Candidates Hiring Analysis – ETL

Project Documentation

Performed by:

Nicolas Peña Irurita - 2232049

Índice

9. **Documentation**
   9.1 Summary
   9.2 Key Takeaways
10. **Conclusion & Next Steps**

Títulos y Subtítulos

1. Introduction

- Overview of the ETL project
- Goals and objectives

2. Project Structure

2.1 Repository Structure

- README.md
- .gitignore
- DAGs directory

3. Exploratory Data Analysis (EDA)

- Data insights from Spotify dataset
- Data insights from Grammy dataset
- API response structure analysis

4. Extract Phase

4.1 Extracting Spotify Data (CSV)

- Reading and preprocessing the CSV file

4.2 Extracting Grammy Data (Database)

- SQL queries to fetch Grammy data

4.3 Extracting Data from API

- API request handling

5. Transform Phase

5.1 Data Cleaning

- Handling missing values, duplicates, and standardizing formats

5.2 Data Normalization

- Merging datasets and fuzzy matching

6. Load Phase

6.1 Storing in a Database

- Creating and storing transformed data in SQL

6.2 Saving to Google Drive (CSV)

- Exporting final dataset to Google Drive

7. Airflow Pipeline Implementation

7.1 DAG Overview

- Task dependencies and workflow automation

8. Visualization & Graphs

8.1 Key Charts

- Spotify genre trends
- Grammy award trends
- Correlation between audio features and awards

9. Documentation

9.1 Summary

- Recap of the ETL pipeline

9.2 Key Takeaways

- Achievements and results

10. Conclusion & Next Steps

- Challenges faced
- Future improvements

1. Introduction

Overview of the Project

This project involves building an **ETL pipeline using Apache Airflow** to automate the extraction, transformation, and loading (ETL) of music-related data from **three different sources**:

1. **Spotify Dataset (CSV)** – Contains audio features and popularity metrics for various tracks.
2. **Grammy Awards Dataset (Database - PostgreSQL)** – Provides historical data on Grammy winners and nominees.
3. **Billboard API** – Supplies real-time and historical chart data, tracking the most popular songs and artists over time.

The objective is to **clean, process, and merge these datasets**, store the final data in **PostgreSQL**, and generate **interactive visualizations** using Tableau to analyze trends in **music popularity, award patterns, and track characteristics**.

Project Workflow

The ETL process follows these key steps:

1. **Extraction**: Gather data from the three sources (**Spotify CSV, Grammy Database, Billboard API**).
2. **Transformation**: Clean, normalize, and integrate datasets (handling missing values, duplicates, and standardizing formats).
3. **Loading**: Store the processed data in a **PostgreSQL database** and export a CSV file to **Google Drive**.
4. **Visualization**: Generate insights through a **dashboard in Tableau**, analyzing music trends, award-winning characteristics, and artist success over time.

To ensure automation and efficiency, **Apache Airflow** is used to schedule and manage the ETL pipeline inside a **Dockerized environment**.

Datasets Used

1. Spotify Dataset (CSV)

- **Source**: Spotify Tracks Dataset
- **Description**: Contains track metadata and audio features for various songs, including:

    - `track_name`, `artist_name`, `popularity`, `danceability`, `energy`, `tempo`, etc.

2. Grammy Awards Dataset (Database - PostgreSQL)

- **Source**: Grammy Awards Dataset
- **Description**: Provides historical data on Grammy nominations and winners from 1958 to 2019, including:
    - `year`, `category`, `artist`, `winner` (boolean field).

3. Billboard API

- **Source**: Billboard API
- **Description**: Provides real-time and historical **chart rankings** for songs and artists, including:
    - **Billboard Hot 100 positions**, **album rankings**, **top streaming artists**, and **trending songs**.
- **Usage**: This API is used to enrich the dataset by adding **chart performance data**, allowing comparisons between **Spotify streaming trends, Grammy winners, and Billboard rankings**.

Technologies and Tools Used

To efficiently handle **data extraction, transformation, storage, and visualization**, this project utilizes the following technologies:

1. ETL Pipeline & Workflow Orchestration

- **Apache Airflow**: Manages and schedules ETL workflows.
- **Docker**: Runs Airflow in a containerized environment for easy deployment.

2. Data Processing & Transformation

- **Python**: Main programming language for scripting and automation.
- **Polars**: High-performance library for data manipulation, faster than Pandas.
- **Pandas**: Used for data analysis and transformation.
- **Seaborn**: Generates attractive statistical graphics.
- **Matplotlib**: For creating static data visualizations.

3. Database & Storage

- **PostgreSQL**: Chosen database to store processed data.
- **psycopg2**: Python library to connect and interact with PostgreSQL.

- **Google Drive API**: Saves the processed dataset as CSV in Google Drive.

4. Data Visualization

- **Dash From Ploty**: Creates interactive dashboards and visual reports.

2. Project Structure

```
├── dags                  # Airflow DAGs (workflow definitions)
├── data                  # Raw and processed data storage
│   ├── raw               # Raw data files (Spotify CSV, Grammy CSV, Billboard JSON)
│   └── processed         # Transformed data and the final merged dataset
├── src                   # ETL source code
│   ├── extract.py        # Extraction functions for each data source
│   ├── transform.py      # Transformation functions for cleaning and normalization
│   ├── merge_data.py     # Function to merge datasets into one final CSV
│   └── load.py           # Functions to load data into PostgreSQL and upload to Google Drive
├── config                # Additional configuration files
├── plugins               # Airflow plugins (if applicable)
├── logs                  # Airflow logs
├── requirements.txt      # Python dependencies
├── docker-compose.yml    # Docker Compose configuration for Airflow and related services
├── Dockerfile            # Custom Dockerfile for the Airflow image
├── .env                  # Environment variables (DB credentials and others)
└── client_secrets.json   # Google Drive API credentials (OAuth client)
```

2.1 Repository Structure

- README.md
- .gitignore

3. Exploratory Data Analysis (EDA)

Data insights from Spotify dataset

After conducting a comprehensive exploratory data analysis (EDA) and a series of predictive modeling experiments on the Spotify dataset, several key insights and technical takeaways emerged:

Dataset Structure and Integrity

- The dataset contains **114,000 rows** and **21 well-defined features**, including both numerical (e.g., tempo, loudness, danceability) and categorical (e.g., artist, genre, album) variables.
- The data structure is robust:
    - No duplicate rows were found.
    - Only 2 missing values were detected in the artists and album_name columns.
    - Each column has a clear, defined data type and a meaningful interpretation.

Audio and Musical Feature Patterns

- The analysis revealed genre-informed patterns in key audio features:
    - **Tempo** is multimodal, with peaks near 90, 120, and 140 BPM.
    - **Energy** is right-skewed, with the majority of tracks scoring above 0.6.
    - **Danceability** exhibits a normal distribution centered around 0.6.
- The **time_signature** is predominantly 4/4, confirming the prevalence of Western popular music structures.
- A strong correlation exists between **loudness** and energy (correlation coefficient $r \approx 0.76$), while **acousticness** shows a strong negative correlation with energy ($r \approx -0.73$), reflecting predictable relationships within musical physics.

Popularity and Listener Behavior

- The **popularity distribution** is heavily skewed, with most tracks scoring below 40.

- **Popularity** shows weak correlations with individual audio features (absolute correlation values $|r| < 0.10$). For instance:
    - Loudness has a correlation of $r = 0.05$.
    - Danceability exhibits a correlation of $r = 0.04$.
    - Instrumentalness has a correlation of $r = -0.10$.
- This suggests that popularity is influenced more by external factors (such as artist fame or inclusion in popular playlists) rather than by intrinsic musical characteristics.

Entity-Level Trends

- Top-performing **artists** like The Beatles, Arijit Singh, and Pritam exhibit high average popularity, while lesser-known names are prevalent in the lower ranks.
- Many albums appear frequently in the dataset but have low average popularity, particularly compilations or themed releases.
- **Genre insights** indicate that genres such as synth-pop, singer-songwriter, and alternative achieve high maximum popularity scores, whereas genres like iranian, tango, and grindcore tend to have lower ceilings, even with high overall representation.
- These trends demonstrate a **long-tail effect**: while a vast number of genres and tracks exist, only a few consistently dominate user engagement.

Explicit Content and Mode Distribution

- Only **8.6%** of tracks are marked as explicit, indicating that the bulk of the music in the dataset is aimed at a general audience.
- Explicit content is more concentrated in certain genres, particularly comedy, emo, and hip-hop.
- Approximately **63.8%** of the tracks are in a major key, while **36.2%** are in a minor key. This reflects a tonal preference for brighter, more accessible music.

Data Quality Observations

- Outliers are notably present in features like:
    - instrumentalness (with over 25,000 extreme values),
    - speechiness and liveness, which likely capture non-standard or niche content (e.g., podcasts or ambient recordings).

- Many artist and album names include special characters or formatting inconsistencies, likely the result of metadata ingestion challenges.

Predictive Modeling Insights

- Among the various models tried, the **CatBoost Regressor** significantly outperformed other methods:
    - Achieving a Mean Absolute Error (MAE) of 6.00, a Mean Squared Error (MSE) of 75.76, and an R^2 score of **0.85**.
    - CatBoost's native handling of high-cardinality categorical variables (such as artist, album, genre) provided a substantial advantage in performance.
- The **Random Forest** model performed moderately well (R^2: 0.65), while **XGBoost** lagged behind (R^2: 0.60) due to challenges associated with sparse one-hot encoded features.
- These outcomes underline the **critical role of feature preprocessing and model selection** when dealing with complex categorical metadata within music datasets.

Overall Summary
The EDA confirms that Spotify track data is both rich and expressive, yet it highlights the inherent challenges of working with real-world music data:

- **Popularity is a multifactorial phenomenon** that cannot be fully explained by audio features alone.
- **Modeling success heavily depends on how metadata is represented and processed.**
- Tools like **CatBoost**, which respect the native structure of categorical features, are particularly well-suited to this domain.

