucket='etl-pipeline-spotify',
        Key='raw_data/to_processed/' + filename,
        Body=json.dumps(data)
        )
```

After reading and storing data from Spotify API, we will trigger the transformation Glue Job (in point 3.2) by the following code:

```python
glue = boto3.client('glue')
gluejobname = 'spotify_etl_job'

try:
    runId = glue.start_job_run(JobName=gluejobname)
    status = glue.get_job_run(JobName=gluejobname, RunId=runId['JobRunId'])
    print('Job status: ', status['JobRun']['JobRunState'])
except Exception as e:
    print(e)
```

We will use EventBridge (CloudWatch) to set a schedule to execute the Lambda function. In this project, I set it daily, so the data sources from Spotify API will be ingested into AWS S3 everyday.



Add trigger

Trigger configuration Info

EventBridge (CloudWatch Events)
aws asynchronous schedule management-tools

Rule
Pick an existing rule, or create a new one.
[✓ Create a new rule]
Existing rules

Rule name
Enter a name to uniquely identify your rule.
Ingest_data_daily

Rule description
Provide an optional description for your rule.

Rule type
Trigger your target based on an event pattern, or based on an automated schedule.
Event pattern
[✓ Schedule expression]

Schedule expression
Self-trigger your target on an automated schedule using Cron or rate expressions. Cron expressions are in UTC.
rate(1 day)
e.g. rate(1 day), cron(0 17 ? * MON-FRI *)

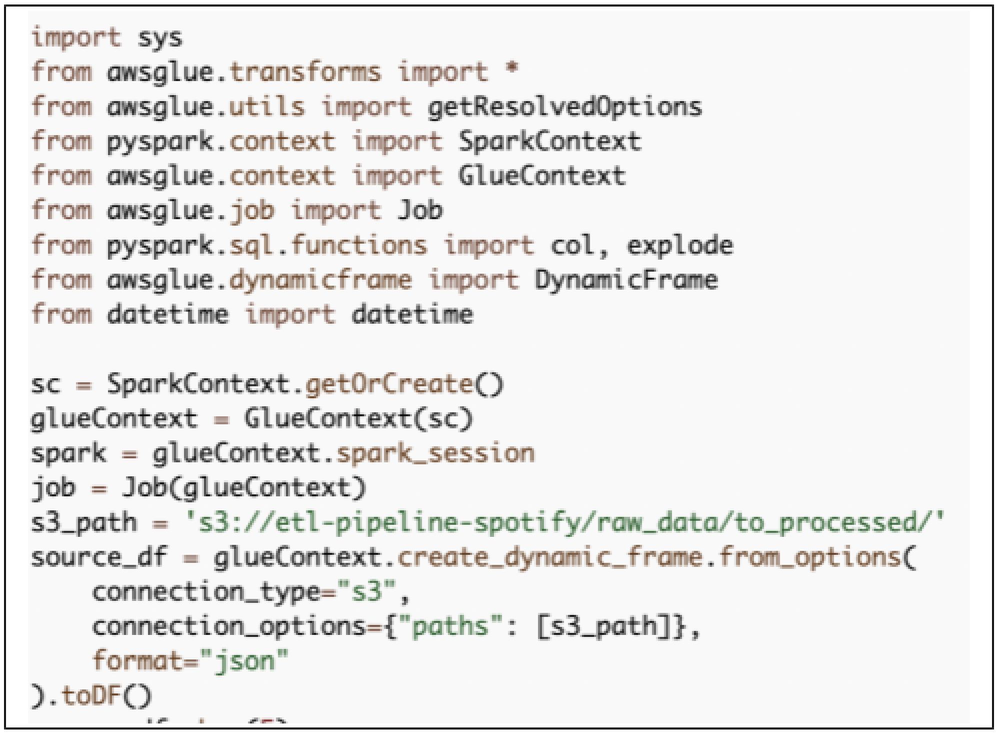
Lambda will add the necessary permissions for Amazon EventBridge (CloudWatch Events) to invoke your Lambda function from this trigger. Learn more about the Lambda permissions model.

3.2. Transform raw data file into readable data

In this step, we will use AWS Glue for the transformation step in the ETL process.

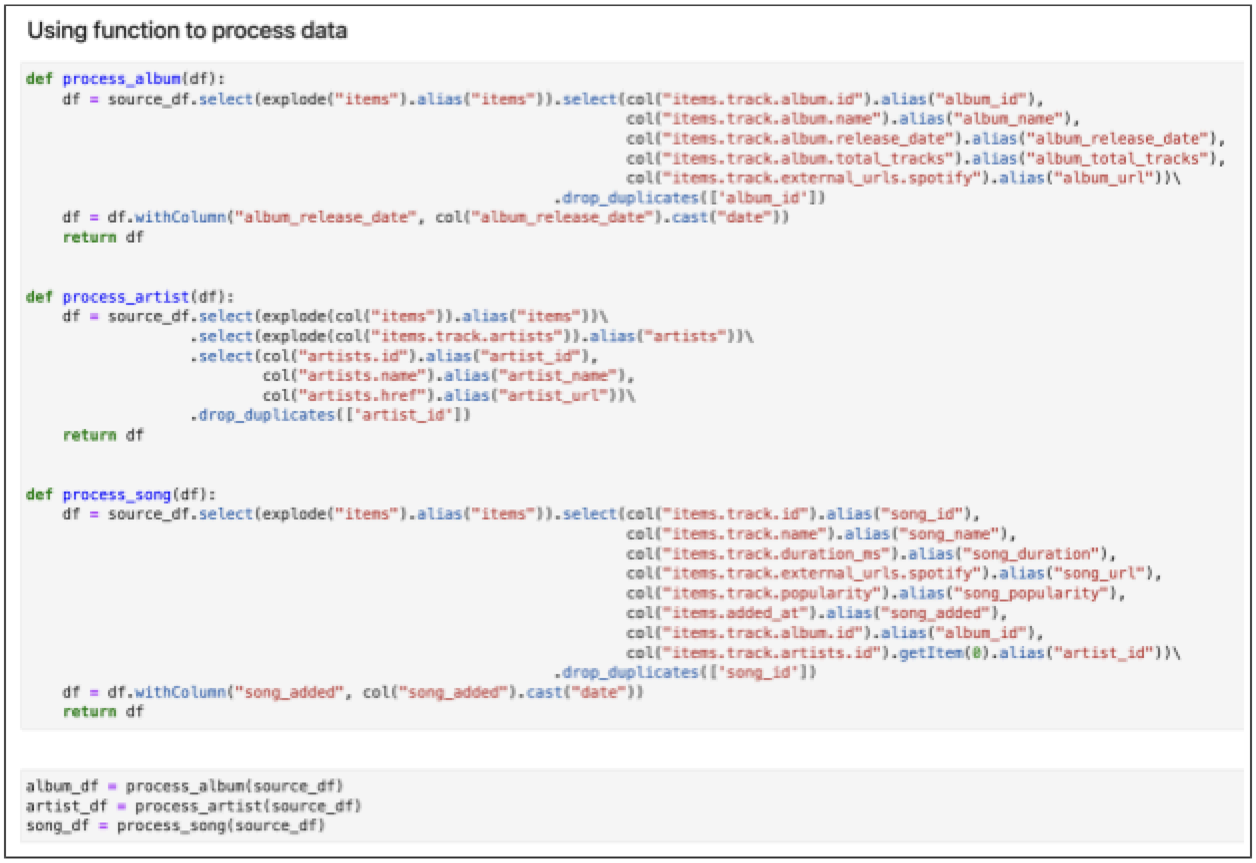
In AWS Glue, we can create Notebook to process data step by step, it will be automatically converted into a script.

Firstly, we need to import some necessary libraries, then continue with initializing SparkContext and GlueContext. They are some kinds of protocol that we use to connect with Spark and Glue. Afterward, we read the raw file (json) in S3 and convert it to Dataframe.