These insights provide a solid foundation for future work, whether building music recommendation systems, designing trend analysis tools, or developing artist discovery engines. The analysis reinforces that while musical attributes provide valuable context, the overall popularity of a track is driven by a more complex interplay of factors that extend beyond the audio features themselves.

- Data insights from Grammy dataset

Final Conclusion

After a comprehensive exploratory data analysis (EDA) on the Grammy Awards dataset, several key insights and conclusions have emerged:

Dataset Structure and Quality:

- The dataset comprises 4,810 records and 10 columns.
- It contains rich metadata on nominations, categories, and contributors, though approximately 40% of rows miss artist information—particularly in classical or ensemble projects.
- The diversity of categories (638 unique entries) confirms the broad scope of the awards, covering both mainstream and niche genres.

Content Diversity and Category Trends:

- Mainstream categories such as **Song of the Year**, **Record of the Year**, and **Album of the Year** dominate, reaffirming their central role in the Grammy ceremony.
- The frequent appearance of the term “best” in category names underlines standard industry language and structures.
- The dataset also highlights a wide range of niche and technical categories, illustrating the detailed segmentation used by the awards to honor different aspects of music production.

Entity-Level and Temporal Trends:

- High-profile artists (for example, U2, Aretha Franklin, and Beyoncé) are among the most recognized, while ensemble projects and recurring contributors (e.g., Various Artists, Steven Epstein) reflect the collaborative nature of music production.
- Temporal analysis reveals a significant increase in award counts during the 2000s and 2010s, with peaks in 2009 and 2019. This suggests that the awards have evolved to include more categories and a broader range of contributions over time.
- The stabilization of category counts since 2018 indicates that the awards have reached a mature and consistent framework in recent years.

Insights on Recognition and Impact:

- The diverse and high-cardinality nature of categories, nominees, and artists not only enriches the dataset but also poses challenges for granular analysis.
- The analysis indicates that while some awards are strongly associated with individual achievements, others reflect the collaborative and ensemble aspects of music, a critical consideration for integrating this dataset with other sources (such as track-level data from platforms like Spotify).

Implications for Future Analysis:

- These findings lay a robust foundation for future studies, particularly for merging this awards data with commercial music data to examine how industry recognition correlates with audience metrics and musical trends.
- The insights suggest that future predictive models or trend analyses should account for both the quantitative (number of wins, category frequencies) and qualitative aspects (artist reputation, genre context) inherent in the Grammy Awards data.

Overall Summary:
The EDA of the Grammy Awards dataset highlights a rich, multifaceted landscape that documents the evolution of musical recognition over more than five decades. The dataset not only captures the dynamics of mainstream award categories but also reflects the growth of diverse and specialized categories. With detailed metadata on nominees, artists, and contributors, it provides a valuable resource for understanding industry trends, validating artistic impact, and exploring the relationship between critical acclaim and commercial success.

- # API Billboard structure and data insights

What is the Billboard Hot 100?

The Billboard Hot 100 is the definitive weekly chart that ranks the most popular songs in the United States by combining metrics from physical and digital sales, radio airplay, and streaming activity (e.g., YouTube, Spotify). This chart has been published weekly since 1958, making it one of the most important barometers of musical popularity in the world.

Why Analyze Billboard Data?

This project seeks to explore the full history of Billboard Hot 100 rankings—from its inception to today—to uncover long-term trends in music such as:

• The evolution of artist dominance across decades

• Longevity of songs on the charts

• Patterns in chart-topping hits

• Shifts in musical styles, collaborations, and industry dynamics

These insights will prove useful for musicologists, data analysts, industry professionals, and fans interested in how music success has evolved over time.

Dataset Source and Structure

Dataset Source:

Instead of scraping data directly from Billboard.com or using unstable third-party APIs, this project uses a public, open-source JSON dataset hosted on GitHub. This dataset, acting as a static API, provides comprehensive weekly chart data.

Source:

https://github.com/mhollingshead/billboard-hot-100

The dataset is obtained from the all.json file, which includes every weekly Billboard Hot 100 chart in history. An example structure is as follows:

Key Insights

- **Unprecedented Chart Longevity:**
  Certain songs, such as *Old Town Road* and *A Bar Song (Tipsy)*, tied records by spending 19 weeks at number one, underscoring their viral momentum. Additionally, *Blinding Lights* set a historic record with over 55 weeks in the Billboard Top 10.
- **Gradual Climbers vs. Instant Hits:**
  The dataset reveals distinct chart trajectories. Some tracks, like *Drivers License* and *7 Rings*, reach the top quickly, while others, such as *All I Want For Christmas Is You*, have a slow, enduring climb—sometimes taking years to achieve number one status.
- **Seasonal and Viral Chart Behavior:**
  Certain tracks display strong seasonal patterns, with *All I Want For Christmas Is You* reappearing on the chart annually. On the other hand, songs benefiting from social media virality, such as *Heat Waves* and *Levitating*, demonstrate the impact of digital trends on chart performance.
- **Artist Dominance and Career Longevity:**
  Artists like *Taylor Swift* dominate the dataset, with over 1,600 chart entries, reflecting both prolific output and sustained relevance. Other dominant figures include *Drake*, *Morgan Wallen*, *Elton John*, and *Madonna*, highlighting how both new and established artists maintain high visibility over decades.
- **Slow-Burn Success and Chart Re-entry:**
  Some tracks exhibit "sleeper hit" behavior—starting modestly on the charts and gradually ascending due to factors such as live performances, media exposure, or

social media support. Moreover, recurrent chart re-entries for classic hits indicate that songs can regain popularity years after their original release.

Final Conclusion

After conducting a detailed exploratory data analysis on the Billboard Hot 100 dataset, several critical insights have emerged regarding the evolution of musical success in the United States:

Dataset and Analysis Approach:

- The dataset provides a comprehensive record of weekly Billboard Hot 100 rankings since 1958.
- Using a public GitHub-hosted JSON file as a stable API source, we seamlessly integrated and analyzed historical chart data with consistent preprocessing and visualization methods.

Artist and Song Longevity:

- The analysis highlights varying trajectories of chart success—some songs become instant hits while others exhibit slow-burn popularity that endures over long periods.
- The unprecedented chart longevity observed in songs like *Blinding Lights* demonstrates the impact of modern streaming and social media platforms on maintaining prolonged chart presence.

Seasonal and Viral Phenomena:

- Seasonal patterns, exemplified by perennial favorites like *All I Want For Christmas Is You*, indicate that some songs are revived year after year, driven by cultural and seasonal factors.
- Viral trends contribute significantly to the chart performance of many modern hits, reflecting the complex interplay between fan engagement and digital media.

Dominance of Leading Artists:

- Artists such as *Taylor Swift* and *Drake* exemplify sustained relevance through consistent output and dynamic adaptability in a rapidly evolving industry.
- The data supports the understanding that musical success is no longer linear; rather, it is influenced by timing, innovation, and the power of digital networks in shaping audience behavior.

Implications for Future Studies:

- The insights from this analysis provide a solid foundation for further research into the dynamics of musical popularity. By integrating Billboard data with other datasets (such as Spotify tracks or Grammy awards), future studies can explore how industry recognition correlates with audience engagement and the evolution of musical trends.
- Understanding these long-term trends is critical for predicting future chart behavior and for stakeholders across the music industry—from record labels to independent artists—who seek to navigate an ever-changing musical landscape.

4. Extract Phase

4.1 Extracting Spotify Data (CSV)

- Reading and preprocessing the CSV file

```
def extract_spotify(**kwargs):
    df = pl.read_csv("/data/raw/spotify_dataset.csv")
    df.write_csv("/data/raw/spotify_dataset2.csv")
```

- **Purpose:** Reads the raw Spotify dataset CSV file using Polars and writes a processed copy to another CSV file.
- **How it works:**
    - Uses pl.read_csv() to load the dataset from the "/data/raw/spotify_dataset.csv" file.
    - Writes the loaded dataset to "/data/raw/spotify_dataset2.csv" for further processing or backup.
- **Usage Scenario:** This function is part of the "Extract Phase" and is triggered to standardize the raw extraction of Spotify data.

4.2 Extracting Grammy Data (Database)

- SQL queries to fetch Grammy data

```
def extract_grammys(**kwargs):
```

```
    load_dotenv(dotenv_path="/opt/airflow/.env")

    dbname = os.getenv("DB_NAME")
    user = os.getenv("DB_USER")
    password = os.getenv("DB_PASSWORD")
    host = os.getenv("DB_HOST")
    port = os.getenv("DB_PORT")

    conn = psycopg2.connect(
        dbname=dbname,
        user=user,
        password=password,
        host=host,
        port=port
    )

    cur = conn.cursor()
    cur.execute("SELECT * FROM public.grammy_awards;")
    columns = [desc[0] for desc in cur.description]
    rows = cur.fetchall()
    cur.close()
    conn.close()

    df = pl.DataFrame(rows, schema=columns)
    df.write_csv("/data/raw/grammy_awards_full.csv")
```

- **Purpose:** Connects to a PostgreSQL database to extract Grammy Awards data and saves it as a CSV file.

- **How it works:**

  - Loads environment variables from a `.env` file using `load_dotenv()` to obtain database credentials.
  - Establishes a connection using `psycopg2.connect()` with the loaded credentials.
  - Executes an SQL query (`SELECT * FROM public.grammy_awards;`) to fetch all rows from the Grammy awards table.
  - Retrieves the column names and rows, then creates a Polars DataFrame from the fetched data.
  - Writes the DataFrame to "/data/raw/grammy_awards_full.csv" for subsequent processing.

- **Usage Scenario:** This function is used in the Extract Phase to pull Grammy data directly from a database, ensuring current and complete data retrieval.