```
import sys
from awsglue.transforms import *
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job
from pyspark.sql.functions import col, explode
from awsglue.dynamicframe import DynamicFrame
from datetime import datetime

sc = SparkContext.getOrCreate()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
s3_path = 's3://etl-pipeline-spotify/raw_data/to_processed/'
source_df = glueContext.create_dynamic_frame.from_options(
    connection_type="s3",
    connection_options={"paths": [s3_path]},
    format="json"
).toDF()
```

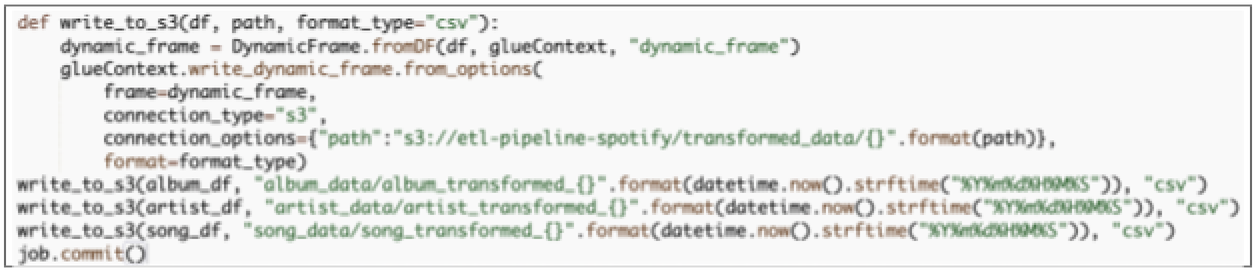
We will need some functions to process data for corresponding tables.


Using function to process data

```python
def process_album(df):
    df = source_df.select(explode("items").alias("items")).select(col("items.track.album.id").alias("album_id"),
                                                                  col("items.track.album.name").alias("album_name"),
                                                                  col("items.track.album.release_date").alias("album_release_date"),
                                                                  col("items.track.album.total_tracks").alias("album_total_tracks"),
                                                                  col("items.track.external_urls.spotify").alias("album_url"))\
                                                          .drop_duplicates(['album_id'])
    df = df.withColumn("album_release_date", col("album_release_date").cast("date"))
    return df


def process_artist(df):
    df = source_df.select(explode(col("items")).alias("items"))\
                  .select(explode(col("items.track.artists")).alias("artists"))\
                  .select(col("artists.id").alias("artist_id"),
                          col("artists.name").alias("artist_name"),
                          col("artists.href").alias("artist_url"))\
                  .drop_duplicates(['artist_id'])
    return df


def process_song(df):
    df = source_df.select(explode("items").alias("items")).select(col("items.track.id").alias("song_id"),
                                                                  col("items.track.name").alias("song_name"),
                                                                  col("items.track.duration_ms").alias("song_duration"),
                                                                  col("items.track.external_urls.spotify").alias("song_url"),
                                                                  col("items.track.popularity").alias("song_popularity"),
                                                                  col("items.added_at").alias("song_added"),
                                                                  col("items.track.album.id").alias("album_id"),
                                                                  col("items.track.artists.id").getItem(0).alias("artist_id"))\
                                                          .drop_duplicates(['song_id'])
    df = df.withColumn("song_added", col("song_added").cast("date"))
    return df
```

```python
album_df = process_album(source_df)
artist_df = process_artist(source_df)
song_df = process_song(source_df)
```

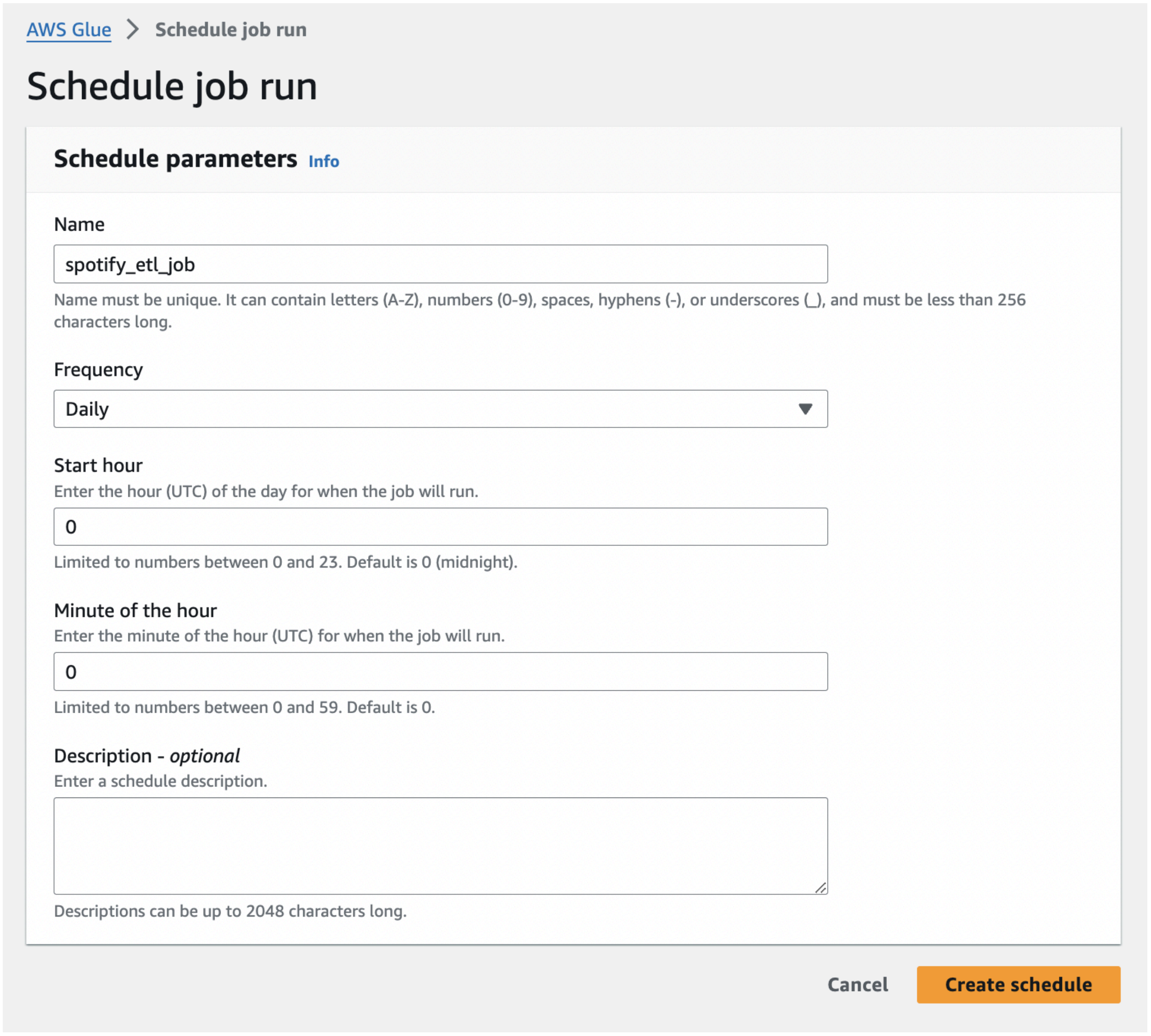
Finally, we write the Dataframes into CSV files and store them in AWS S3.