4.3 Extracting Data from API

- API request handling

```
def extract_billboard(**kwargs):
    url = "https://raw.githubusercontent.com/mhollingshead/billboard-hot-100/main/all.json"
    response = requests.get(url)

    if response.status_code == 200:
        charts = response.json()
        all_data = []

        for chart in charts:
            chart_date = chart["date"]
            for entry in chart["data"]:
                all_data.append({
                    "date": chart_date,
                    "rank": entry.get("this_week"),
                    "title": entry.get("song"),
                    "artist": entry.get("artist"),
                    "last_week": entry.get("last_week"),
                    "peak_position": entry.get("peak_position"),
                    "weeks_on_chart": entry.get("weeks_on_chart")
                })

        df = pd.DataFrame(all_data)
        df.to_csv("/data/raw/billboard_full_chart_data.csv", index=False)
        print("Billboard data extracted successfully!")

    else:
        print(f"Failed to fetch chart data. Status code: {response.status_code}")
```

- **Purpose:** Retrieves historical Billboard Hot 100 chart data from an online JSON file and stores it locally as a CSV.

- **How it works:**

  - Sends a GET request to a GitHub URL containing the static JSON dataset.
  - Checks if the HTTP response is successful (status code 200).

- Iterates over each chart entry in the JSON data, extracting fields such as date, rank, song title, artist, last week's position, peak position, and weeks on chart.
- Constructs a Python list of dictionaries with the extracted data, then converts it into a Pandas DataFrame.
- Saves the DataFrame to "/data/raw/billboard_full_chart_data.csv".
- Prints a success message or an error message if the request fails.

- **Usage Scenario:** This is part of the Extract Phase for retrieving and standardizing Billboard data that will later be transformed and merged with other datasets.

5. Transform Phase}

```python
def normalize_text(value):
    if isinstance(value, str):
        value = value.lower()
        value = re.sub(r"[^a-z0-9\s;]", "", value)
        value = re.sub(r"\s+", " ", value)
        return value.strip()
    return value
```

- **Purpose:** Standardizes text strings by converting them to lowercase, removing special characters, and trimming extra spaces.

- **How it works:**

  - Checks if the input value is a string.
  - Converts the string to lowercase.
  - Uses regular expressions to remove any characters that aren't alphanumeric or whitespace.
  - Collapses multiple spaces into a single space and trims leading or trailing spaces.

- **Usage Scenario:** This helper function is applied within transformation functions to ensure consistent text formatting across datasets (e.g., for song names and artist names).

```python
def transform_spotify(**kwargs):
    df = pl.read_csv(f"{DATA_RAW}/spotify_dataset2.csv")
    df = df.rename({col: col.strip().lower().replace(" ", "_") for col in
df.columns})

    df = df.with_columns([
```

```
        pl.col("track_name").apply(normalize_text, return_dtype=pl.Utf8),
        pl.col("artists").apply(normalize_text, return_dtype=pl.Utf8)
    ])

    df = df.select([
        pl.col("track_name").alias("song_name"),
        pl.col("artists").alias("artist"),
        "track_genre",
        "popularity",
        "explicit",
        "tempo",
        "valence",
        "energy",
        "danceability",
        "acousticness",
        "duration_ms"
    ])

    df = df.with_columns([
        (pl.col("duration_ms") / 60000).alias("duration_minutes")
    ])

    df.write_csv(f"{DATA_PROCESSED}/spotify_transformed.csv")
```

- **Purpose:** Cleans and normalizes the Spotify dataset for consistency and easier merging with other datasets.

- **How it works:**

    - Reads the intermediate CSV produced from the extraction phase.
    - Renames all columns by stripping extra spaces, converting to lowercase, and replacing spaces with underscores.
    - Applies the `normalize_text` function to the "track_name" and "artists" columns.
    - Selects a subset of relevant columns, renaming "track_name" to "song_name" and "artists" to "artist".
    - Adds a new column "duration_minutes" by converting "duration_ms" into minutes.
    - Writes the cleaned data to "/data/processed/spotify_transformed.csv".

- **Usage Scenario:** This function is used during the Transform Phase to prepare Spotify data for integration with other music datasets.

```
def transform_grammys(**kwargs):
```

```
    df = pl.read_csv(f"{DATA_RAW}/grammy_awards_full.csv")
    df = df.rename({col: col.strip().lower().replace(" ", "_") for col in
df.columns})

    df = df.with_columns([
        pl.col("nominee").apply(normalize_text, return_dtype=pl.Utf8),
        pl.col("artist").apply(normalize_text, return_dtype=pl.Utf8),
        pl.col("category").apply(normalize_text, return_dtype=pl.Utf8)
    ])

    df = df.filter(
        (pl.col("category").str.contains("song") |
pl.col("category").str.contains("record")) &
        ~(pl.col("category").str.contains("album") |
pl.col("category").str.contains("artist"))
    )

    df = df.with_columns([
        pl.col("nominee").alias("song_name")
    ])

    df.write_csv(f"{DATA_PROCESSED}/grammys_transformed.csv")
```

- **Purpose:** Normalizes and filters the Grammy Awards dataset to focus on song and record categories.

- **How it works:**

    - Reads the raw Grammy CSV file.
    - Standardizes column names as in the Spotify transformation.
    - Applies the `normalize_text` function to textual fields like "nominee", "artist", and "category".
    - Filters the dataset to include only rows where the category contains "song" or "record", excluding those that mention "album" or "artist" to focus on track-specific entries.
    - Renames the "nominee" column to "song_name" for consistency.
    - Writes the transformed data to "/data/processed/grammys_transformed.csv".

- **Usage Scenario:** This transformation refines the Grammy data, ensuring compatibility when merging datasets related to individual songs.

```
def transform_billboard(**kwargs):
```

```
    df = pl.read_csv(f"{DATA_RAW}/billboard_full_chart_data.csv")
    df = df.rename({col: col.strip().lower().replace(" ", "_") for col in
df.columns})

    df = df.with_columns([
        pl.col("title").apply(normalize_text, return_dtype=pl.Utf8),
        pl.col("artist").apply(normalize_text, return_dtype=pl.Utf8)
    ])

    df = df.with_columns([
        pl.col("title").alias("song_name"),
        pl.col("date").str.slice(0,
4).cast(pl.Int64).alias("first_year_on_chart")
    ])

    df.write_csv(f"{DATA_PROCESSED}/billboard_transformed.csv")
```

- **Purpose:** Processes and standardizes the Billboard Hot 100 data for downstream analysis.

- **How it works:**

    - Loads the raw Billboard chart data CSV.
    - Renames columns following the standard convention (lowercase and underscores).
    - Applies text normalization to the "title" (later renamed to "song_name") and "artist" columns.
    - Extracts the year from the "date" field (first four characters) and converts it into an integer as "first_year_on_chart".
    - Writes the cleaned data to "/data/processed/billboard_transformed.csv".

- **Usage Scenario:** Used in the Transform Phase to clean the Billboard dataset and extract additional fields required for merging with other datasets.

5.2 Merge Phase

```
def merge_datasets(**kwargs):
    df_spotify = pl.read_csv(f"{DATA_PROCESSED}/spotify_transformed.csv")
    df_grammys = pl.read_csv(f"{DATA_PROCESSED}/grammys_transformed.csv")
    df_billboard =
pl.read_csv(f"{DATA_PROCESSED}/billboard_transformed.csv")

    df_spotify = df_spotify.unique()
```

```python
df_grammys = df_grammys.unique()
df_billboard = df_billboard.unique()

df_spotify = df_spotify.with_columns([
    (pl.col("song_name") + "|" + pl.col("artist")).alias("merge_key")
]).sort("popularity", descending=True)

df_spotify = df_spotify.groupby("merge_key").agg([
    pl.col("song_name").first(),
    pl.col("artist").first(),
    pl.col("track_genre").unique().alias("track_genres"),
    pl.col("popularity").first(),
    pl.col("explicit").first(),
    pl.col("tempo").first(),
    pl.col("valence").first(),
    pl.col("energy").first(),
    pl.col("danceability").first(),
    pl.col("acousticness").first(),
    pl.col("duration_minutes").first()
])

df_spotify = df_spotify.with_columns([
    pl.col("track_genres").list.join(", ").alias("track_genre")
]).drop("track_genres")

df_grammys = df_grammys.sort("year", descending=True)
df_grammys = df_grammys.groupby("song_name").agg([
    pl.col("year").first(),
    pl.col("category").first(),
    pl.col("published_at").first()
])

df_billboard = df_billboard.with_columns([
    (pl.col("song_name") + "|" + pl.col("artist")).alias("merge_key")
])

df_billboard_grouped = df_billboard.groupby("merge_key").agg([
    pl.col("date").min().alias("first_chart_date"),
    pl.col("rank").min().alias("billboard_peak"),
    pl.col("weeks_on_chart").sum().alias("total_weeks_on_chart"),
    pl.col("first_year_on_chart").min()
])

merged = df_spotify.join(df_grammys, on="song_name", how="left")
merged = merged.join(df_billboard_grouped, on="merge_key", how="left")
```

```
    merged = merged.drop("merge_key")

    columns_to_drop = [
        "duration_ms",
        "nominee",
        "artist_grammy",
        "workers",
        "img",
        "winner",
        "first_year_on_chart",
        "billboard_artist",
        "updated_at",
        "title",
        "published_at"
    ]
    existing_to_drop = [col for col in columns_to_drop if col in
merged.columns]
    merged = merged.drop(existing_to_drop)

    merged.write_csv(f"{DATA_PROCESSED}/final_dataset.csv")
```

- **Purpose:** Combines the transformed datasets from Spotify, Grammy Awards, and Billboard into a single unified dataset.