```python
def write_to_s3(df, path, format_type="csv"):
    dynamic_frame = DynamicFrame.fromDF(df, glueContext, "dynamic_frame")
    glueContext.write_dynamic_frame.from_options(
        frame=dynamic_frame,
        connection_type="s3",
        connection_options={"path":"s3://etl-pipeline-spotify/transformed_data/{}".format(path)},
        format=format_type)
write_to_s3(album_df, "album_data/album_transformed_{}".format(datetime.now().strftime("%Y%m%d%H%M%S")), "csv")
write_to_s3(artist_df, "artist_data/artist_transformed_{}".format(datetime.now().strftime("%Y%m%d%H%M%S")), "csv")
write_to_s3(song_df, "song_data/song_transformed_{}".format(datetime.now().strftime("%Y%m%d%H%M%S")), "csv")
job.commit()
```

Do not forget to commit the job at the end.
We need to run this job to make sure it works well before setting it to run automatically by the Schedule feature of Glue.

We do not need to set the Schedule for the Glue job if we need it to run followed by the Ingest step (3.1) as in the 3.1 step, we set the Glue job to be triggered whenever the Lambda function is called.



Schedule job run

Schedule parameters Info

Name

spotify_etl_job

Name must be unique. It can contain letters (A-Z), numbers (0-9), spaces, hyphens (-), or underscores (_), and must be less than 256 characters long.

Frequency

Daily

Start hour

Enter the hour (UTC) of the day for when the job will run.

0

Limited to numbers between 0 and 23. Default is 0 (midnight).

Minute of the hour

Enter the minute of the hour (UTC) for when the job will run.

0

Limited to numbers between 0 and 59. Default is 0.

Description - *optional*

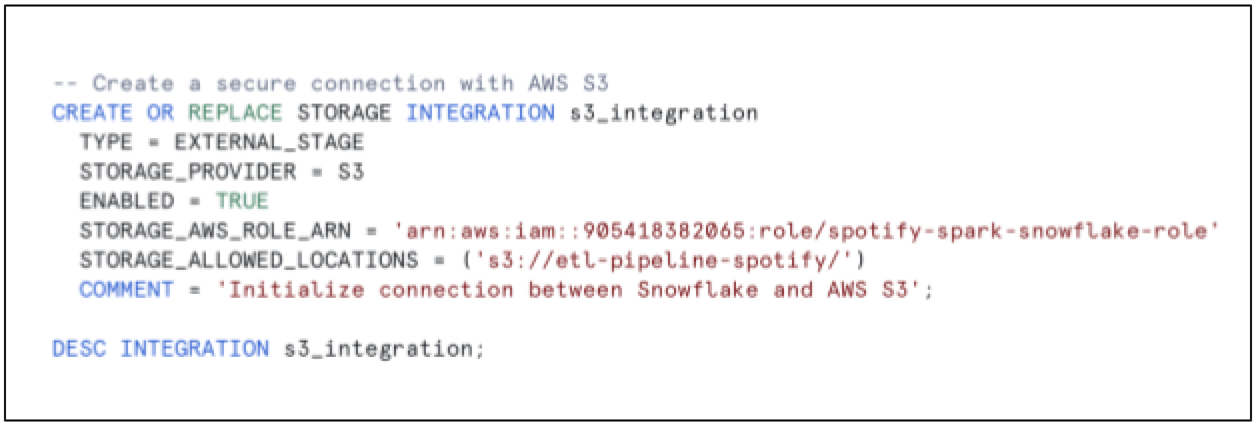
Enter a schedule description.

Descriptions can be up to 2048 characters long.

3.3. Loading transformed data into Data Warehouse (Snowflake database)

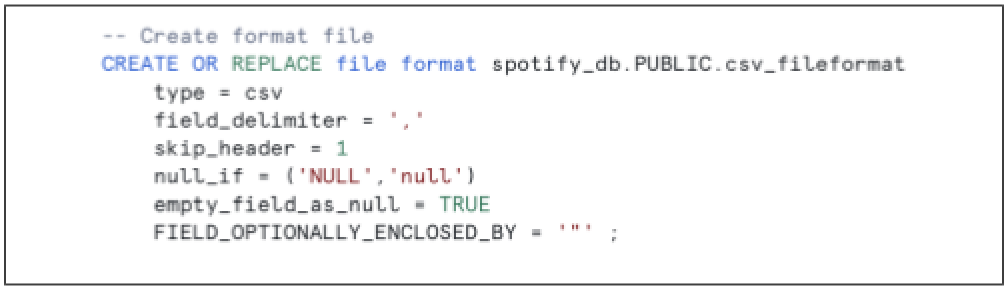
The most important thing to load data from AWS S3 to the external platform, in this project is the Data Warehouse Snowflake, is to create a secure connection between AWS and this external platform.

- Create an IAM role with the S3 Access right.
- In Snowflake, create an INTEGRATION with STORAGE_AWS_ROLE_ARN contains information from ARN of the IAM role above.
- When the INTEGRATION was created, take the information of STORAGE_AWS_IAM_USER_ARN and STORAGE_AWS_EXTERNAL_ID.
- Paste them back the Trust Policy of the IAM Role.
- Now the Snowflake Data Warehouse can connect to AWS S3 to get data.



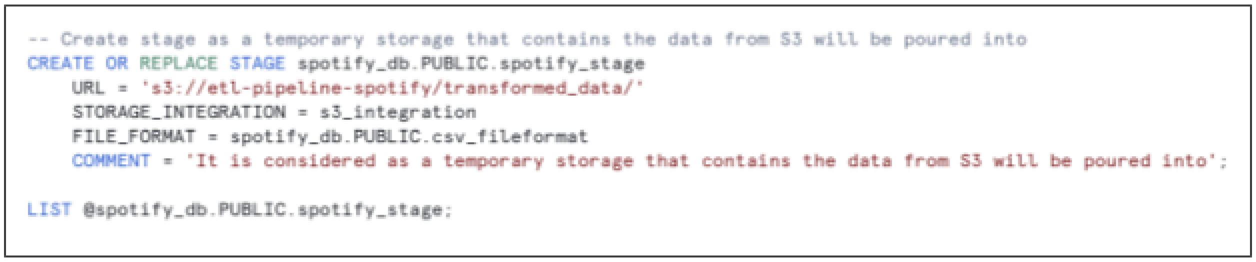
```sql
-- Create a secure connection with AWS S3
CREATE OR REPLACE STORAGE INTEGRATION s3_integration
  TYPE = EXTERNAL_STAGE
  STORAGE_PROVIDER = S3
  ENABLED = TRUE
  STORAGE_AWS_ROLE_ARN = 'arn:aws:iam::905418382065:role/spotify-spark-snowflake-role'
  STORAGE_ALLOWED_LOCATIONS = ('s3://etl-pipeline-spotify/')
  COMMENT = 'Initialize connection between Snowflake and AWS S3';

DESC INTEGRATION s3_integration;
```

In the next step, we will create a file format corresponding to the files in AWS S3 which will be loaded into Snowflake.



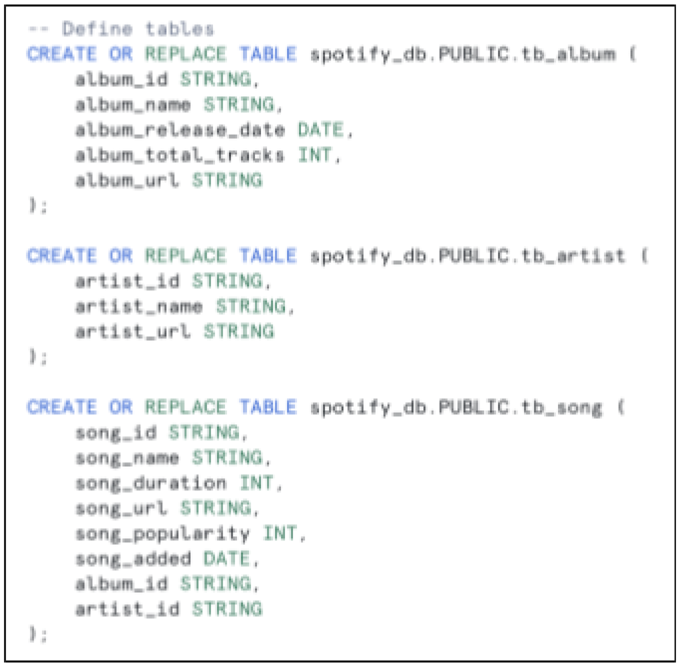
```sql
-- Create format file
CREATE OR REPLACE file format spotify_db.PUBLIC.csv_fileformat
    type = csv
    field_delimiter = ','
    skip_header = 1
    null_if = ('NULL','null')
    empty_field_as_null = TRUE
    FIELD_OPTIONALLY_ENCLOSED_BY = '"' ;
```

The final step in the preparation process is to create an INTEGRATION, which is considered as a storage to store data from AWS S3.