- **How it works:**

    - Reads the processed CSV files for Spotify, Grammy, and Billboard datasets.
    - Removes duplicate rows using the unique() function for each dataset.
    - Creates a "merge_key" in the Spotify dataset by concatenating "song_name" and "artist" to facilitate the join with Billboard data.
    - Groups and aggregates the Spotify data by "merge_key" to collect unique genres and other attribute values.
    - Groups the Billboard data on the same "merge_key" to aggregate chart metrics like the first chart date, peak rank, and total weeks on chart.
    - Joins the Spotify dataset with the Grammy dataset on "song_name" (using a left join) and then joins the result with the Billboard grouped data on "merge_key".
    - Drops the "merge_key" and any extraneous columns not needed in the final output.
    - Writes the merged final dataset to "/data/processed/final_dataset.csv".

- **Usage Scenario:** This function integrates all three sources into one dataset for comprehensive analysis, enabling cross-referencing between streaming data, awards data, and historical chart performance.

6. Load Phase

```
def load_and_store_final_dataset(**kwargs):
    load_dotenv()

    dbname   = os.getenv("DB_NAME")
    user     = os.getenv("DB_USER")
    password = os.getenv("DB_PASSWORD")
    host     = os.getenv("DB_HOST")
    port     = os.getenv("DB_PORT")

    conn = psycopg2.connect(
        dbname=dbname,
        user=user,
        password=password,
        host=host,
        port=port
    )
    cur = conn.cursor()

    create_table_query = """
    CREATE TABLE IF NOT EXISTS public.music_dataset (
        song_name TEXT,
        artist TEXT,
        popularity INTEGER,
        explicit BOOLEAN,
        tempo FLOAT,
        valence FLOAT,
        energy FLOAT,
        danceability FLOAT,
        acousticness FLOAT,
        duration_minutes FLOAT,
        track_genre TEXT,
        year INTEGER,
        category TEXT,
        first_chart_date TEXT,
        billboard_peak INTEGER,
        total_weeks_on_chart INTEGER
    );
    """
    cur.execute(create_table_query)
    conn.commit()

    copy_query = """
    COPY public.music_dataset (
```

```
        song_name, artist, popularity, explicit, tempo, valence, energy,
        danceability, acousticness, duration_minutes, track_genre, year,
        category, first_chart_date, billboard_peak, total_weeks_on_chart
    )
    FROM STDIN WITH CSV HEADER DELIMITER ',' NULL ''
    """
    with open(CSV_FILE_PATH, 'r', encoding='utf-8') as f:
        cur.copy_expert(copy_query, f)

    conn.commit()
    cur.close()
    conn.close()
    print("Final dataset loaded into PostgreSQL.")

    gauth = GoogleAuth()
    gauth.LoadClientConfigFile("/client_secrets.json")
    gauth.LocalWebserverAuth()
    drive = GoogleDrive(gauth)

    file_to_upload = drive.CreateFile({'title': CSV_FILENAME})
    file_to_upload.SetContentFile(CSV_FILE_PATH)
    file_to_upload.Upload()
    print("Final dataset uploaded to Google Drive.")
```

- **Purpose:** Loads the final merged dataset into a PostgreSQL database and uploads the CSV file to Google Drive for backup and sharing.

- **How it works:**

    - Loads database credentials from environment variables using `load_dotenv()`.
    - Establishes a connection to PostgreSQL with `psycopg2.connect()`.
    - Creates a table (`public.music_dataset`) if it does not already exist, defining columns that match the final merged dataset.
    - Uses the PostgreSQL `COPY` command (via `copy_expert`) to load data from the CSV file into the database table.
    - Commits the transaction, closes the connection, and prints a confirmation message.
    - Authenticates with Google Drive using PyDrive, loads client configuration from a secrets file, and initiates a local web server authentication.
    - Creates a file on Google Drive with the CSV content and uploads it, printing a confirmation on success.

- **Usage Scenario:** This function is part of the Load Phase, ensuring that the integrated dataset is stored securely in a database for further querying and is also backed up to Google Drive.

7. Airflow Pipeline Implementation

This section documents the configuration and execution of the Airflow pipeline for the ETL process. The implementation runs on Apache Airflow 2.6.2 using a Docker Compose setup based on version 3.8. The pipeline integrates tasks for extraction, transformation, merging, and loading data from different sources (Spotify, Grammys, and Billboard).

The Docker Compose file sets up the necessary services: PostgreSQL (for the Airflow metadata and data storage), Redis (as the Celery broker), and several Airflow components (webserver, scheduler, worker, initialization, and Flower for monitoring Celery tasks). The configuration specifies environment variables, volumes for persistent storage (such as dags, logs, and data), and dependency conditions to ensure that Redis and PostgreSQL are healthy before Airflow services start.

Key Elements:

- **version: '3.8'**
  Specifies the Docker Compose version used.
- **Common Airflow Configuration (x-airflow-common):**
  - Uses the Apache Airflow 2.6.2 Docker image and custom Dockerfile.
  - Sets environment variables for the CeleryExecutor, database connection, result backend, and broker URL.
  - Mounts volumes for DAGs, logs, plugins, configuration, data, source code, and required credential files (such as .env and client_secrets.json).
  - Specifies user IDs to avoid permission issues and depends on Redis and PostgreSQL services.
- **Service Definitions:**
  - **postgres**: Runs a PostgreSQL instance with credentials set to "airflow". It uses a healthcheck command to ensure the database is ready.
  - **redis**: Runs a Redis container with the required healthcheck for connectivity.
  - **airflow-webserver**, **airflow-scheduler**, **airflow-worker**: Utilize the common configuration to run the web interface, scheduler, and Celery workers respectively.
  - **airflow-init**: Installs requirements, upgrades the Airflow database, and creates an admin user.

- **flower**: Provides a monitoring dashboard for Celery tasks.

```
x-airflow-common: &airflow-common
  image: apache/airflow:2.6.2
  build:
    context: .
    dockerfile: Dockerfile
  environment:
    &airflow-common-env
    AIRFLOW__CORE__EXECUTOR: CeleryExecutor
    AIRFLOW__DATABASE__SQL_ALCHEMY_CONN:
postgresql+psycopg2://airflow:airflow@postgres/airflow
    AIRFLOW__CELERY__RESULT_BACKEND:
db+postgresql://airflow:airflow@postgres/airflow
    AIRFLOW__CELERY__BROKER_URL: redis://:@redis:6379/0
    AIRFLOW__CORE__FERNET_KEY: ''
    AIRFLOW__CORE__DAGS_ARE_PAUSED_AT_CREATION: 'true'
    AIRFLOW__CORE__LOAD_EXAMPLES: 'true'
    AIRFLOW__API__AUTH_BACKENDS: 'airflow.api.auth.backend.basic_auth'
  volumes:
    - ./dags:/opt/airflow/dags
    - ./logs:/opt/airflow/logs
    - ./plugins:/opt/airflow/plugins
    - ./config:/config
    - ./data:/data
    - ./db_models:/db_models
    - ./src:/opt/airflow/src
    - ./requirements.txt:/requirements.txt
    - .env:/opt/airflow/.env
    - ./client_secrets.json:/client_secrets.json
  user: "${AIRFLOW_UID:-50000}:${AIRFLOW_GID:-50000}"
  depends_on:
    redis:
      condition: service_healthy
    postgres:
      condition: service_healthy

services:
  postgres:
    image: postgres:15
    environment:
```

```
    POSTGRES_USER: airflow
    POSTGRES_PASSWORD: airflow
    POSTGRES_DB: airflow
  volumes:
    - postgres-db-volume:/var/lib/postgresql/data
  healthcheck:
    test: ["CMD", "pg_isready", "-U", "airflow"]
    interval: 5s
    retries: 5
  restart: always

redis:
  image: redis:latest
  ports:
    - 6379:6379
  healthcheck:
    test: ["CMD", "redis-cli", "ping"]
    interval: 5s
    timeout: 30s
    retries: 50
  restart: always

airflow-webserver:
  <<: *airflow-common
  command: webserver
  ports:
    - 8080:8080
  healthcheck:
    test: ["CMD", "curl", "--fail", "http://localhost:8080/health"]
    interval: 10s
    timeout: 10s
    retries: 5
  restart: always

airflow-scheduler:
  <<: *airflow-common
  command: scheduler
  restart: always

airflow-worker:
  <<: *airflow-common
  command: celery worker
  restart: always

airflow-init:
```

```
    <<: *airflow-common
    command: bash -c "pip install -r /requirements.txt && airflow db upgrade
&& airflow users create --username admin --password admin --firstname Admin
--lastname Admin --role Admin --email admin@example.com"
    environment:
      <<: *airflow-common-env
      _AIRFLOW_DB_UPGRADE: 'true'
      _AIRFLOW_WWW_USER_CREATE: 'true'
      _AIRFLOW_WWW_USER_USERNAME: ${_AIRFLOW_WWW_USER_USERNAME:-airflow}
      _AIRFLOW_WWW_USER_PASSWORD: ${_AIRFLOW_WWW_USER_PASSWORD:-airflow}

  flower:
    <<: *airflow-common
    command: celery flower
    ports:
      - 5555:5555
    healthcheck:
      test: ["CMD", "curl", "--fail", "http://localhost:5555/"]
      interval: 10s
      timeout: 10s
      retries: 5
    restart: always

volumes:
  postgres-db-volume:
```

7.2 DAG Implementation

The DAG (Directed Acyclic Graph) in Airflow defines the ETL pipeline tasks and their dependencies. The code below is placed in a file within the `dags` directory, ensuring Airflow automatically detects and executes it.

Key Aspects of the DAG Code:

- **Importing Required Modules:**
  The code starts by importing the necessary modules such as `DAG` and `PythonOperator` from Airflow. It also imports utility functions from the `src` directory, ensuring the processing functions (extraction, transformation, merging, and load) are available within the Airflow environment.
- **Logging Configuration:**
  A custom `log_task` function wraps each Python callable to log the start and end of every task, which aids in debugging and monitoring.