```sql
-- Create stage as a temporary storage that contains the data from S3 will be poured into
CREATE OR REPLACE STAGE spotify_db.PUBLIC.spotify_stage
    URL = 's3://etl-pipeline-spotify/transformed_data/'
    STORAGE_INTEGRATION = s3_integration
    FILE_FORMAT = spotify_db.PUBLIC.csv_fileformat
    COMMENT = 'It is considered as a temporary storage that contains the data from S3 will be poured into';

LIST @spotify_db.PUBLIC.spotify_stage;
```

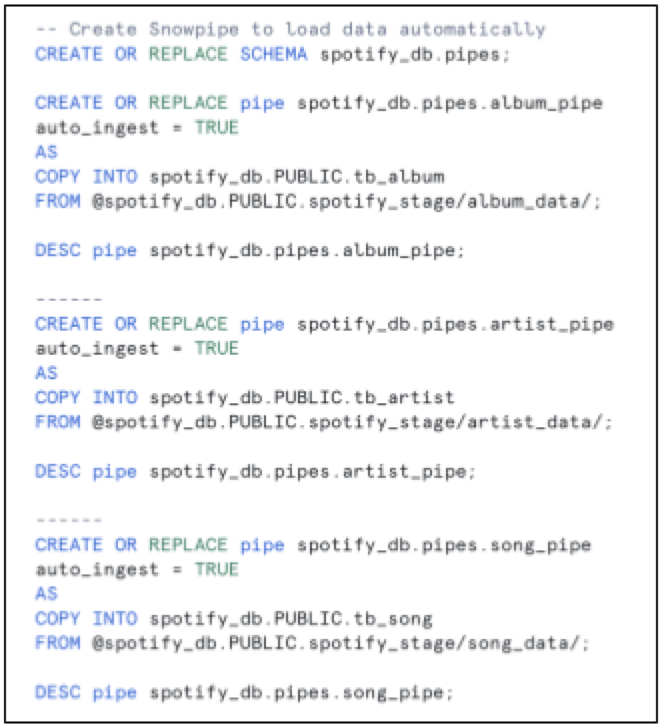
After the preparation steps are done, we will start to define the necessary tables with the structures corresponding to the data files in AWS S3.



```sql
-- Define tables
CREATE OR REPLACE TABLE spotify_db.PUBLIC.tb_album (
    album_id STRING,
    album_name STRING,
    album_release_date DATE,
    album_total_tracks INT,
    album_url STRING
);

CREATE OR REPLACE TABLE spotify_db.PUBLIC.tb_artist (
    artist_id STRING,
    artist_name STRING,
    artist_url STRING
);

CREATE OR REPLACE TABLE spotify_db.PUBLIC.tb_song (
    song_id STRING,
    song_name STRING,
    song_duration INT,
    song_url STRING,
    song_popularity INT,
    song_added DATE,
    album_id STRING,
    artist_id STRING
);
```

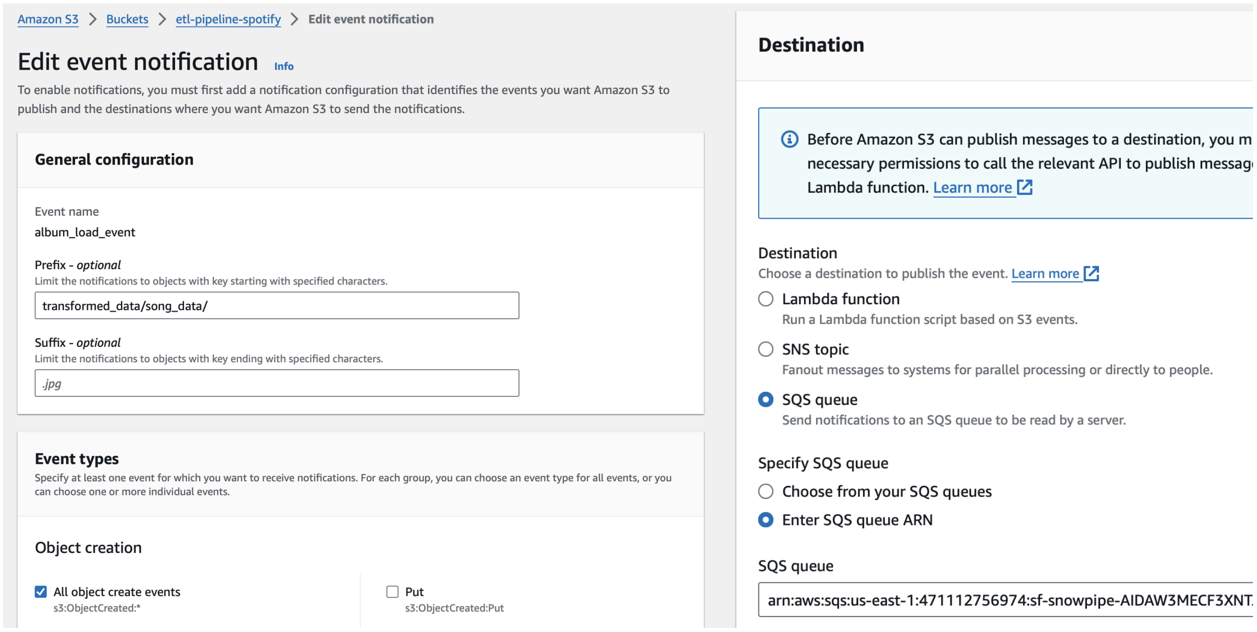
We can try to test by manually loading data into those tables to ensure the connection and the table structures were created correctly.

Finally, we create Snowpipe which is used to load data automatically from AWS S3 to Snowflake whenever a new file is uploaded into AWS S3.