- **DAG Definition:**
  The DAG is defined with an ID ("etl_music_pipeline"), default arguments (owner, start date, retry count), and the scheduling interval set to daily (`@daily`). Catchup is disabled to run only the current schedule.
- **Task Definitions:**
  Tasks are created using `PythonOperator`:
  - **Extraction Tasks:**
    - `extract_spotify`
    - `extract_grammys`
    - `extract_billboard`
  - **Transformation Tasks:**
    - `transform_spotify`
    - `transform_grammys`
    - `transform_billboard`
  - **Merge Task:**
    - `merge_datasets`
  - **Load Task:**
    - `load_and_store_final_dataset`

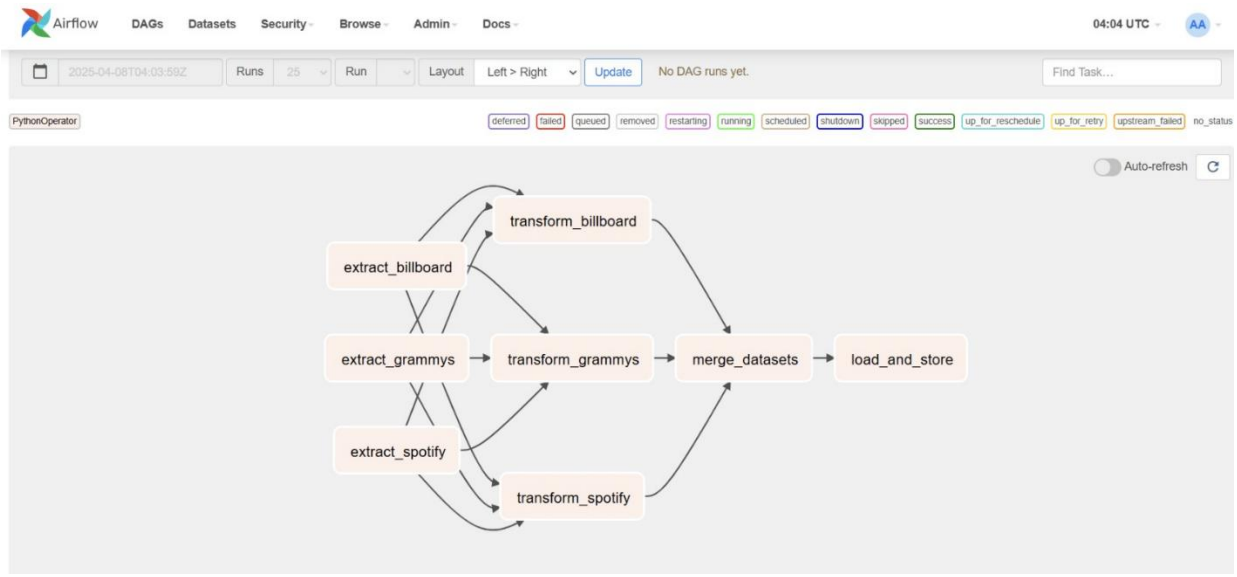
  Dependencies are set such that:

  - Each extraction task feeds into all transformation tasks.
  - All transformation tasks must complete before the merge task.
  - Finally, the merge task triggers the load task.

```python
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime
import logging
import sys

# Para que Airflow encuentre el módulo src dentro del contenedor
sys.path.append("/opt/airflow")

# === IMPORTACIÓN DE FUNCIONES DEL PIPELINE ===
from src.extract import extract_spotify, extract_grammys, extract_billboard
from src.transform import transform_spotify, transform_grammys,
transform_billboard
from src.merge_data import merge_datasets
from src.load import load_and_store_final_dataset

# === LOGGING CONFIGURATION ===
logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)
```

```python
def log_task(task_name, func):
    def wrapper(**kwargs):
        logger.info(f"[START] Task: {task_name}")
        result = func(**kwargs)
        logger.info(f"[END] Task: {task_name}")
        return result
    return wrapper

def return_none_wrapper(func):
    def wrapper(**kwargs):
        func(**kwargs)
        return None
    return wrapper

with DAG(
    dag_id="etl_music_pipeline",
    default_args={
        "owner": "airflow",
        "start_date": datetime(2024, 3, 21),
        "retries": 1,
    },
    description="ETL pipeline for music data from Spotify, Grammys, and
Billboard",
    schedule_interval="@daily",
    catchup=False,
) as dag:

    task_extract_spotify = PythonOperator(
        task_id="extract_spotify",
        python_callable=log_task("extract_spotify",
return_none_wrapper(extract_spotify)),
    )

    task_extract_grammys = PythonOperator(
        task_id="extract_grammys",
        python_callable=log_task("extract_grammys",
return_none_wrapper(extract_grammys)),
    )

    task_extract_billboard = PythonOperator(
        task_id="extract_billboard",
        python_callable=log_task("extract_billboard",
return_none_wrapper(extract_billboard)),
    )
```

```
    task_transform_spotify = PythonOperator(
        task_id="transform_spotify",
        python_callable=log_task("transform_spotify",
return_none_wrapper(transform_spotify)),
    )

    task_transform_grammys = PythonOperator(
        task_id="transform_grammys",
        python_callable=log_task("transform_grammys",
return_none_wrapper(transform_grammys)),
    )

    task_transform_billboard = PythonOperator(
        task_id="transform_billboard",
        python_callable=log_task("transform_billboard",
return_none_wrapper(transform_billboard)),
    )

    task_merge = PythonOperator(
        task_id="merge_datasets",
        python_callable=log_task("merge_datasets",
return_none_wrapper(merge_datasets)),
    )

    task_load = PythonOperator(
        task_id="load_and_store",
        python_callable=log_task("load_and_store_final_dataset",
return_none_wrapper(load_and_store_final_dataset)),
    )

    # Dependencias definidas con bucles (compatibles con tu entorno)
    extract_tasks = [task_extract_spotify, task_extract_grammys,
task_extract_billboard]
    transform_tasks = [task_transform_spotify, task_transform_grammys,
task_transform_billboard]

    for extract_task in extract_tasks:
        extract_task >> transform_tasks

    for transform_task in transform_tasks:
        transform_task >> task_merge

    task_merge >> task_load
```



8. Visualization & Graphs

FINAL MERGED DATASET OVERVIEW

Purpose:
This final merged dataset integrates music data from three sources—Spotify, Grammy Awards, and Billboard Hot 100—to provide a comprehensive view of each song's performance and recognition in the music industry. The dataset is designed to enable analysis of:

- The overlap between streaming popularity (Spotify), chart performance (Billboard), and award recognition (Grammys).
- Trends in musical features, genre distributions, and artist impact.
- Insights for developing recommendation systems, trend analysis tools, or business intelligence dashboards.

Dataset Structure:

- **Base Dataset:**
  The core of the final dataset comes from the Spotify data, ensuring that all songs in the dataset are maintained.
- **Joined Data:**
  - **Grammys Data:** Joined on a standardized key (song_name) derived from the nominee field, adding award-related metadata (e.g., category, year).

    - **Billboard Data:** Joined on the same standardized key to include chart performance metrics (e.g., billboard peak, total weeks on chart, and first chart year).

Key Fields in the Final Dataset:

- **song_name:**
  The standardized name of the song (cleaned and normalized from the source fields).
- **artist:**
  The primary performer(s) of the song, derived and normalized across the datasets.
- **track_genre:**
  The musical genre as captured in the Spotify dataset.
- **popularity:**
  A numerical score (e.g., 0–100) representing the song's streaming popularity.
- **tempo:**
  The tempo of the song in beats per minute, providing insight into the energy of the track.
- **nominee:**
  The song name as recorded in the Grammy Awards data (if available).
- **category:**
  The award category from the Grammy dataset, such as "Record of the Year" or "Song of the Year" (if applicable).
- **grammy_year:**
  The year the song was nominated or won a Grammy award (if applicable).
- **billboard_peak:**
  The highest position the song reached on the Billboard Hot 100 chart (if available).
- **weeks_on_chart:**
  The total number of weeks the song spent on the Billboard Hot 100 chart (if available).
- **first_year_on_chart:**
  The first year the song appeared on the Billboard Hot 100 chart, representing its debut period (if available).

Dashboard

Dashboard Technologies Overview

The music industry dashboard was developed using a combination of modern, open-source technologies for data visualization and web application development. The key technologies utilized include:

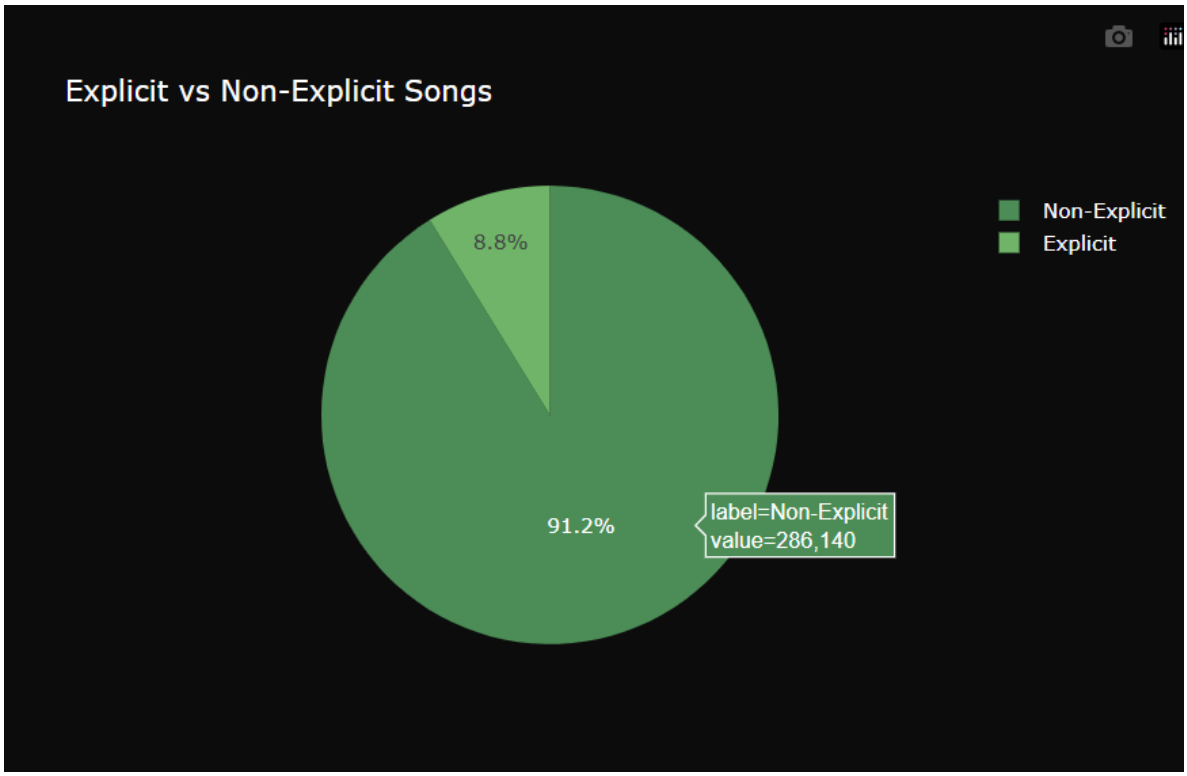
- **Python:**
  Python is used as the primary programming language for data processing, analysis,

and dashboard development. Its rich ecosystem of libraries makes it an ideal choice for handling large datasets and building interactive applications.

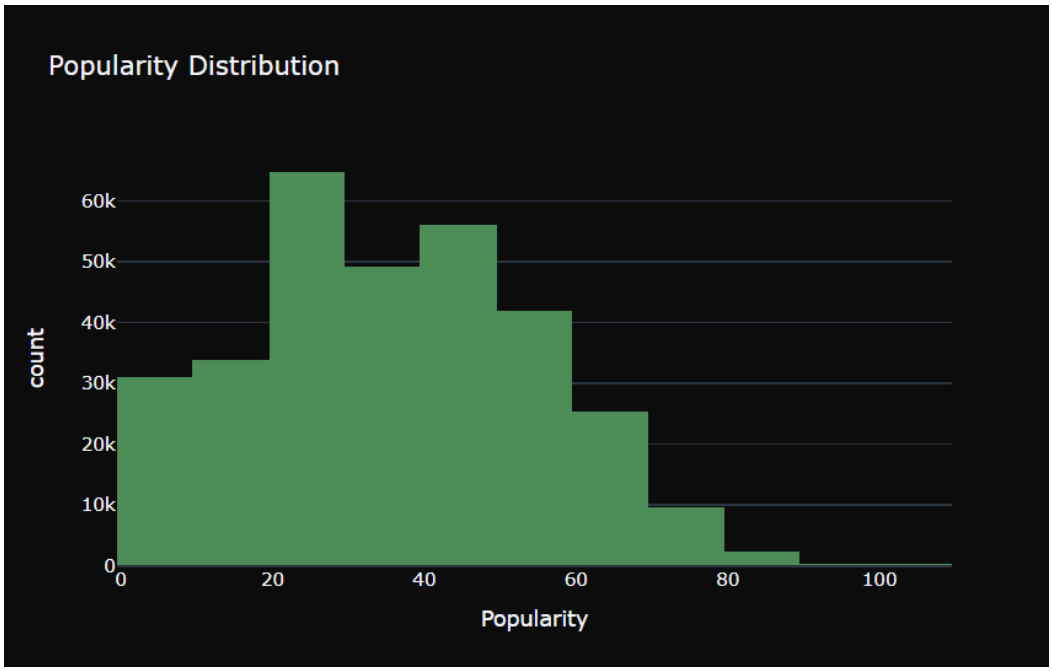
- **Dash:**
  Dash is a productive Python framework for building web-based analytic applications. It leverages Plotly for interactive visualizations and is designed to integrate seamlessly with Python code. With Dash, the dashboard can display real-time updates, interactive charts, and custom layouts, making the data exploration process highly engaging.
- **Plotly Express & Plotly Graph Objects:**
  Plotly provides both high-level (Plotly Express) and detailed (Plotly Graph Objects) interfaces for creating interactive and visually appealing graphics. These libraries are used extensively throughout the dashboard to produce a variety of charts—such as pie charts, histograms, bar charts, treemaps, and heatmaps—ensuring that visualizations are both informative and aesthetically consistent with a dark theme.
- **SQLAlchemy:**
  SQLAlchemy is used for database interaction. It provides a flexible and efficient way to create a connection to a PostgreSQL database, execute SQL queries, and retrieve the final merged dataset into Pandas DataFrames for visualization. This abstraction simplifies the process of working with relational databases in Python.
- **PostgreSQL:**
  PostgreSQL is the relational database chosen for storing and managing the final merged dataset. It offers robust performance, advanced features, and reliability, making it a preferred option for handling complex datasets in data-driven applications.
- **Pandas & NumPy:**
  These libraries are used for data manipulation and numerical operations. Pandas simplifies data handling by converting SQL query results into DataFrames, which are then used to prepare data for visualization. NumPy supports efficient numerical computations that are required for processing data before plotting.
- **Dash HTML Components (dcc, html):**
  Dash's HTML components provide the building blocks for the dashboard layout. They are used to structure the user interface by incorporating headers, containers, and other layout elements, ensuring that each visualization is neatly embedded within a styled container.

Graphs

1. **Explicit vs Non-Explicit Pie Chart:**
   - **Data:** Counts from the `explicit` column of the dataset.
   - **Graph:** A pie chart created using Plotly Express that maps explicit songs to a specific color sequence (`COLOR_SEQ`).
   - **Customization:** Uses a dark template with a custom background for consistent styling.

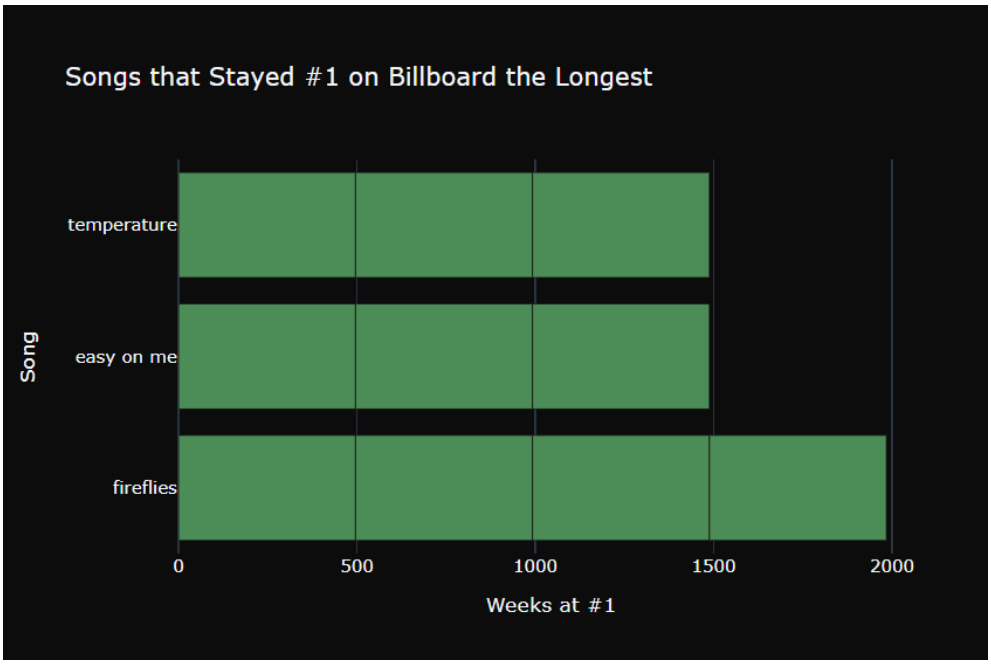


2. **Popularity Distribution Histogram:**
   - **Data:** The `popularity` field is histogrammed with 20 bins.
   - **Graph:** A histogram is produced with a dark theme to highlight the distribution of song popularity.