```sql
-- Create Snowpipe to load data automatically
CREATE OR REPLACE SCHEMA spotify_db.pipes;

CREATE OR REPLACE pipe spotify_db.pipes.album_pipe
auto_ingest = TRUE
AS
COPY INTO spotify_db.PUBLIC.tb_album
FROM @spotify_db.PUBLIC.spotify_stage/album_data/;

DESC pipe spotify_db.pipes.album_pipe;

------
CREATE OR REPLACE pipe spotify_db.pipes.artist_pipe
auto_ingest = TRUE
AS
COPY INTO spotify_db.PUBLIC.tb_artist
FROM @spotify_db.PUBLIC.spotify_stage/artist_data/;

DESC pipe spotify_db.pipes.artist_pipe;

------
CREATE OR REPLACE pipe spotify_db.pipes.song_pipe
auto_ingest = TRUE
AS
COPY INTO spotify_db.PUBLIC.tb_song
FROM @spotify_db.PUBLIC.spotify_stage/song_data/;

DESC pipe spotify_db.pipes.song_pipe;
```

There is one more step that needs to be configured for automatic loading data is to create an Event in AWS S3. We will use SQS queue with information from the notification_channel of the corresponding Snowpipe.



Amazon S3 > Buckets > etl-pipeline-spotify > Edit event notification

Edit event notification Info

To enable notifications, you must first add a notification configuration that identifies the events you want Amazon S3 to publish and the destinations where you want Amazon S3 to send the notifications.

General configuration

Event name
album_load_event

Prefix - *optional*
Limit the notifications to objects with key starting with specified characters.
transformed_data/song_data/

Suffix - *optional*
Limit the notifications to objects with key ending with specified characters.
.jpg

Event types
Specify at least one event for which you want to receive notifications. For each group, you can choose an event type for all events, or you can choose one or more individual events.

Object creation

[✓ All object create events] s3:ObjectCreated:*
Put s3:ObjectCreated:Put

Destination

Before Amazon S3 can publish messages to a destination, you m
necessary permissions to call the relevant API to publish messag
Lambda function. Learn more

Destination
Choose a destination to publish the event. Learn more

Lambda function
Run a Lambda function script based on S3 events.

SNS topic
Fanout messages to systems for parallel processing or directly to people.

[✓ SQS queue]
Send notifications to an SQS queue to be read by a server.

Specify SQS queue
Choose from your SQS queues
[✓ Enter SQS queue ARN]

SQS queue
arn:aws:sqs:us-east-1:471112756974:sf-snowpipe-AIDAW3MECF3XNT