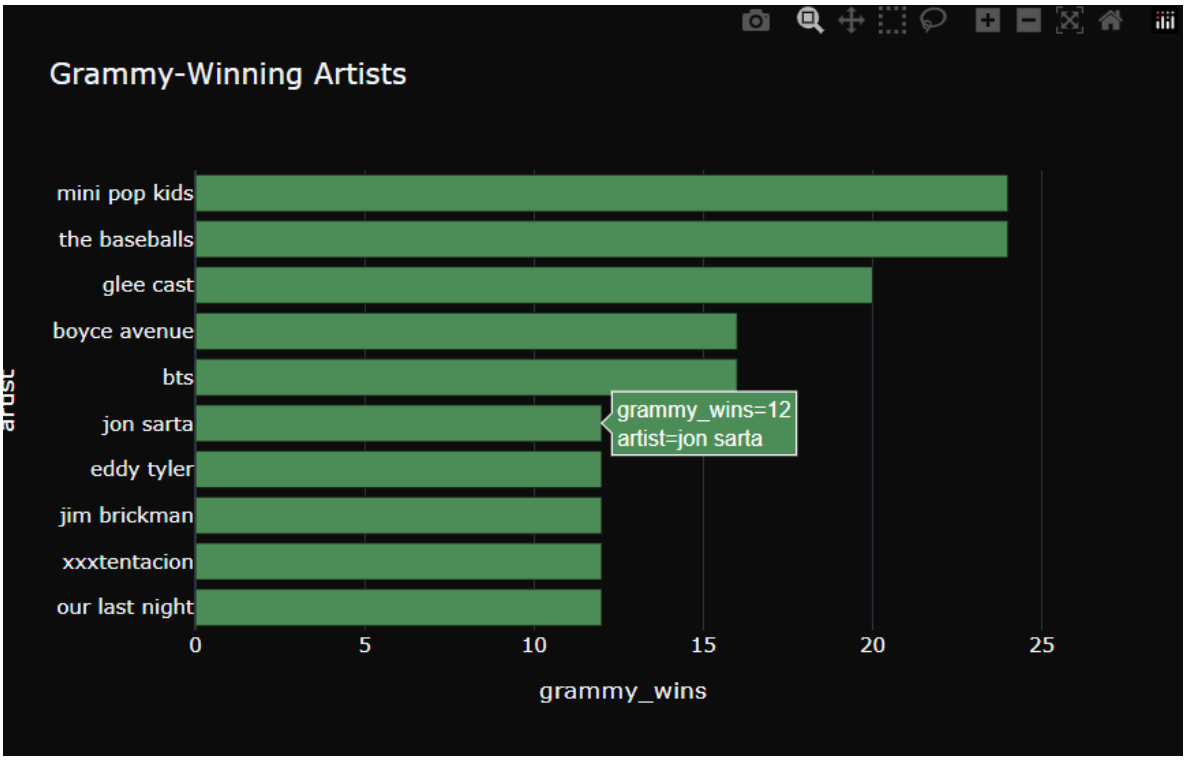


3. **Songs with Most Weeks on Billboard Bar Chart:**

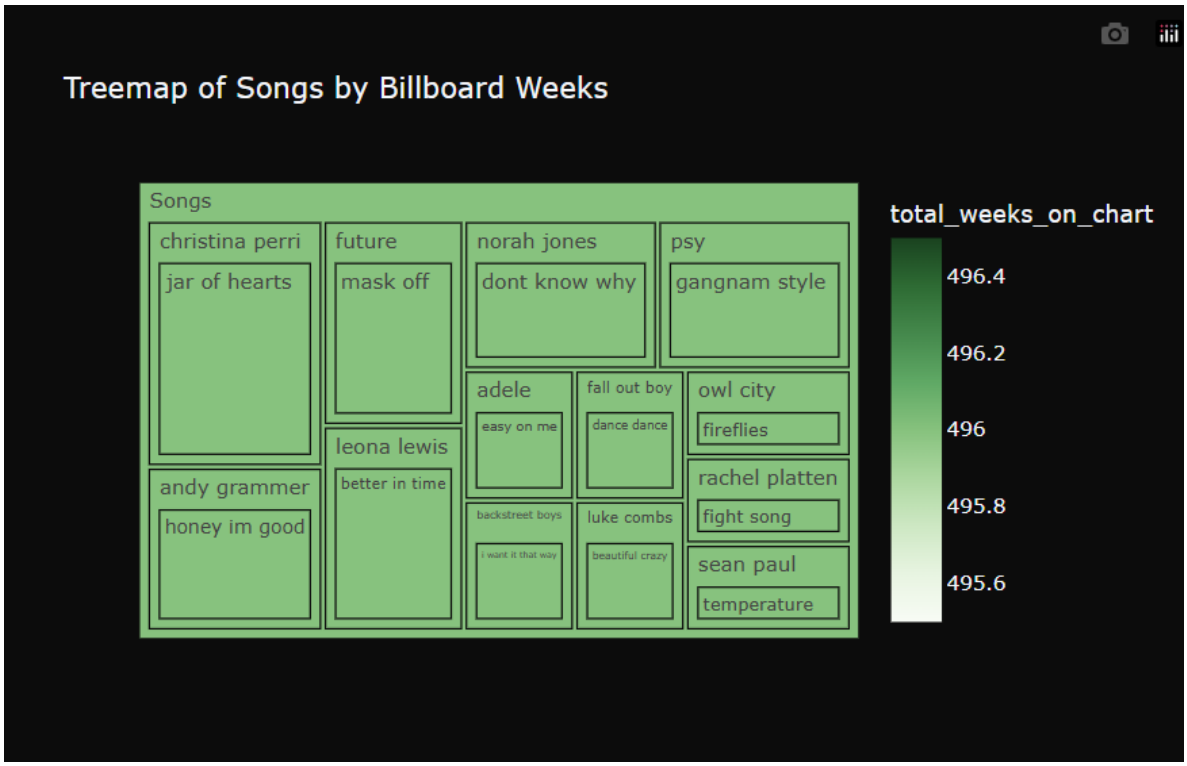
- **Data:** Top 20 songs sorted by `total_weeks_on_chart` (excluding extreme outliers).
- **Graph:** A horizontal bar chart shows the songs with the highest cumulative weeks on the Billboard chart.



4. **Top Grammy-Winning Artists Bar Chart:**
   - **Data:** Aggregation of Grammy-winning counts by artist, showing the top 10.
   - **Graph:** A horizontal bar chart displays the count of Grammy wins per artist.

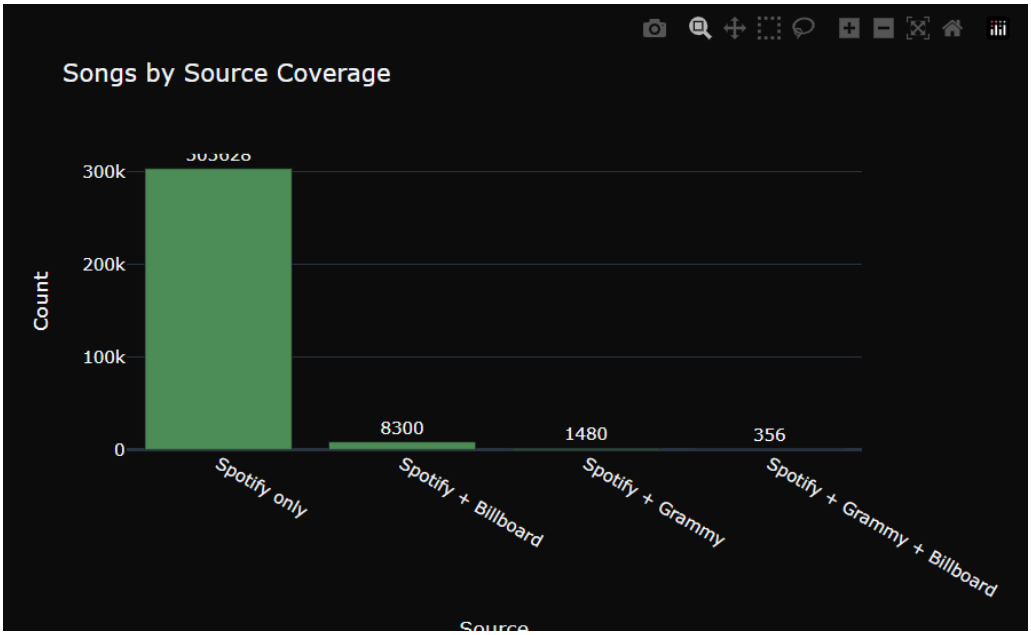


5. **Treemap of Billboard Songs:**
    - **Data:** A treemap using the top Billboard songs, grouped by artist.
    - **Graph:** Visualizes the distribution of `total_weeks_on_chart` across songs.

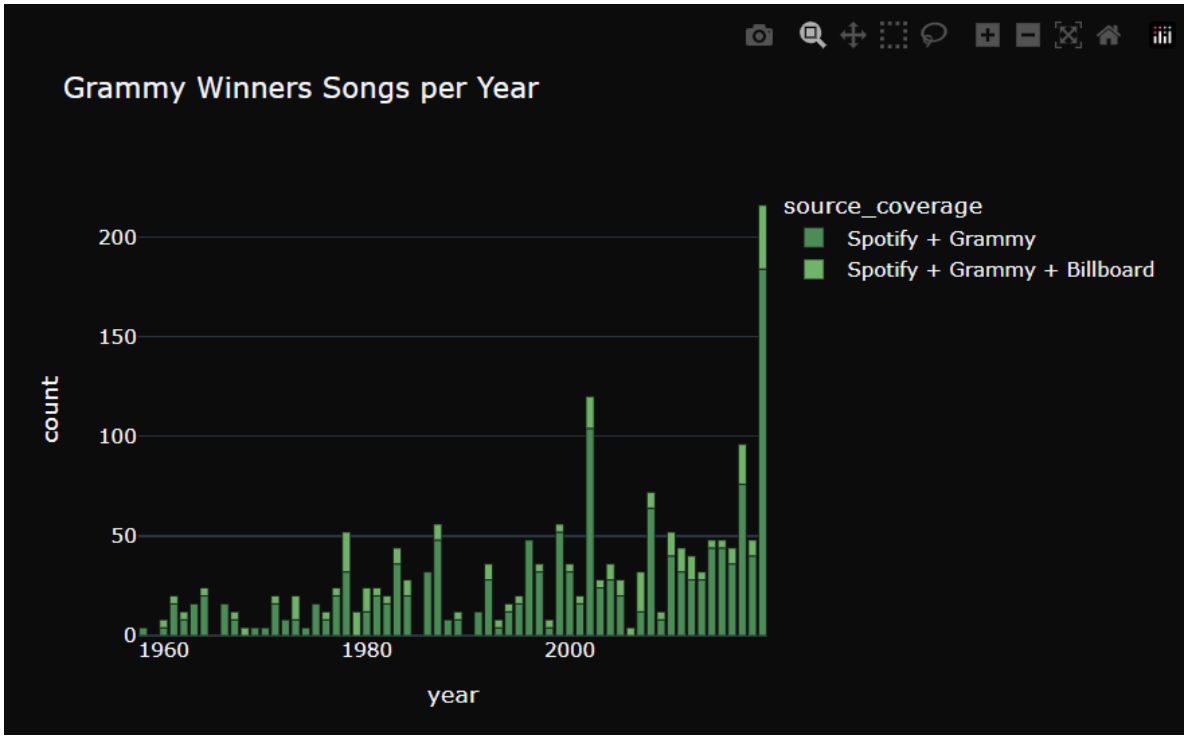


6. **Average Danceability by Genre Bar Chart:**
    - **Data:** Mean danceability grouped by `track_genre`.
    - **Graph:** A horizontal bar chart showing which genres have higher average danceability.

7. **Source Coverage Bar Chart:**



8. **Grammy Winners per Year Bar Chart:**



Dashboard Layout with Dash

To provide an interactive user interface, a Dash application is created. Two helper functions define styled containers for embedding each Plotly graph within the dashboard.

- **styled_container(graph):**
  Returns a Dash `Div` containing a Graph component with defined width, padding, background color, border, and border-radius for a neat and uniform appearance.
- **full_width_container(graph):**
  Similar to the above but spans 96% width for full-width visualizations, such as the correlation heatmap.

The dashboard layout is defined with a series of HTML Div blocks that embed the graphs. Multiple rows are created by grouping related visualizations side by side, and one full-width container is used for the correlation heatmap.

Usage Scenario:

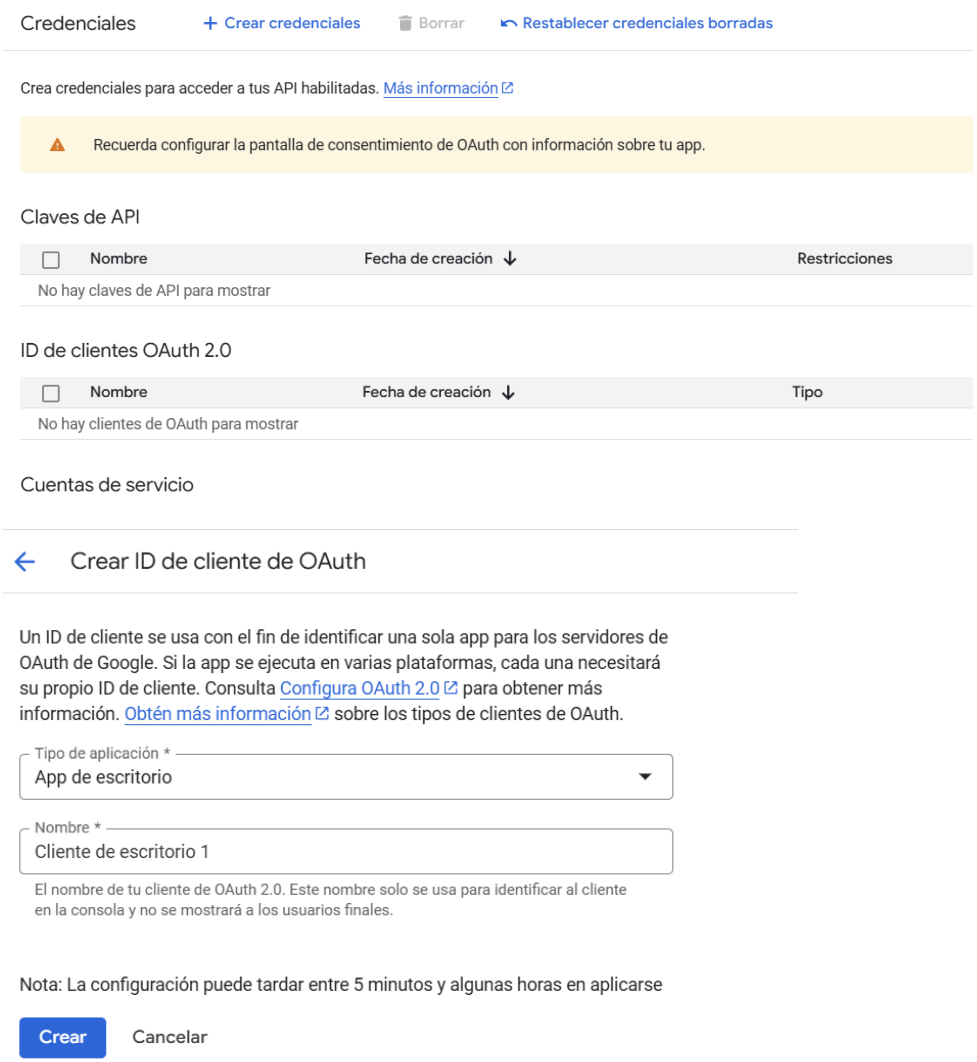
- The dashboard connects to the PostgreSQL database, retrieves the final merged dataset, and produces interactive visualizations that help in exploring various aspects of the music data, including explicit content, popularity trends, chart performance, and award recognition.

- The use of a dark theme (consistent with the visual style used in Plotly graphs) ensures a visually appealing presentation with high contrast and readability.

9 Credentials and Google Drive API Setup

To enable the upload of the final dataset to Google Drive:

1. **Download Google Cloud Credentials:**
    - Go to Google Cloud Console.
    - Create a new project or select an existing one.
    - Enable the Google Drive API.
    - Create OAuth client credentials and download the `client_secrets.json` file.



- Place `client_secrets.json` in the project root (or the path specified in the Docker Compose file).

2. **Configure Environment Variables:**
   - Create a `.env` file in the project root with your database credentials and any other required variables:

     ```
     DB_NAME=your_database_name
     DB_USER=your_database_user
     DB_PASSWORD=your_database_password
     DB_HOST=your_database_host
     DB_PORT=your_database_port
     ```

   - Include any other necessary configurations.

9.6. Setup and Usage Instructions

A. Setting Up the Python Virtual Environment

1. **Create a Virtual Environment:**

   ```
   python3 -m venv venv
   ```

2. **Activate the Virtual Environment:**
   - On Linux/Mac:

     ```
     source venv/bin/activate
     ```

   - On Windows:

     ```
     venv\Scripts\activate
     ```

3. **Install Python Dependencies:**

   ```
   pip install -r requirements.txt
   ```

B. Deploying the Project with Docker Compose

1. **Ensure Docker and Docker Compose are installed.**
2. **Build and Start the Containers:** From the project root directory, run:

   ```
   docker-compose up --build
   ```

   This command starts Airflow along with PostgreSQL, Redis, and other required services.

3. **Access the Airflow UI:** Open your browser and navigate to:

   `http://localhost:8080`

   Use the admin credentials created during initialization (default username: `airflow`, password: `airflow`) to log in.

4. **Trigger the ETL Pipeline:** In the Airflow UI, locate the DAG named `etl_music_pipeline` and trigger it manually or wait for the scheduled run.

Running the Dashboard

1. **Launch the Dashboard:** With your virtual environment activated, execute:
2. **Access the Dashboard:** Open your browser and go to:

   `http://localhost:8050`

   Explore interactive visualizations covering explicit content, popularity trends, chart performance, and award metrics.

10. Conclusion & Next Steps

KEY INSIGHTS

1. Comprehensive Data Integration:
   The project successfully combines data from three diverse sources—Spotify, Grammy Awards, and Billboard Hot 100—into a unified dataset. This integration enables cross-analysis of streaming performance, award recognition, and chart success, providing a holistic view of musical impact.
2. Standardization and Data Quality:
   Rigorous extraction, transformation, and merging processes ensure data is consistently formatted across all sources. Text normalization, consistent column naming, and conversion of data types (such as song duration) result in high data quality and facilitate reliable analysis.
3. Richness of Music Metrics:
   The final dataset encompasses detailed metrics including popularity scores, tempo, explicit content indicators, Grammy categories and years, Billboard peak positions, and weeks on chart. These metrics offer a deep understanding of various factors that contribute to a song’s overall success.

4. Insights into Genre and Artist Trends:
   Analysis of the unified dataset reveals clear trends in musical genres and artist performance. The data highlights how different factors—such as artist prominence, chart longevity, and award wins—intersect to drive success. Patterns uncovered indicate that the integration of streaming popularity, chart performance, and award data can shed light on the evolving dynamics of the music industry.
5. Enhanced Understanding Through Visualization:
   The interactive dashboard, developed with Dash and Plotly, presents visual insights through various charts and graphs (such as pie charts, histograms, bar charts, treemaps, and heatmaps). These visualizations simplify the interpretation of complex data, making it easier for analysts and industry stakeholders to identify key trends and support decision-making.

CONCLUSION

This project demonstrates the power of a fully integrated ETL pipeline applied to complex music data. By merging Spotify streaming metrics, Billboard chart performance, and Grammy Awards data into a single comprehensive dataset, the project provides a multi-dimensional view of musical success.

Key outcomes include:

- A robust framework that ensures high data quality and consistency, enabling reliable comparisons across data sources.
- Insights into the correlations between commercial success (streaming and charts) and critical recognition (awards), offering valuable information for artists, producers, and industry professionals.
- An interactive dashboard that empowers users to dynamically explore trends and patterns, supporting both strategic planning and detailed analytical research.

Overall, the project lays a strong foundation for advanced analytics, predictive modeling, and the development of recommendation systems in the music industry. It highlights the interplay between various success factors and paves the way for more nuanced, data-driven strategies in an evolving musical landscape.

BIBLIOGRAPHY

1. **Dash Framework:**
   - Official Documentation: https://dash.plotly.com/
     Explains how Dash is used to create interactive web applications using Python, integrating Plotly charts for real-time data visualization.
2. **Plotly:**
   - Plotly Python Open Source Graphing Library: https://plotly.com/python/
     Provides comprehensive guidance on creating interactive and animated plots with Plotly, including high-level interfaces like Plotly Express.

3. **Apache Airflow:**
    - Apache Airflow Documentation: https://airflow.apache.org/
      Describes how Airflow manages workflows and DAGs, supports scheduling, and facilitates robust ETL pipeline orchestration.
4. **Docker Compose:**
    - Docker Compose Documentation: https://docs.docker.com/compose/
      Offers detailed instructions on how Docker Compose is used to deploy multi-container Docker applications, including configurations for Airflow, PostgreSQL, and Redis.
5. **SQLAlchemy:**
    - SQLAlchemy Documentation: https://docs.sqlalchemy.org/
      Explains the usage of SQLAlchemy for building database connections and executing SQL queries within a Python environment.